Loughborough University of Technology

NON-CONVENTIONAL CONTROL OF THE FLEXIBLE POLE-CART BALANCING PROBLEM

By

Elmer P. Dadios

BSEE, MSCS

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy

of the Loughborough University of Technology

February 1996

TO MY BELOVED

wife: tita

&

children: leo, zoula, maymay, and e.j.

SUMMARY

Emerging techniques of intelligent or learning control seem attractive for applications in manufacturing and robotics. It is however important to understand the capabilities of such control systems. In the past the inverted pendulum has been used as a test case.

The thesis begins with an examination of whether the inverted pendulum or pole-cart balancing problem is a representative problem for experimentation for learning controllers for complex nonlinear systems. Results of previous research concerning the inverted pendulum problem are presented to show that this problem is not sufficiently testing.

This thesis therefore concentrates on the control of the inverted pendulum with an additional degree of freedom as a testing demonstrator problem for learning control system experimentation. A flexible pole is used in place of a rigid one. The transverse displacement of the flexible pole adds a degree of freedom to the system. The dynamics of this new system are more complex as the system needs additional parameters to be defined due to the pole's elastic deflection. This problem also has many of the significant features associated with flexible robots with lightweight links as applied in manufacturing.

Novel neural network and fuzzy control systems are presented that control such a system both in simulation and real time. A fuzzy-genetic approach is also demonstrated that allows the creation of fuzzy control systems without the use of extensive knowledge.

ACKNOWLEDGEMENTS

I would like to thank my mentor Prof. David J. Williams for the advice, help, and friendship which he has given me over the long years which I have spent on this research.

I would like to acknowledge Quanzer Consulting Incorporated for their contributions to the design of the experimental system, the Department of Science and Technology - Philippines and the De la Salle University - Manila for giving financial assistance for my Ph.D. studies. Also, I would like to thank my friends Teody, Gwen, Homer, Jimmy, Alfred, John, Kaweh, Tien, Mei Ling, Ming and the LUT Manufacturing Engineering laboratory personnel for their support.

Finally, I am very grateful to my wife, Tita, for understanding my work and having enough patience to care for our four children (Leo, Zoula, Maymay, and EJ), to them I dedicate this thesis.

ABBREVIATIONS

NN	Neural network.
FLS	Fuzzy logic systems.
PD	Proportional plus derivative control.
e(t)	Steady state error.
t	Time
P	Proportional control.
PI	Proportional plus integral control.
PID	Proportional plus integral pus derivative control.
$\mathbf{x}(k) = \dot{x}(t)$	State equation.
$\mathbf{y}(k)$	Output vector.
$\mathbf{u}(k)$	Input vector.
$\mathbf{G}(k) = \mathbf{A}(t)$	State matrix.
$\mathbf{H}(k) = \mathbf{B}(t)$	Input matrix.
$\mathbf{C}(k)$	Output matrix.
$\mathbf{D}(k)$	Direct transmission matrix.
o(t)	State of the plant.
i(t)	Input of the plant.
p(t)	Performance vector.
a(t)	Weighting input.
m(t)	Measuring device.
ADALINE	Adaptive linear element.
ACE	Adaptive critic element.
ASE	Adaptive search element.
BOXES	A system that learned to control an inverted pendulum using a state space representation of discrete regions.

SG(t)	A subgoal that represents a weighted vector distance of the state of the system at time *t* from the origin of the state space.
AI	Artificial Intelligence.
PC	Personal computer.
I/O	Input-output.
DC	Direct current.
UDC	Up down counter.
IBM	A computer trademark.
m_p	Mass of the pole.
m_c	Mass of the cart.
a_c	Acceleration of the cart.
a^B	Acceleration of the pole.
L	Total length of the pole.
g	Acceleration due to gravity
e_t	A unit vector for tangential component.
e_n	A unit vector for normal component.
a_t	Tangential acceleration.
a_n	Normal acceleration.
x	Displacement of the cart.
$\dot{x} = dx/dt$	Velocity of the cart.
$\theta = \theta_r$	Rigid pole angle.
$\dot{\theta} = \dot{\theta}_r$	Rigid pole angle velocity
$\ddot{\theta} = \ddot{\theta}_r$	Rigid pole angular acceleration.
θ_e	Elastic pole angle.
$\dot{\theta}_e$	Elastic pole angular velocity.
$\ddot{\theta}_e$	Elastic pole angular acceleration.
$d = d_e$	Deflection of the pole.

$\dot{d} = \dot{d}_e$	Velocity of the deflection of the pole.
a_{ch}	The highest value of the acceleration for the entire time of balancing excluding the first one.
$Time_{ach1}$	The value of time for the first a_{ch}.
$Time_{ach2}$	The time it takes to have another acceleration approximately equal to a_{ch}.
tot_time	The sum of the time recorded from $Time_{ach1}$ to $Time_{achN}$.
N	The final time of balancing the pole.
f	Average frequency.
MAYMAY	A computer program that will simulate the dynamics of the flexible pole cart balancing system.
RUNGE()	A numerical integration process that uses fourth order Runge-Kutta.
time_co	Time elapsed.
t_angle	Total angle of the flexible pole.
angle_co	Value of rigid pole angle.
c_ac	Cart acceleration
PE	Processing element of a neural network ; a neurone.
Xi	Number of inputs for each processing elements.
Wi	Connection weight.
Yj	Output signal
γ	Momentum parameter.
λ	Learning rate parameter for hidden layer.
μ	Learning rate parameter for the output layer.
e_i	ith component of output error at the output layer.
t_i	ith component of output error at the hidden layer.
el_ang_n	Normalised value of the flexible pole's angle.
el_ang_r	Raw value of the flexible pole's angle.
max_el_ang	Largest absolute value of the flexible pole's angle.

cart_vel_n	Normalised value of the cart velocity
cart_vel_r	Raw value of the cart velocity.
max_cart_vel	Largest absolute value of the cart's velocity.
cart_dis_n	Normalised value of the cart displacement.
cart_dis_r	Raw value of the cart's displacement.
max_el_ang	Largest absolute value of the cart displacement.
force_n	Normalised value of the force exerted to the cart.
max_force	Maximum force exerted to the cart.
TMAX	Total computer simulation time.
AD	Analog to digital converter.
DA	Digital to analog converter.
FNN	Feedforward neural network.
PLANT	The flexible pole cart balancing system.
MF	Membership function.
FAM	Fuzzy associative matrix.
NL	Negatively large.
NS	Negatively small.
NM	Negatively medium.
ZE	Zero.
PL	Positively large.
PS	Positively small
PM	Positively medium.
GA	Genetic algorithm.
max_gen	The maximum number of cycles that the GA's operate.
Pop_size	The size of the population in GA's operation.
max_tolerance	A value that flags GA's operation to stop.
LQR	MATLAB control system design technique.

Contents

List of Figures

List of Tables

CHAPTER 1

1.1. Introduction

There has been a considerable development of nonlinear control theory in the last 10 years with the exploration of such ideas as feedback linearization, input output linearization, fuzzy control theory [1], and neural networks [2]. However, actual implementations of these techniques in manufacturing industries have been rare. This may be because of the considerable computational requirement needed by these new algorithms, and a lack of communication between theoreticians and industrial practitioners of control engineering.

However, due to technological breakthroughs in digital signal processors and other technologies, the capabilities of computational hardware have steadily increased. Thus, the implementation of complex nonlinear learning control algorithms with relatively inexpensive components may now be possible.

The inverted pendulum (pole-cart balancing) problem has received a great deal of attention as a model problem for the establishment of learning control systems [3, 4, 5, 6, 9, 10, 11]. That authors are successful in this field can be seen in results of their published work. However, using a rigid pole as the pendulum, analysis shows that this system has only two degrees of freedom and little non-linearity. As a result of these limitations the learning controllers developed using such a demonstrator problem have limited power and are unlikely to have broad applications to manufacturing industries. Because of the limitations mentioned, this author modifies the pole-cart balancing problem to give a more exacting testbed for learning controllers by replacing the rigid pole by an elastic pole. The dynamics of this new system are more complex and highly nonlinear when compared to the traditional rigid pole-cart balancing system as a result of the additional degree of freedom within the system, e.g., the transverse displacement of the elastic pole.

Modelling and control of flexible robot systems has attracted much interest in recent years [12, 29, 30, 31, 37, 38, 39]. This has arisen, in particular, in the area of space and industrial robots that require lightweight and flexible links [43]. Flexible robot manipulators have many advantages compared to robot manipulators constructed from rigid links. This is discussed in section 3.1. If the advantages associated with the lightweight machine elements are not to be sacrificed then advanced control systems for such flexible robot manipulators have to be developed [29]. The flexible pole testbed explored in this thesis allows the examination of some of the control issues within flexible linked robots.

1.2. The Area of Investigation

The author began his work by investigating the limitations of the inverted pendulum as a benchmark for learning controllers. Here, the objective was to investigate the inverted pendulum problem and analyse its usefulness as a benchmark for developing learning control systems and their application in manufacturing industries. It shows particularly that the pole-cart problem may not be sufficiently testing, hence, the author extended the problem using a flexible pole as a replacement for the rigid pole.

To verify the feasibility of solving the flexible pole-cart balancing problem, the author has generated a computer simulation of this system. Equations of the dynamics of this system have been derived and a rule based bang-bang control system developed. A graphical representation of the system behaviour has been made to show the cart balancing the pole along a track in real time. Having shown by computer simulation that it is possible to control the flexible pole-cart balancing problem under its first mode of vibration, the next stage of the research addressed this in the real application.

This thesis therefore, focuses on developing and testing on line and off line learning controllers that balance a flexible pole hinged by its root on top of a cart moving

along a limited track. The capabilities of neural network algorithms, fuzzy logic systems and genetic algorithms have been investigated and tested in control of the system.

1.3. Structure of the Thesis

There are 7 further chapters within this document. Chapter 2 first provides an introduction to the general problems associated with control engineering. Reviews of the concepts used in developing classical and intelligent controllers are discussed and particular attention is paid to the inverted pendulum problem as a testbed for learning controllers.

Chapter 3 presents a new model problem (a highly nonlinear system) to be used as a testbed application for developing intelligent controllers. Simplified mathematical equations of the dynamics of the flexible pole-cart balancing problem are derived and a computer simulation conducted in order to verify the validity of these equations. A rule based controller is implemented to control the system off line and a graphical representation of the motion of the system is presented.

Chapters 4 and 5 concentrate on controlling the flexible pole-cart balancing system using neural network techniques. Chapter 4 discusses particularly the development and testing of off-line controller using backpropagation (feedforward neural network) and a Kohonen's Self Organising Map to control the system. Chapter 5 focuses on developing and testing an on line controller for the real system. A physical flexible pole-cart balancing system is developed and controlled in real time.

Chapter 6 concentrates on developing and testing an on-line fuzzy logic system controller for the flexible pole-cart balancing system. Chapter 7 deals with development of a genetic algorithm combined with a fuzzy logic systems controller. This is a particularly novel approach to solving the problem since the fuzzy logic system does not need human expertise to calculate the values of its parameters. Chapter 8 reviews the contribution of the thesis, and identifies areas in which it will be fruitful to conduct future work. The

diagram of figure 1.1 shows the general structure of the thesis and the relationships between the individual elements of the work

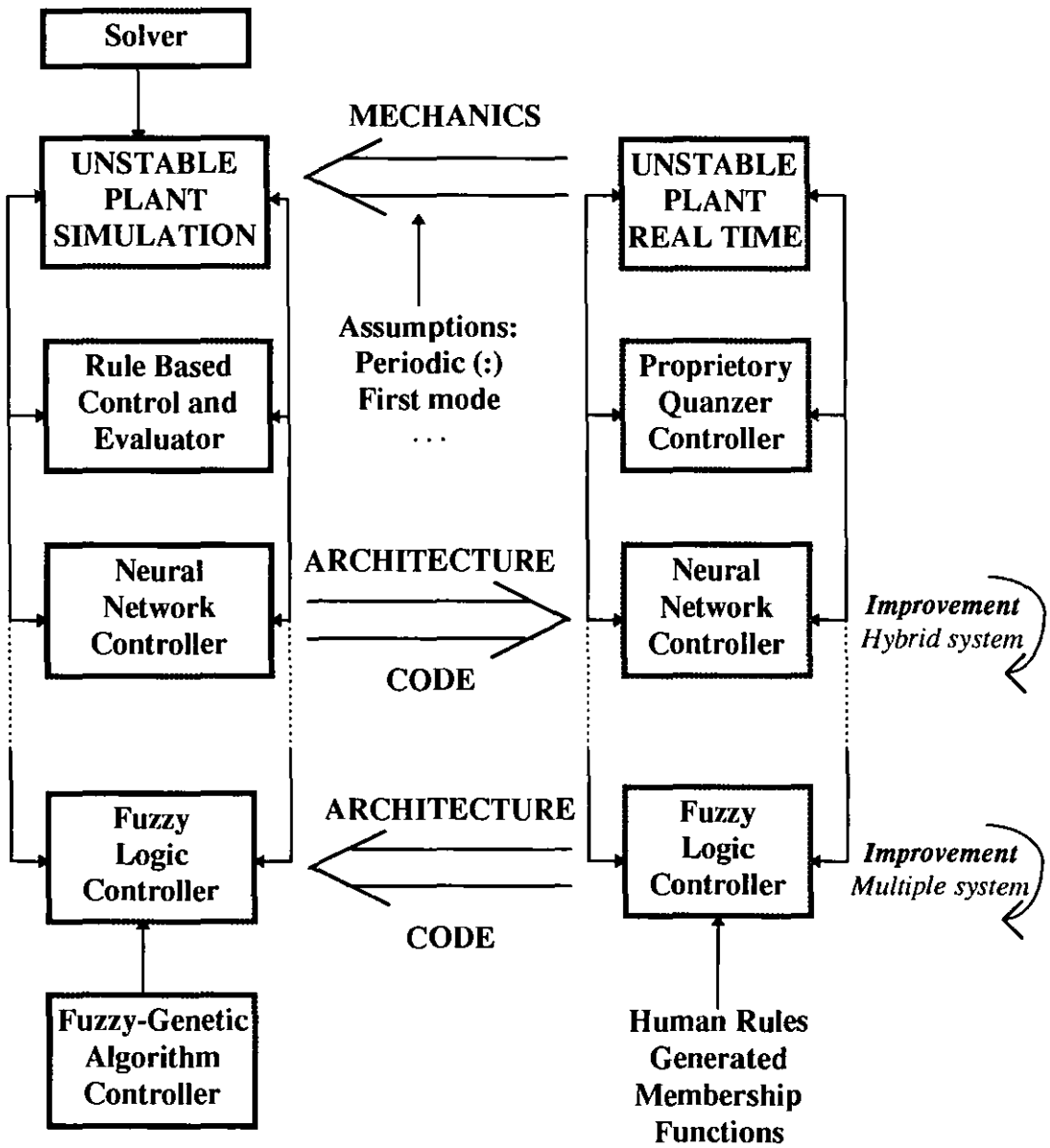


Figure 1.1
General structure of the thesis

CHAPTER 2

Literature Review

2.1. Introduction

The objective of this chapter is to review and investigate the inverted pendulum problem and analyse its usefulness as a benchmark for developing learning control systems and their application. Results of previous research concerning the inverted pendulum problem are presented. A review and analysis of the dynamics of this problem is undertaken. This chapter also summarises key elements of the theory of conventional control and trainable control.

A common problem in controlling a system is to provide the correct input vector to drive a nonlinear plant from an initial state to a subsequent desired state. The typical approach to solve this problem involves linearizing the plant around a number of operating points, and then building a controller [3]. For nonlinear plants this approach is usually computationally intensive and requires considerable design effort.

When constructing a controller there are three kinds of information available. The first is numerical information from measuring instruments, the second is the linguistic information from human experts, and the third is the behavioural characteristics of the plant and its environment. Most of the supervised learning methods associated with neural networks, such as the perceptron, the back-propagation algorithm, and reduced coulomb energy network, utilise only numerical data [8]. The unsupervised learning methods of neural networks have demonstrated the capability to handle behavioural characteristics of the plant and environment. Fuzzy control is one of the most useful approaches for utilising expert knowledge. Many hybrid techniques of fuzzy control

systems and neural networks have been also proposed for utilising numerical data [21]-[26]. In these techniques, the learning ability of neural networks has been incorporated into fuzzy control systems to generate and adjust fuzzy if-then rules using numerical data. Other proposed techniques use learning methods of neural networks to utilise not only numerical data but also expert knowledge represented by fuzzy if-then rules [8]. The idea of this technique is to utilise fuzzy if-then rules obtained from human experts in support of neural network learning.

This chapter begins by reviewing the concepts of developing traditional controllers, and it continues by the discussion of the concepts of constructing adaptive and learning control systems. Also, reviews of the past and recent work pertaining to the pole-cart balancing problem (inverted pendulum) are presented.

2.2. Technological Control Systems

The Merriam - Webster dictionary defines control as a device for regulating a mechanism. It is a mean of directing and influencing an object in order for that object to behave in a desired way. The term "control system" can be substituted by "cybernetical system" where cybernetics is defined by Wiener as the science of control and communication in the animal and the machine [13].

There are two major objectives in designing control systems [54]. The first one is to make the state or output of the system to be very close or equal, if possible to the set points or reference input. In short it is necessary to have a very small steady state error e(t) with time t. The second objective is to maintain the transient performance of the system within reasonable limits.

Conventional or basic control systems are classified into two categories. The open-loop control system and the closed-loop control system. In an open-loop control, the amount of the corrective effort is determined by the desired value of the controlled variable. An example for this is a gasoline engine whose function is to drive a load. Since a small pressure on the engine throttle will cause a large change in the power output, the speed of the shaft for a constant load is a function of the position of the throttle.

In a closed-loop control system the amount of corrective effort is determined by the actual value of both the controlled variable and the desired value. The closed-loop control system uses feedback to regulate the mechanism. Hence, sometimes, it is called the feedback control system (see figure 2.1). The actual output of the system in this case is returned to the controlling system. The error is determined from the difference of the actual output and the prescribed reference input. This error is used by the controller to adjust the mechanism in order for this to give the correct, desired behaviour.

A "conventional" control system can also be divided into sub-groups depending on the relationship of the output of the controller to the error. Among this group are: proportional control (P), proportional-plus-derivative control (PD), proportional-plus-integral control (PI), and proportional-plus-integral-plus-derivative control (PID) [13].

The proportional (P) control system is a feedback control system in which the output of the controller is directly proportional to the error (see figure 2.2). The proportional-plus-derivative (PD) control is a feedback control system in which the output of the controller is a linear combination of the error and its first time derivative (see figure 2.3). Derivative control causes the changes in the output of the controller in anticipation of an error in the immediate future. The proportional-plus-integral (PI) control is a feedback control system in which the output of the controller is a linear combination of the error and its first time integral (see figure 2.4). The use of integral control in addition to proportional control eliminates steady state errors. The proportional-plus-integral-plus-derivative (PID) control is a feedback control system in

which the output of the controller is a linear combination of the error, its first time integral, and its first time derivative (see figure 2.5). This type of controller is particularly useful for high steady-state accuracy and high speed settling.

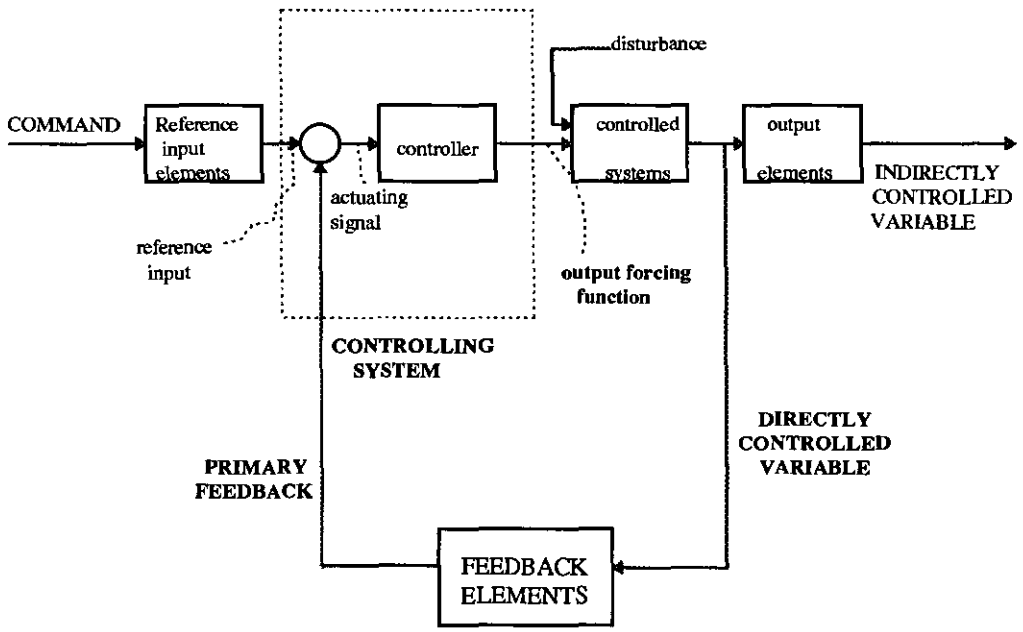


Figure 2.1
Generalised feedback control system

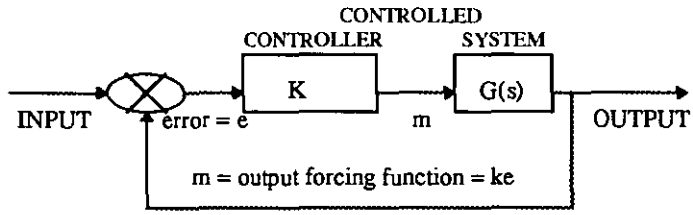


Figure 2.2
Proportional Control

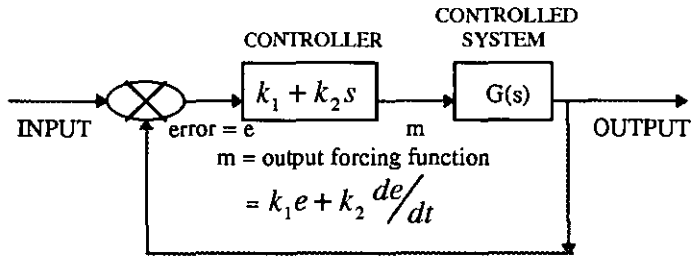


Figure 2.3
Proportional Plus Derivative Control

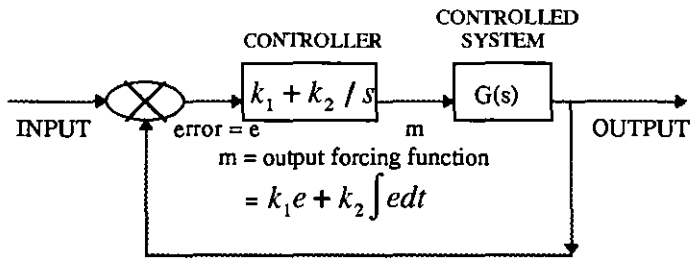


Figure 2.4
Proportional Plus Integral Control

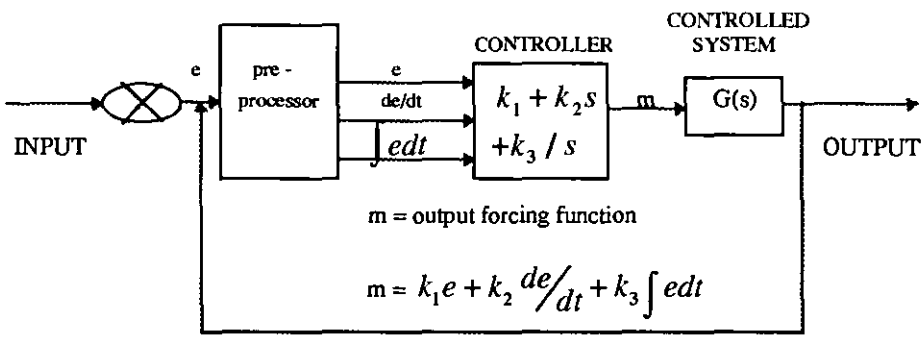


Figure 2.5
Proportional Plus Integral Plus Derivative Control

2.3. State Space Analysis

Section 2.2 introduced conventional control systems (open loop and closed loop control system). The analysis and designed of such systems is based on converting a system's differential equation to a transfer function, thus generating a mathematical model of the system that algebraically relates a representation of the output to a representation of the input. Conventional methods are conceptually simple and require only a reasonable number of computations, but they are applicable only to linear time-invariant systems having a single input and single output [70].

With the arrival of space exploration, requirements for control systems increased in scope. A modern control system may have many inputs and many outputs, and these may be interrelated in a complicated manner. Modelling systems by using linear, time invariant differential equations and subsequent transfer functions became inadequate. The state-space approach (also referred to as the modern, or time-domain, approach) is a unified method for modelling, analysing, and designing a wide range of systems [71]. This type of approach can handle, conveniently, systems with nonzero initial conditions. Multiple-input multiple-output systems (e.g., a vehicle with input direction and input velocity yielding an output direction and an output velocity) can be compactly represented in state space with a model similar in form and complexity to that used for single input, single-output systems. The state-space approach can be used to represent systems with a digital computer in the loop or to model systems for digital simulations. It can be used also for the same class of systems modelled by the classical approach which gives the control systems designer another perspective from which to create a design. The disadvantage of the state-space approach is that it is not intuitive as the conventional approach. The designer has to engage in several calculations before the physical interpretation of the model is apparent, whereas in conventional control, some straightforward calculations or a graphic presentation of data rapidly yields the physical interpretation.

The concept of the state-space approach is based on the description of system equations in terms of n first-order differential equations, which may be combined into a first-order vector-matrix differential equations [70]. The use of the vector-matrix notation greatly simplifies the mathematical representation of the system equations.

2.3.1. General Representation of State-Space Approach

In order to formalise the representation of state-space approach it is necessary to define the following terms [70, 71].

- State - The state of a dynamic system is the smallest set of variables (called state variables) such that the knowledge of these variables at $t = t_0$, together with the knowledge of the input for $t >= t_0$, completely determines the behaviour of the system for any time $t >= t_0$.
- State variables - The state variables of a dynamic system are the variables making up the smallest set of variables that determine the state of the dynamic system. If at least n variables $x_1, x_2, \ldots, x_n$ are needed to completely describe the behaviour of a dynamic system (so that once the input is given for $t >= t_0$ and the initial state at $t = t_0$, is specified, the future of the system is completely determined), then such n variables are a set of state variables.
- State vector - A state vector is a vector that determines uniquely the system state **x**(t) for any time $t >= t_0$, once the state at $t = t_0$ is specified. It is a vector whose elements are the state variables.
- State Space - The n-dimensional space whose coordinate axes are the state variables. Any state can be represented by a point in the state space. This is a new term and is illustrated in figure 2.6, where the state variables are assumed to be, v_c, and v_r. These variables form the axes of the state space. A trajectory can be thought of as

being mapped out by the state vector, $\mathbf{x}(t)$, for a range of t. Also shown is the state vector at the particular time $t = 4$.

- State Equations - A set of n simultaneous, first order differential equations with n variables, where the n variables to be solved are the state variables.
- Output Equation - The algebraic equation that expresses the output variables of a system as linear combinations of the state variables and the inputs.

Now that the definitions have been formally stated, a state-space representation of a system is determined using the following equations [70].

1. For time-varying (linear or nonlinear) discrete-time systems the state equation may be written as:

$$\mathbf{x}(k+1) = \mathbf{f}[\mathbf{x}(k), \mathbf{u}(k), k] \tag{2.1}$$

and the output equation as:

$$\mathbf{y}(k) = \mathbf{g}[\mathbf{x}(k), \mathbf{u}(k), k] \tag{2.2}$$

For linear time-varying discrete-time systems, the state equation and output equation may be simplified to:

$$\mathbf{x}(k+1) = \mathbf{G}(k)\mathbf{x}(k) + \mathbf{H}(k)\mathbf{u}(k) \tag{2.3}$$

$$\mathbf{y}(k) = \mathbf{C}(k)\mathbf{x}(k) + \mathbf{D}(k)\mathbf{u}(k) \tag{2.4}$$

where

$\mathbf{x}(k)$ = n-vector	(state vector)
$\mathbf{y}(k)$ = m-vector	(output vector)
$\mathbf{u}(k)$ = r-vector	(input vector)
$\mathbf{G}(k)$ = n x n matrix	(state matrix)
$\mathbf{H}(k)$ = n x r matrix	(input matrix)
$\mathbf{C}(k)$ = m x n matrix	(output matrix)
$\mathbf{D}(k)$ = m x r matrix	(direct transmission matrix)

Note that the appearance of the variable k in the arguments of matrices $\mathbf{G}(k)$, $\mathbf{H}(k)$, $\mathbf{C}(k)$, $\mathbf{D}(k)$ implies that these matrices are time varying.

If the system is time-invariant, then the state equation and output equation may be simplified to:

$$\mathbf{x}(k+1) = \mathbf{G}\mathbf{x}(k) + \mathbf{H}\mathbf{u}(k) \quad (2.5)$$

$$\mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k) \quad (2.6)$$

2. For continuous-time (linear or nonlinear) systems, the state equation may be written as:

$$\dot{x}(t) = \mathbf{f}[\mathbf{x}(t), \mathbf{u}(t), t] \quad (2.7)$$

and the output equation as:

$$\mathbf{y}(k) = \mathbf{g}[\mathbf{x}(t), \mathbf{u}(t), t] \quad (2.8)$$

For linear time-varying continuous-time systems, the state equation and output equation may be written as:

$$\dot{x}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \quad (2.9)$$

$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t) \quad (2.10)$$

If the system is time-invariant, then the state equation and output equation may be simplified to

$$\dot{x}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad (2.11)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \quad (2.12)$$

Figure 2.7 shows the block diagram representation of a discrete-time control system defined by equations 2.5 and 2.6, and figure 2.8 shows the continuous-time control system defined by equations 2.11 and 2.12 [70].

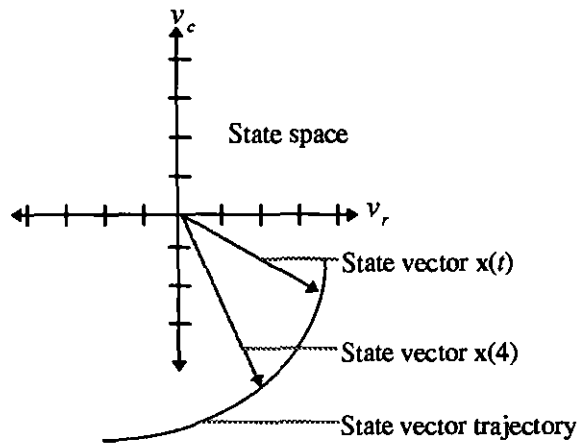


Figure 2.6
Graphic Representation of state space and a state vector

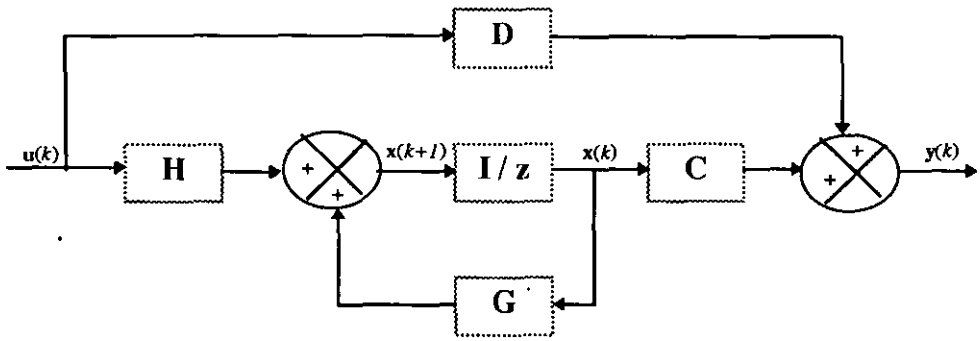


Figure 2.7
Block diagram of the linear time-invariant discrete-time control system represented in state space

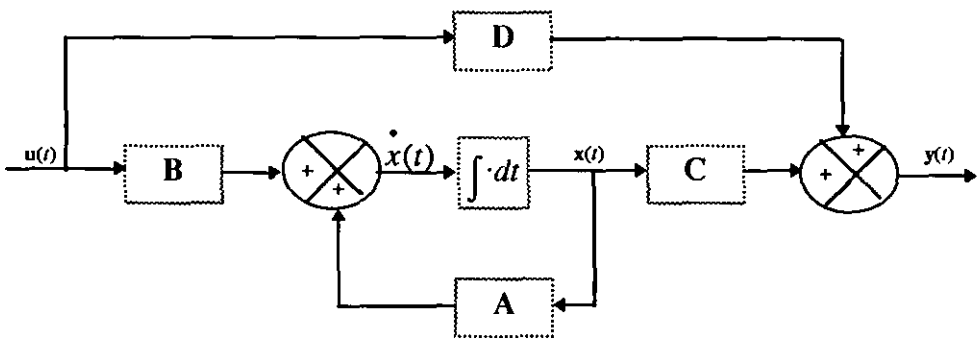


Figure 2.8
Block diagram of the linear time-invariant continuous-time control system represented in state space

2.4. Adaptive Control Systems

Researchers from different fields give different meanings to the word adaptation. For engineers adaptation often is equated to learning. For the psychologist adaptation and learning are totally different. Life scientists take an intermediate position.

In 1965 Sklansky [17] made a formal distinction between adaptation and learning. He defines learning and self-repair as a special form of adaptation. He considers a species, an organism, or a cell to be adaptive if its behaviour in a changing environment is successful in some sense. In a machine, success can be distinguished either by stability or reliability. Thus, to an engineer adaptation is demonstrated by the presence of reliability, or stability, or both.

Ashby [18] defines adaptation as a form of behaviour that maintains its essential variables within physiological limits, like homeostasis. This means that for an unpredictable environment persistence of success should be attained. This definition strengthens the requirement of stability. Glorioso [13] defines adaptation as a pre-requisite to reliability. He said that if a portion of a system is damaged and the effect of the damage is gradually masked until the system performance reaches an acceptable level, then the system is adaptive.

Thus, adaptation can be defined formally as the property or a system that reacts favourably with respect to any performance function in the face of changes into environment or to its own internal structure. There are two major functional elements of an adaptive control system [13], a controller and a plant to be controlled. The design of the controller is usually based on the nominal but inexact mathematical model of the plant and/or its environment.

Figure 2.9 describes the functional block diagram of an adaptive control system. Achieving satisfactory response of the plant state o(t) to an input i(t) is the objective of

the system. Inputs **i**(t) are applied to the controller. Normally this is unknown *a priori*. The controller then generates an output which is used as the input to the plant. Direct or indirect measurement of the plant state is carried out through the measuring devices. These measurements are compared with the input **i**(t) via performance assessment identification in order to establish the present performance vector **p**(t). The adaptation algorithm then maps **p**(t) to a weighting input **a**(t). The output of the measuring devices **m**(t) and the weighting input **a**(t) are used as the inputs of the variable structure controller to modify the relationship between the command and the plant inputs. Hence, the behaviour of the system can be improved by changing dynamically the original nominal design.

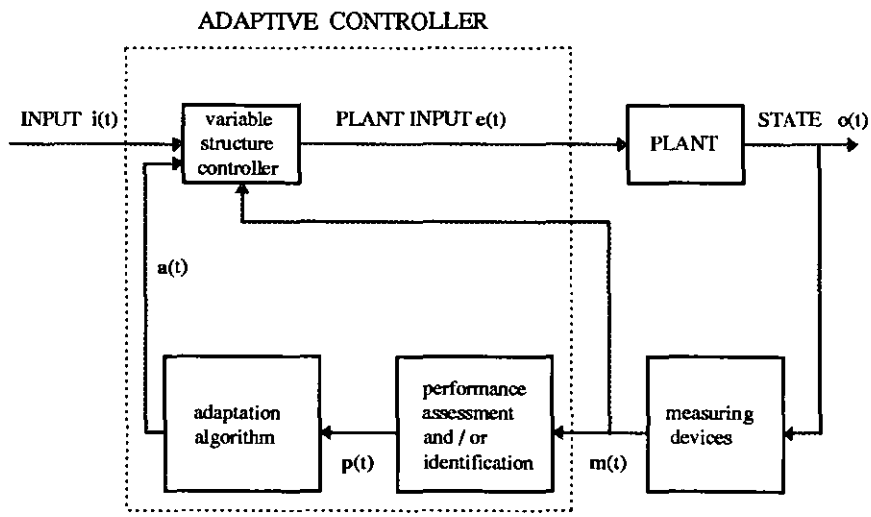


Figure 2.9
Functional block diagram of an adaptive control system

2.5. Learning Control Systems

A system is learning if it reacts favourably with respect to performance function **p** in some time t after a change in its environment [13]. Learning is dependent on time. One can say that a system learns if given a change in the state of environment at $t = 0$, the performance index at $t > 0$ is greater than the performance index at time $t = 0$.

In order that learning controllers know the acceptable reactions of the system at any given time it is necessary to classify the performance of the system resulting from any change in the controller. This classification can be either good or bad and the system must be awarded or punished, respectively. Moreover, since the learning process is dependent on time , the learning control system must have memory. In determining the future behaviour of the system it must be capable of using past and present behaviour results.

Learning controllers generally show performance which gradually improves with time. When there are reductions in the bounds of prescribed information or the improved identification of certain attributes then the system is learning. A learning controller can be defined [11] as a control design that improves the performance of the system being controlled without knowing completely its mathematical model. In this case learning is derived from the behaviour of the plant, either from its operation or experimentation. This learned information is used as the knowledge to influence future decisions to be executed by the controller.

There are various techniques applied when designing learning controllers. It can be that the learning process is obtained by considering all possible answers, that is consolidating short term memory into long term memory, and exhibiting altered behaviour because of what was remembered [32]. It is possible also that the controller itself employs a performance measure to supervise learning. Whenever the same situation occurs the experience of the controller based on learned information is used to improve the quality of control. The information extracted from different control situations

constitutes different experiences. Learning schemes like, supervised learning, unsupervised learning, are explained in section 2.8.2

2.6. The Algorithms used for Developing Intelligent Controllers

An intelligent control system is a system that possess the properties of an adaptive control system or a learning control system (see section 2.4 and 2.5). There are many techniques that can be used in developing intelligent controllers. Among them are neural networks , fuzzy logic systems, and genetic algorithms. Each of these techniques has its own strengths and weaknesses.

A neural network is an information processing system that is nonalgorithmic, nondigital, and intensely parallel [44]. It consists of groups of very simple and highly interconnected processors called neurons or processing elements. A neurone is an analogue of the biological neural cell in the brain. Detailed explanations of neural network techniques are contained in chapter 4.

A fuzzy logic system describes complex systems with linguistic descriptions [54]. Here, the information is described in terms of fuzzy sets. The concept of a fuzzy set is made precise through the definition of an associated membership function. Again chapter 6 contains more discussion of these techniques.

Genetic algorithms are algorithms for optimisation and learning based on the mechanism of genetic evolution [63]. They give solutions to problems using a probabilistic optimisation method based on evolution strategies as nature solves the problem of adapting living organisms to the harsh realities of life in a hostile world. Chapter 7 includes a detailed explanation of this topic.

2.7. The Inverted Pendulum

2.7.1. The Problem

The inverted pendulum system consists of a pole hinged at the top of a wheeled cart that travels along a limited track. The task of the controller applied to this system is to balance the pole when the cart is pushed back and forth by a force of magnitude F. The pole can only swing in a vertical plane parallel to the direction of motion of the cart. Balancing fails when the cart hits the end of the track or the inclination of the pole exceeds preset limits. The overall goal is to find a controller that prevents the system from failing. A more demanding version of the inverted pendulum experiments requires the controller to balance the pole and bring back the cart to the centre of the track [10]. Figure 2.10 describes the system.

The state of the inverted pendulum system is described by four variables :

x = the position of the cart in a track.

v = dx/dt = the velocity of the cart.

θ = the angular position of the pole.

ω = $d\theta / dt$ = the angular velocity of the pole.

Assuming that the system is frictionless the dynamic equations [10] are:

$$\frac{d\omega}{dt} = \frac{g\sin\theta - \partial\cos\theta - \mu_p \omega^2 l \cos\theta \sin\theta}{l(\frac{4}{3} - 3\mu_p \cos^2\theta)} \qquad (1)$$

$$\frac{dv}{dt} = \frac{\frac{4}{3}\partial + \left(\frac{4}{3}\omega^2 l - g\cos\theta\right)\mu_p \sin\theta}{\frac{4}{3} - 3\mu_p \cos^2\theta} \qquad (2)$$

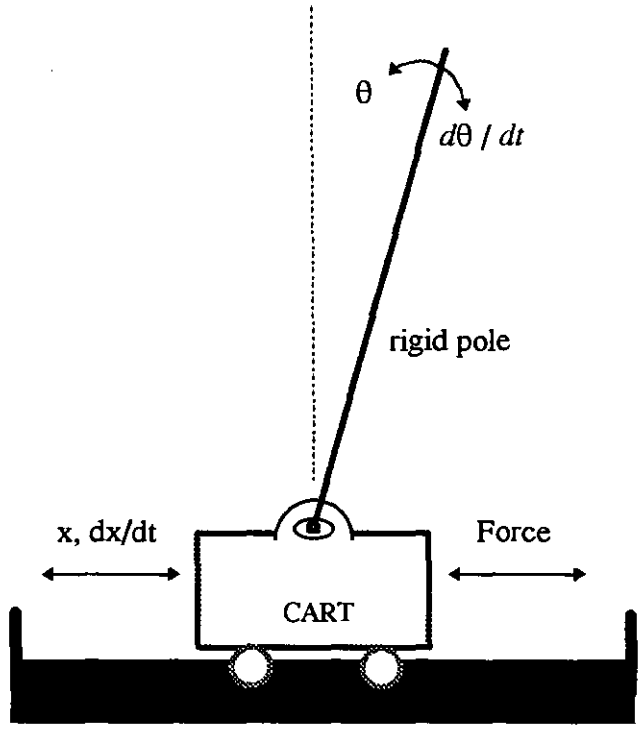


Figure 2.10
The inverted pendulum

where μ_p is the reduced mass of the pole

$$\mu_p = \frac{m_p}{m_p + m_c}$$

and

$$\partial = \frac{F}{m_p + m_c}$$

with the standard parameters as

acceleration due to gravity	$= g$
length of the pole	$= l$
mass of the cart	$= m_c$
mass of the pole	$= m_p$
magnitude of the control force	$= F$

2.7.2. The Reasons for using the Inverted Pendulum as a Control Benchmark.

The inverted pendulum problem is popular in the field of research in learning control systems for the following reasons [10]:

1) The problem is apparently difficult but it is easy to understand and simple to describe.
2) It is a textbook example of an inherently unstable control system. This has been analysed in detail with conventional control theory providing a convenient reference for assessing neural network controllers [20].
3) The system is not too expensive. Thus, it is easier to develop and demonstrate.
4) There are constraints on the response time of any controller because the cart-pole is a real time problem.

2.8. Previous Research in Control of the Inverted Pendulum

2.8.1. General Experiments

Widrow and Smith [6] pioneered the inverted pendulum problem in 1963 as an application for neural network based control. They demonstrated that a network of one computing element, an adaptive linear element (ADALINE), was capable of balancing an inverted pendulum if the ADALINE input consisted of the four state variables, each encoded with an n-bit linearly independent code [9]. The force produced by the ADALINE approximated that called for by the equation

$$\mathbf{F} = \mathbf{k}\,\mathrm{sgn}(W_1 \cdot \theta + W_2 \cdot \dot{\theta} + W_3 \cdot x + W_4 \cdot \dot{x})$$

Where:

- F is the force required to stabilise the system at any time.
- k is a positive constant representing the magnitude of the force to be applied to the system.
- sgn is the sign (or direction) of the applied force.

The coefficients W_1, W_2, W_3, W_4, are derived from the physical characteristics of the pendulum system and optimal bang-bang control theory.

Widrow and Smith trained the network using the Widrow-Hoff least mean square (LMS) algorithm with the output on an optimal controller in a form of the equation above as the teacher. The teaching signal was obtained by linearizing the dynamics of the system and applying a conventional control law. In this work they were able to show that a linear control law is sufficient to solve the problem.

In 1983 Barto, Sutton, and Anderson [5] used the inverted pendulum to simulate reinforcement learning control. They used two neuronlike adaptive elements. The first one is an adaptive critic element (ACE), and the second one the adaptive search element (ASE).

The ACE provides an indication of what the reinforcement should be. It evaluates every state of the inverted pendulum system that is used to steer the learning process of the controller [10]. The ACE receives the externally supplied reinforcement signal which it uses to determine how to compute, on the basis of the current system state vector, an improved reinforcement signal that it sends to the ASE [5]. Expressed in terms of the BOXES system [27], the job of the ACE is to store in each box a prediction or expectation of the environment by choosing an action for that box. A BOXES system can learn to control an inverted pendulum using a state representation of discrete regions. For example, in [27] the regions of the state space were formed by the intersections of six intervals along the θ dimension and three intervals along the $\dot{\theta}$, x, and $\dot{x}$ dimensions, making a total of 162 regions. The ACE uses this prediction to determine a reinforcement signal that it delivers to the ASE whenever the box is entered by the inverted pendulum state. Specific knowledge of the dynamics of the inverted pendulum is not necessary. Weights in both the controller and the ACE are adjusted in proportion to the change in prediction from one time step to the next [10]. Initially all the weights of the ACE are set to zero and consequently the prediction is zero for all states. Nonzero predictions spread out gradually from the final failure states as more trials are conducted. The controller is nondeterministic, its output biases a random process towards one of the two control actions.

If the environment cannot provide the necessary responses the ASE must discover what responses lead to improvements in performance. It employs a trial-and-error, or generate-and-test, search process. In the presence of input signals, it generates actions by a random process. Based on feedback that evaluates the problem-solving consequences of the actions, the ASE "tunes in" input signals to bias the action generation process, conditionally on the input, so that it will more likely to generate the actions leading to improve performance [5]. Actions that lead to improved performance when taken in the

presence of certain input signals become associated with those signals in a developing input-output mapping.

The work of Barto et al shows that at least 60 trial runs are needed before the controller successfully balances the system. However, the authors do not mention whether the controller was able to centre the cart.

Anderson [4] in 1989 presented a further paper on the inverted pendulum problem. His paper describes a neural network that learns to generate successful action sequences by acquiring two networks: the action network and the evaluation network.

The action network learns to select actions as a function of states. It consists of a single unit having two possible outputs, one for each of the two allowable control actions of pushing left or right on the cart with a fixed-magnitude force. The output of the unit is probabilistic. The probability of generating each action depends on the weighted sum of the unit's inputs, i.e., the inner product of the input vector and the unit's weight vector.

The evaluation network is needed to apportion the blame for the failure among the actions in the sequence leading to the failure. It consists of a single unit. The evaluation unit learns the expected value of a discounted sum of future failure signals by means of a prediction method called temporal difference. Temporal difference methods learn associations among signals separated in time, such as the inverted pendulum state vectors and failure signals. Through learning, the output of the evaluation network comes to predict the failure signal, with the strength of the prediction indicating how soon failure can be expected to occur. The predictions are adjusted after each step by an amount proportional to the network's input and the difference between the new prediction, based on the current state of the inverted pendulum, and the previous prediction, based on the previous state, i.e., the temporal difference or change in prediction of failure. The temporal difference method allows learning to occur

continuously using the learned evaluation function and differences in its output as reinforcement, rather than waiting for further failure.

The experiments described in Anderson's work were motivated by the work of Michie and Chambers [27], the BOXES system. To compare with the performance of the BOXES learning system, Anderson uses the same state representation (162 regions). The resulting networks are shown in figure 2.11. Each unit receives the 162 binary input components, and the evaluation unit's output directs the learning process for both units. The result of this experiment is shown in figure 2.13.

Anderson extended his experiment by using two layer networks with unquantized state variables. His motive is primarily to improve learning speed. A very fine quantization with many regions permits accurate approximation of complex functions, but learning the correct output for each of the many regions requires much experience. Learning can be faster for a course quantization because learning from one state in a region is transferred to all states in the region, but only functions whose output remains relatively constant over regions can be represented. The architecture of this network is shown in figure 2.12 and the result of this experiment is shown in figure 2.14.

The most recent research into the control of the inverted pendulum problem was carried out in 1991 by Bing Zhang [11], and in 1993 by Geva and Sitte [10]. The results of these experiments are discussed separately.

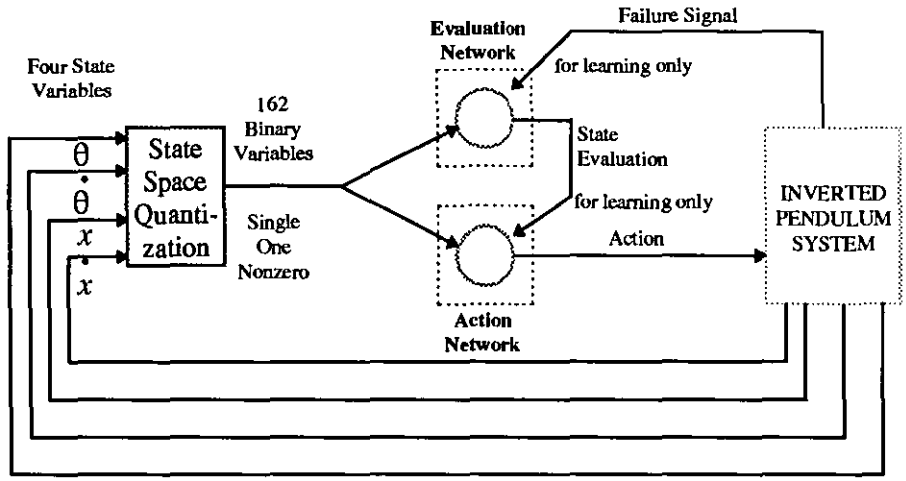


Figure 2.11
Single-layer networks with state-space quantization

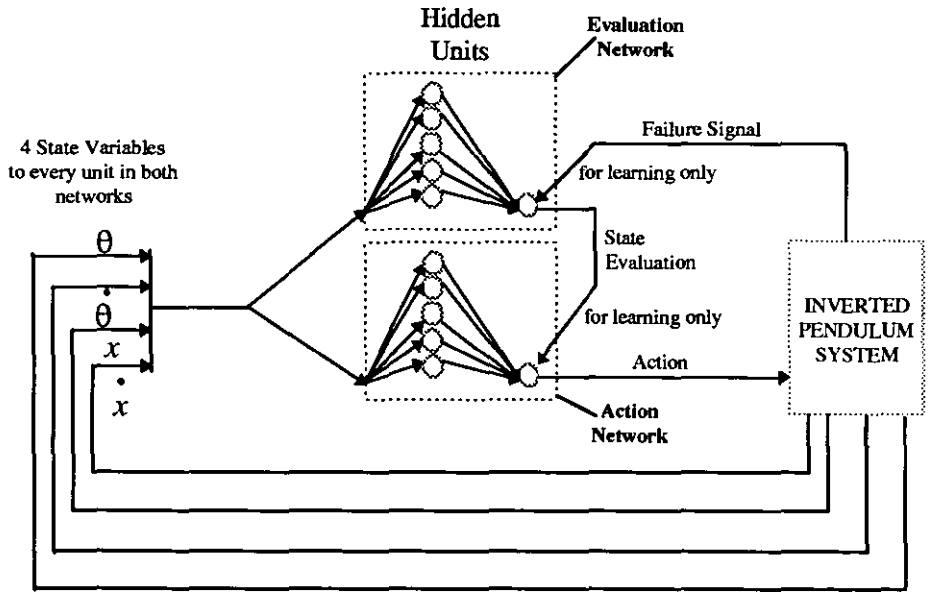


Figure 2.12
Two-layer networks receiving unquantized state variables

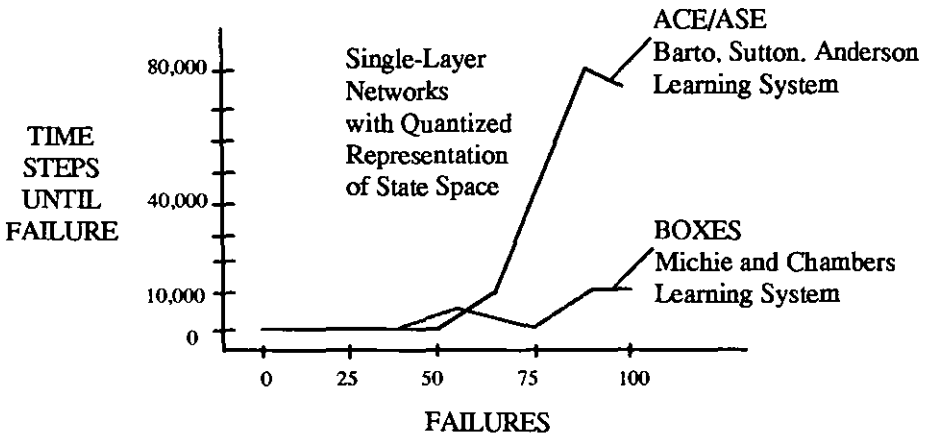


Figure 2.13
Learning curves for single-layer networks and BOXES using quantized representation of state space

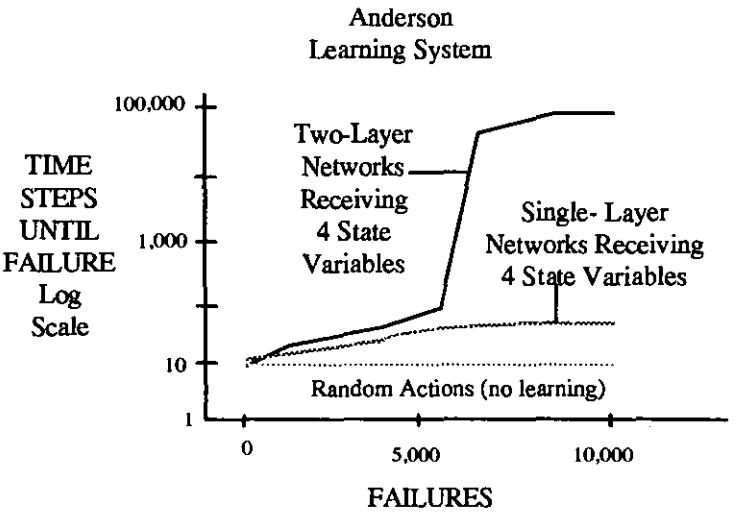


Figure 2.14
Learning curves for two-layer and single-layer networks receiving unquantized state variables

2.8.2. Bing Zhang's Experiment

The objective of Zhang's experiment is to demonstrate by physical experimentation and software simulation, how the ideas of neurocomputing can be used to build adaptive learning controllers to control complex dynamic systems [11]. He presented supervised learning, reinforcement learning, and unsupervised learning solutions to the inverted pendulum problem.

In the reinforcement learning technique, learning is achieved through backpropagation of reinforcement signals provided by a subgoal. Here the learning process is achieved through adopting a reinforcement scheme based on an evaluation of a subgoal which is related to the desired overall system performance. Reinforcement is a feedback process that provides information about the correctness of the actions taken by the system but does not provide information to indicate what the correct action is. The feedback is provided by the environment. Here the system only receives feedback indicating the value of the system's action. This method can be used with systems that vary with the external environment provided the system variables can be measured [11]. The advantage of this method is that less *a priori* information needs to be known about the system. It is useful in those cases where supervisory information is not available and leads to the development of more autonomous system.

Zhang believed that the overall goal of the inverted pendulum balancing task is hard to formulate mathematically, and also that such a goal does not provide any hint as to whether the control action decided by the action network is a good control choice or a bad one. In other words the goal cannot supply the action network with the necessary reinforcement signal required to adjust the connection weights of the action network.

In view of the problem mentioned above Zhang used an approach to establish a subgoal that can be mathematically formulated as a cost function (in this case a quadratic

cost function) so it can be used to direct the updating of action network weights. This subgoal can be expressed as:

Subgoal = $SG(t) = X(t)GX(t)$

where :

- $SG(t)$ = a chosen subgoal that represents a weighted vector distance of the state of the system at time t from the origin of the state space.
- $X(t)$ = a system state variable at time t.
- G = a positive definite diagonal matrix with elements $(g_{11},...,g_{nn})$ which may be either pre-assigned or determined through a learning process.

The best value for G depends on the plant as well as the system overall goal through trial and error. In practice, one of the elements of G may be chosen as 1, others can be pre-assigned initially and then learned precisely through a secondary learning loop to give a result which optimises the system overall goal. This secondary learning loop consists of standard AI techniques (breadth-first, depth-first, best-first, etc.).

Zhang chooses a subgoal equal to:

$$SG(t) = g_{11}x^2(t) + g_{22}\dot{x}^2(t) + g_{33}\theta^2(t) + g_{44}\dot{\theta}^2(t)$$

Here, g_{44} was intentionally chosen as 1 because the weighting factors are all relative to each other.

2.8.2.1. Bing Zhang's Computer Simulation Experiment

1. The program was run 50 times.
2. Each run consists of n number of trials
3. Each trial starts with the pole cart system set to a random state and ends with a system failure.

4. The length of track is 2.4 - 4.8 meters

Computer simulation results

1. In all 50 runs the system learned to balance the pole for 100000 time steps, equivalent to 33 minutes of balancing.
2. In most runs, the system learned to balance in less than 20 trials, sometimes as few as 4 trials, the average is 17 trials.
3. Zhang did not mention whether the cart stayed at the centre.

2.8.2.2. Bing Zhang's Physical Model Experiment

The physical system hardware for Zhang's inverted pendulum experiment is shown in figure 2.15. This experiment uses the following parameters :

Mass of the pole = 0.1 kg

Mass of the cart = 1.0 kg

Length of the pole = 1 meter

Length of the track = 2 meters

The cart is mounted on a parallel track and controlled by a direct current motor that provides propulsion for bang-bang control by applying a constant left or right force only. The voltage of the power amplifier is applied to the motor through two relays. A negative or positive voltage of the same amplitude is applied to the motor, depending on the relays, moving the cart in one direction or another via a steel wire. The relays are controlled by two signals coming from the serial RS232 port of a PC. These two signals are controlled also by the I/O command in the control program.

Two reed switches are placed at each end of the track to detect if the cart has reached the end (the failure signal used in learning algorithm). The switches are activated by a magnet mounted on the cart. The status of the switches is determined through the serial I/O port of the IBM PC/AT computer. This PC is also responsible for the execution of whichever control algorithm currently is under test.

An optical shaft encoder mounted on the cart is used for computing the pole angle and velocity. The angular velocity is calculated as the average rate of change in angle relative to the previous time period. The position of the cart along the track is also sensed using another shaft encoder. The resolution of the two encoders used is 1000 counts per revolution. This gave a resolution of 0.009 centimetre per count for the cart position on the track and 0.36 degree per count for the pole angle.

The computer establishes the current status of the pole and the cart by decoding the count readings from the two encoders attached to the physical system. An 8-bit up-down counter was used to count the pulses from the encoder attached to the pole, and a 16-bit up-down counter is used to count the pulses from the encoder and therefore sensed the position of the cart along the track. The decoding circuit together with the necessary I/O interface with the computer was built on a prototype board which could be easily plugged into one of the I/O slots of the PC.

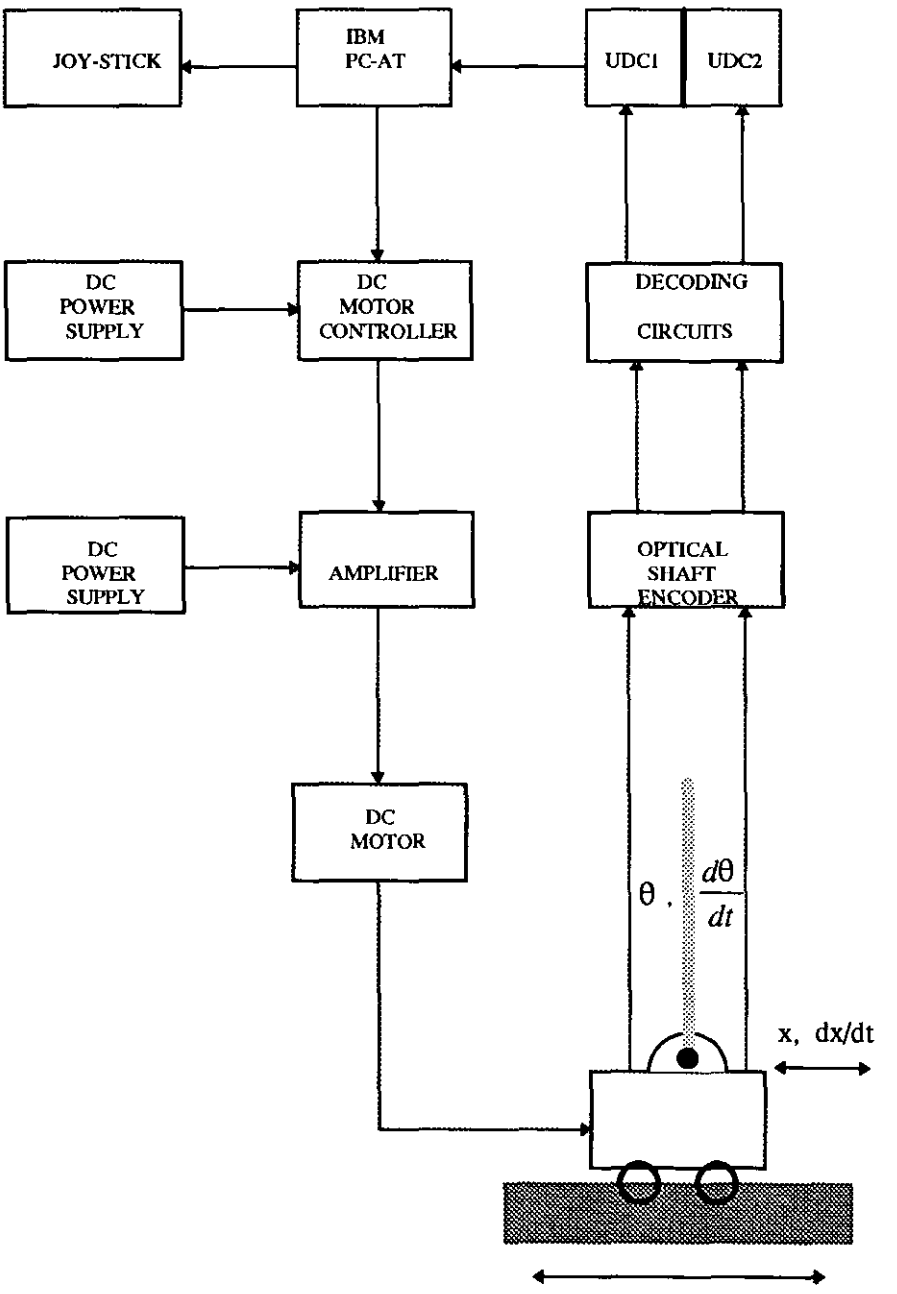


Figure 2.15
High level architecture of pole-cart physical system

2.8.2.2.1. Bing Zhang's Physical Model Experiment Results

Figure 2.16 shows the results of this experiment. The system learned to balance the pole in less than 20 trials, for a period of 5 minutes which was the cut-off time for this experiment. The final pattern of movements of the system in some of the trials which lasted for more than 3 minutes were unexpected. This is shown in figure 2.17. From this figure, it can be seen that the inverted pendulum system is not oscillating around the centre of the track. Also, further learning was found not to improved the result. The reason for this according to Zhang was that the network had reached a local minimum and not the global optimum. Another reason was due to the effects of the unknown friction between the cart and the track.

Zhang encountered a number of problems in this experiment. He mentioned that for most of the time the change in pole angle between subsequent two samplings was too small for the computer to detect. This was because the control decision was calculated and executed very quickly. Direct sensing of pole and cart velocities were not provided. Another problem was the count readings decoded by the computer. It tended to increment or decrement erroneously while the position of the pole remained unchanged. According to Zhang, the reason for this was the shaky vertical movements between the cart and the track during the horizontal movements of the cart along the track. This problem created considerable difficulties in the experiment. The pole often biased in one direction or another, causing the cart to move in the same direction, that eventually resulted in failure (hitting the track end). Zhang solved this problem by remounting the encoder in such a way that when the pole was almost vertical (angle = 0), the reference signal produces an effective low signal which in turn cleared the up-down counters of the decoding logic. In this way, the pole angle (encoder count readings)

sensed by the computer was calibrated to zero whenever the pole passed the straight vertical position.

In an another attempt to solve the problems Zhang retained reinforcement learning, but this time he divided the learning task into two sub-tasks. The first subtask is the pole angle learning control strategy, and the other second subtask is the cart position learning strategy. According to Zhang, this attempt is supported by the fact that, when the pendulum is almost vertical ($\theta = 0$), the fourth order system could be approximated as two decoupled second order systems.

The results of this experiment are:

I) The decoupled system learned to balance for 100000 time steps in an average of 12 trials.

II) When the condition of applying uneven forces was tested, the system failed in all 20 test runs. Figure 2.18 shows the learning curve of Zhang's algorithm and others.

In supervised learning the system learns through a human teacher. This is good in highly complex systems where it is very difficult to construct a composite subgoal function to promote reinforcement learning due to various unspecified parameters. The human can assess the situations and make decisions based on qualitative data. However, using this technique, Zhang had difficulties in doing his experiment physically, since in real time the human teacher itself can not balance the system. The data that was fed to the computer was incorrect because the human teacher failed to do his job in balancing the pole.

Unsupervised learning is a technique used in neural network to control a system without requiring a teacher. The neural network controllers can autonomously learn to control the unknown complex system and adapt the changing environment. In this type of control, Zhang mentioned that he developed two separate learning algorithms to provide adaptive learning of state-space partitioning. Unfortunately this is not reported.

At the conclusion to his experiment Zhang recommended the following for future work.

I) Establish satisfactory subgoals which can successfully guide the learning process.

II) Ideally such subgoals should be learned, and not specified by human being in advance.

III) By incorporating powerful subgoal learning mechanics, the reinforcement scheme based on immediate feedback will provide faster and practically more feasible solutions to the real-time control problems.

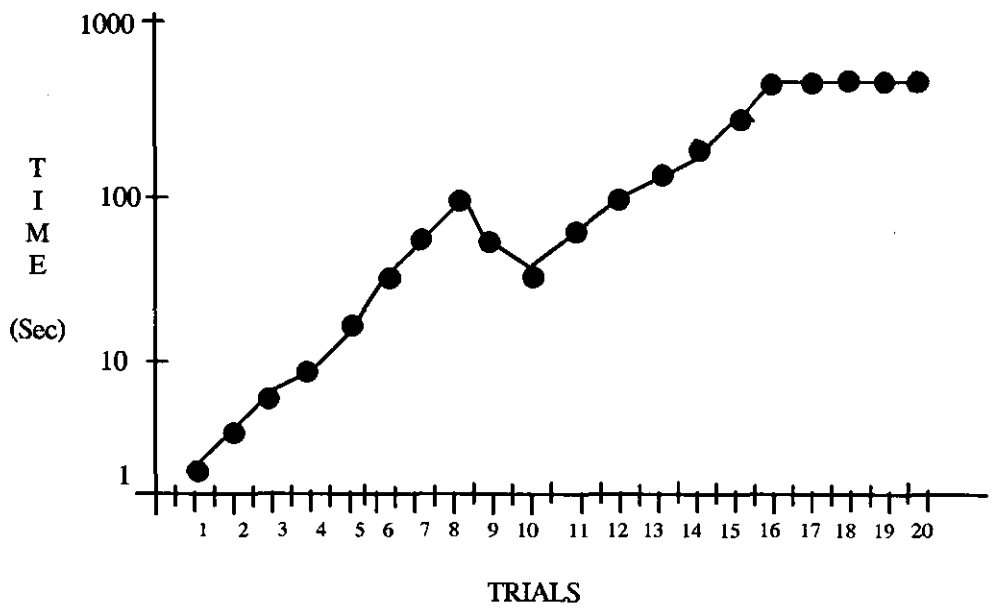


Figure 2.16
Learning curve obtained on physical experiments

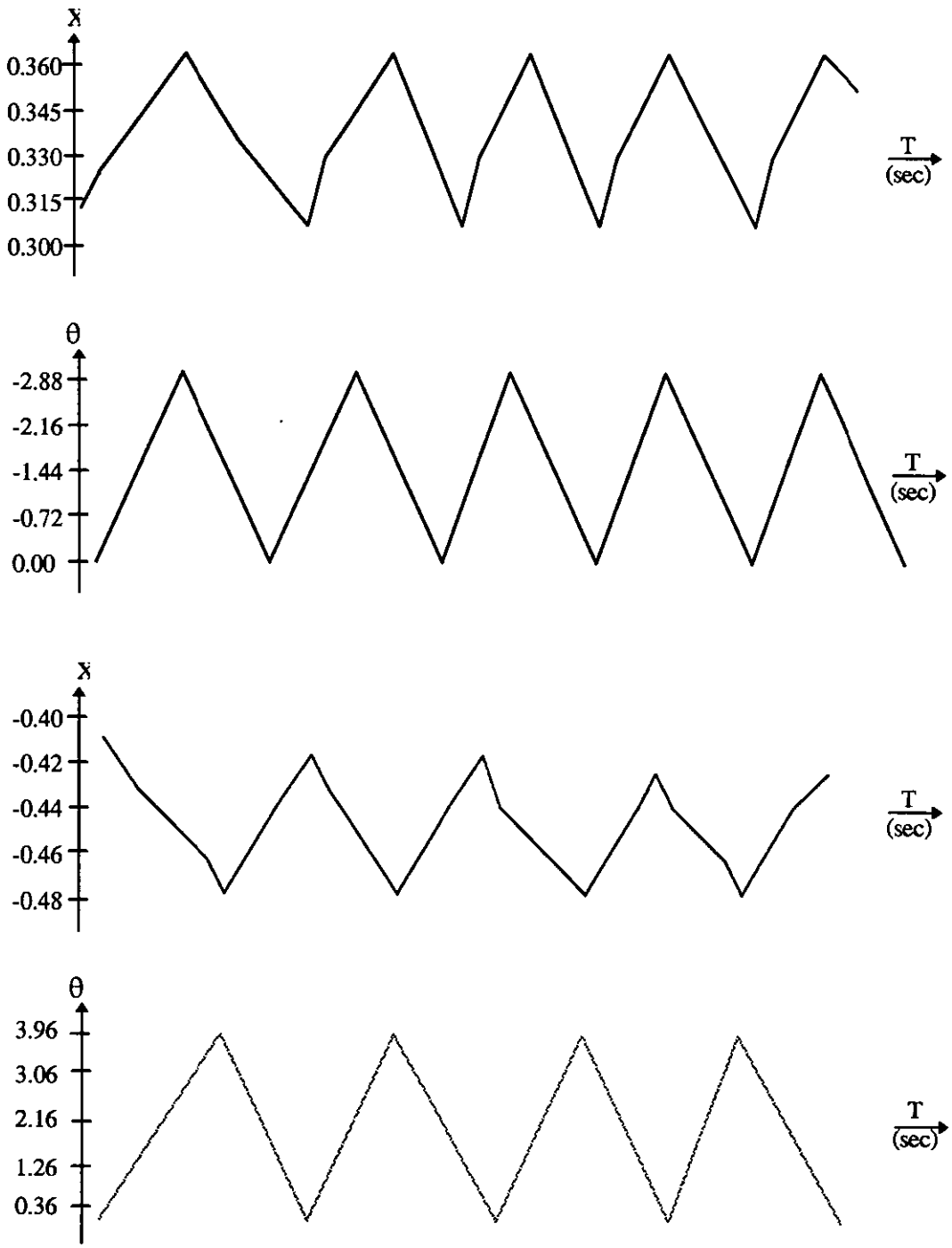


Figure 2.17
Patterns of physical pole-cart system movements

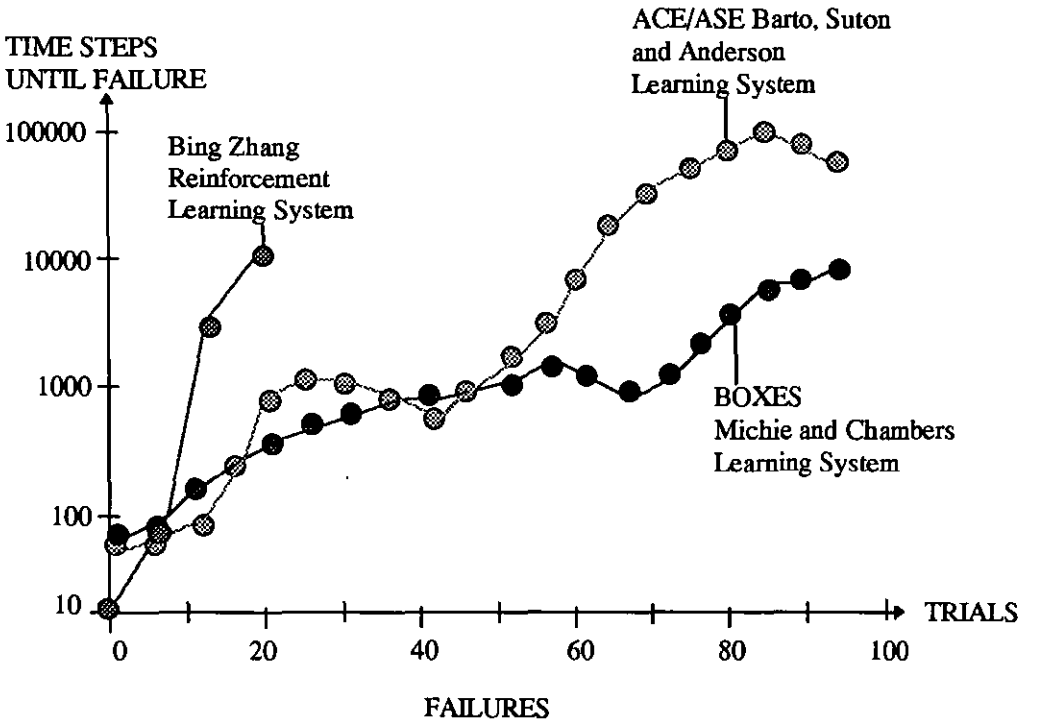


Figure 2.18
A comparison of inverted pendulum learning system algorithm

2.8.3. Geva & Sitte's Experiment

The objectives of Geva & Sitte's experiment is to present a thorough analysis of the inverted pendulum problem, to clear up implied or explicit misconceptions contained in earlier work, and to propose a set of conditions to make it a useful and well defined benchmark for neural network training algorithms [10].

The paper of Geva & Sitte presents control laws that are linear in the state variables of the inverted pendulum, for both bang-bang and proportional control strategies. The experiment reveals that it is easy to find by random search in weight space, single neuron controllers that achieve the fundamental control objectives of maintaining the pole upright and bringing the cart to the centre of the track.

Geva & Sitte claimed that demonstration of supervised learning is no longer needed for the cart-pole problem because a linear control law is sufficient. Therefore a single neuron is sufficient for a satisfactory controller.

2.8.3.1. The Dynamics of the Inverted Pendulum by Geva & Sitte.

As has been described already the earliest application of neural networks to the inverted pendulum was conducted by Widrow and Smith [6]. Their analysis was based on a traditional control approach. They assumed that the applied control force F is a linear function of four state variables $(x, \dot{x}, \theta, \dot{\theta}$, with constant coefficients $W_1 \ldots W_4)$.

$$\mathbf{F} = \mathbf{k}\,\mathrm{sgn}(W_1 \cdot \theta + W_2 \cdot \dot{\theta} + W_3 \cdot x + W_4 \cdot \dot{x})$$

The state of the system was sampled at regular intervals and classified. The output of the classification determined the sign of the fixed force k, that was applied to the cart for the duration of sampling interval. For the linearized dynamic equations the linear

control force $\mathbf{F} = \mathbf{k}(W_1 \cdot \theta + W_2 \cdot \dot{\theta} + W_3 \cdot x + W_4 \cdot \dot{x}) = \mathbf{k}(\mathbf{w}.\mathbf{s})$ minimises the quadratic error measured by the time integral of the square of the four state variables and the control force [19].

Geva and Sitte [10] recently conducted a qualitative analysis of the linear control law of the above equation. They found out that the weights $W\theta$ have to be positive. With this analysis they were able to dissect and understand the control strategy embodied in the linear control law. From their experiments it was found out that if the cart is in equilibrium at the centre of the track, and the pole is leaning at a positive angle with no angular velocity then the control action is $\mathbf{F} = \mathbf{k} W_1 \theta$. In this case only positive force will erect the pole, and W_1 has to be positive. Similarly, if the pole angle is zero, but the angular velocity is positive and the cart is in equilibrium at the centre then only positive W_2 will produce the force that reduces the angular velocity.

On the other hand if the cart is in equilibrium somewhere on the right side of the track , and the pole is balanced perfectly, then the control force is determined by W_3. A positive weight will cause the force to accelerate the cart away from the centre of the track. This action will initially move the cart further away from the centre. However, as a result of this action the pole will start falling to the left, making the angular position and velocity negative. As the angular position and velocity become more negative their negative contribution to the force equation $\mathbf{k}(\mathbf{w}.\mathbf{s})$, attempting to balance the pole, will overcome the positive contribution from x. The net effect over time of the spoiling effect of W_3, and correcting effect of W_1 and W_2 is to accelerate the cart towards the centre. To verify how much this net effect happens consider the pole in a stationary upright position. Because of gravity, a sequence of **m** control actions in the reverse direction are required to compensate the work done by the gravity during the fall and the pull of gravity during recovery. Thus, the inverted pendulum receives a net acceleration in the direction opposite to the initial direction of force application.

When the cart is accelerating towards the centre of the track it will eventually overshoot. In order to bring it back to the centre another opposite sequence of control actions is necessary. Because of this condition the cart will continue oscillating about the

centre of the track. To stop the oscillation, a mechanism like damping is needed. The weight W_4 provides exactly this mechanism. The analysis of this is as follows. Suppose the pole is balanced and the cart is at the centre of the track moving with constant velocity to the right. Positive weight W_4 induced braking through angular contribution, in the same way as positive weight W_3 induced centring. The cart will accelerate to the right because of the contribution of the velocity to the force , hence the cart velocity also will increased. However, the increase of cart velocity to the right will cause the pole to fall to the left. So the next action of the control is to balance the pole. This control action produce the desired net result of slowing down the cart.

With the knowledge of the inverted pendulum dynamics using a linear control strategy Geva and Sitte [10] concluded the following; The angular position weight W_1 and angular velocity weight W_2 work towards maintaining the pole in a balance position. The horizontal displacement weight W_3 indirectly causes the cart to accelerate towards the centre of the track. Finally, the velocity weight W_4 indirectly slows down the cart, by causing the pole to lean in a direction opposite to the direction of movement.

2.8.3.2. Random Searches in Weight Space

Geva and Sitte raise the possibility that a random search in weight space might be effective in balancing the pole. To test this hypothesis they generated 10,000 unitary weight vectors with random orientations. Linear controllers with these weight vectors were tested in a computer simulation for their ability to prevent the cart-pole from failing within the first 300 s after release from various initial conditions. The parameters used with this simulation were those of Barto et al [5]. The control force was updated at every integration time step of 0.02 s (50 Hz sampling). The controllers were tested in the bang-bang and continuous force control mode. The maximum force deliverable by the motor is limited to 10 N with K=50. Because Geva and Sitte knew that the weight

vectors have to have positive components they used a second population of 10,000 vectors chosen from the positive quadrant. The results shows that one out of twelve random weight vectors could balance the pole for at least 5 minutes in the bang-bang regime when the cart was released from the centre of the track and with the pole in equilibrium. For continuous force this initial condition is trivial since the control force is always zero and there is nothing in the simulations to break the unstable equilibrium.

Almost as many controllers from the totally random population pass the test as from the positive quadrant population. The explanation for it according to Geva and Sitte is that controllers with small negative W_3 and W_4 will survive the first 5 minutes, although they will not stay at the centre but rather oscillate or drift away slowly. When the initial condition is made slightly more difficult by releasing the cart at 1 meter to the right of the centre, the controllers of dubious quality no longer pass the test. When the knowledge that the weights have to be positive is discarded (search over all orientations) controllers that pass the test from a difficult initial position become hard to find. Figures 2.19, 2.20, and 2.21 show the results of Geva and Sitte's experiment.

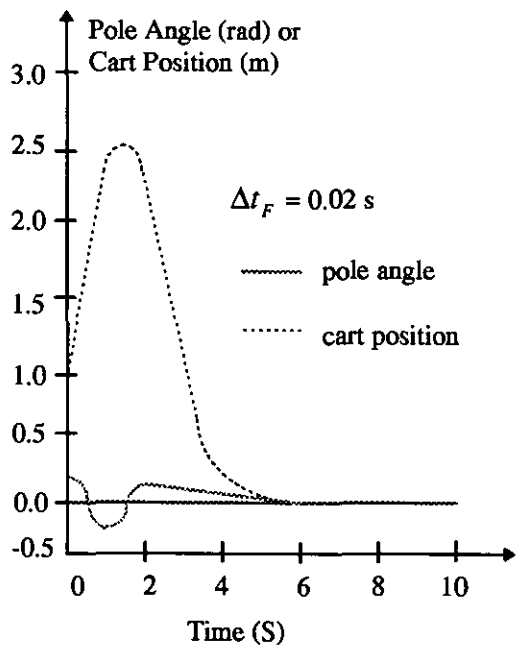


Figure 2.19

Cart position and pole angle versus time for the best controller in the continuous force regime, force update interval Δt_F *= 0.02s. Initial Condition* x *= 1 m,* v *= 1 m/s,* θ *= 0.1 rad, and* ω *= 0.2 rad/s.*
rms values over first 1000 s:

$\bar{x}$ *= 0.0076m,* $\bar{\theta}$ *= 0.0768 rad*

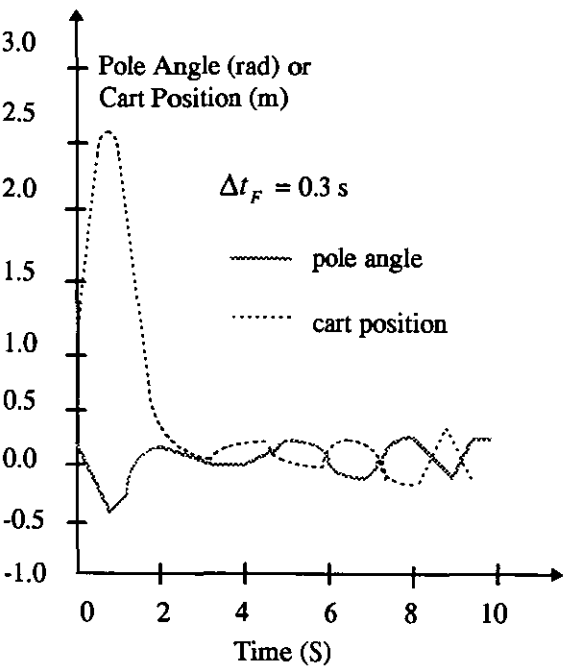


Figure 2.20

Cart position and pole angle versus time for the best controller in the continuous force regime, force update interval Δt_F *= 0.3s. Same initial condition as in figure 2.19*

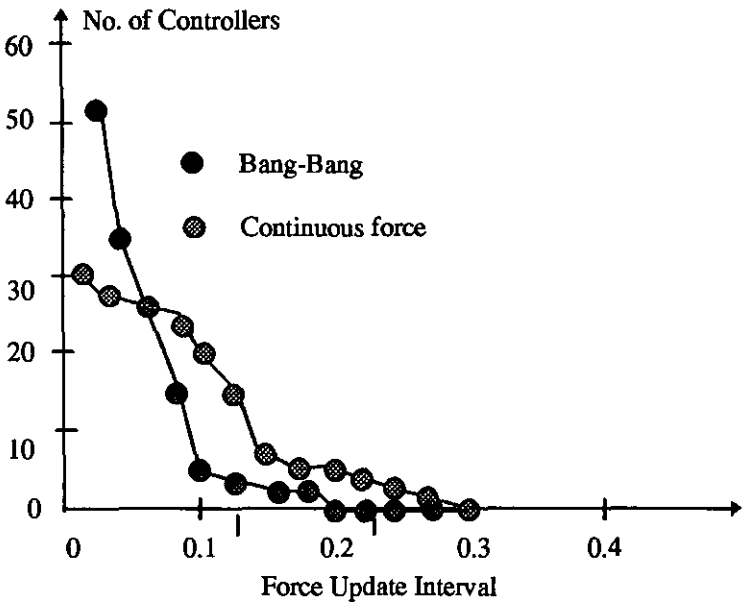


Figure 2.21

Number of controllers, from the sample of 10,000 random unitary weight vectors with positive components, that prevent the cartpole from failing within 5 minutes for increasing force update intervals. Initial conditions : x = 1m, v = 1 m/s, θ = 0.17 rad, ω = 0.1 rad/s.

2.9. Limitations of the Inverted Pendulum as a Benchmark for Learning Controllers

It has been known for many years that a linear control law, implemented by a single artificial neuron can control the inverted pendulum. An investigation of the literature on unsupervised learning methods for the inverted pendulum controllers reveals that it is hard to compare the published results. Also, it shows that most of the methods used present no clear evidence of better performance than the random search method.

What was not recognised before was that the random search in weight space can quickly uncover coefficients (weights) for controllers that work over a wide range of initial conditions [10]. This was tested by Geva and Sitte using linear controllers with those weight vectors in a computer simulations (see section 2.8.3.2). The result of random search in weight space indicates that success in finding a satisfactory neural controller is not sufficient proof for the effectiveness of unsupervised learning method.

Dissecting the dynamics of the inverted pendulum system it is obvious that it has limitations. The system has only two degrees of freedom. Hence, its capability and its ability to represent complex problems is limited. For example, robots need extra degrees of freedom to avoid degeneracies [15] or to manoeuvre past obstacles in the environment [16]. The techniques using neural networks to solve the control problem lacks stability. This is because disturbances and uncertain initial conditions are not well defined in the experiments. What happens if the interval between control force updates is increased or if unequal magnitude forces are applied? The authors above assume that the system is frictionless, in actual or physical simulations, friction does really exist. The controller developed also lacks flexibility and adaptability. Successful research in neural network learning control did not mention about what happens to the system if external disturbances are applied to the plant.

2.10. Summary

This chapter gave an overview of the problem and limitations imposed by the present methods of modelling a control system. A review of the concepts used in developing a conventional and an intelligent control system had been discussed. Special attention has been paid to the rigid pole-cart balancing problem (inverted pendulum) as a benchmark for learning controllers. A comparison of the works of different authors addressing this problem has been presented. It is pointed out that there are limitations on this problem, as, controlling this system is not a difficult nonlinear problem, and the learning controller developed may have limited application to manufacturing industries.

Armed with this knowledge therefore, there are many issues that remain unsolved, that need further investigation. For example, can the techniques developed for learning control applied to a more complex nonlinear system? Can we scale up this new techniques to larger, more complex system without suffering from local minima problems (local minima are the inevitable problem in least minimum squares error methods of learning [7]). To answer these questions it would therefore be better to conduct future work using a model problem which is more complex than the current inverted pendulum problem.

The author therefore conducted research on inverted pendulum learning control using an elastic pole. This type of pole gives an additional degree of freedom to the system, e.g. its elastic transverse displacement and therefore has much more complex dynamics.

The next chapters of this thesis concentrate on the discussion and presentation of the development and test of learning controllers to balance a flexible pole hinged on top of a cart moving along a limited track.

CHAPTER 3

A Model of a Flexible Pole-Cart Balancing System Under its First Mode of Vibration

3.1. Introduction

Flexible beams have been a topic of research in the field of robotics since the early 1970's [37]. Beams of this type have been used to model flexibility in robotic members, a phenomenon that has gained an importance as a result of widespread attempt to lighten robotic assemblies for increased speed and efficiency [38]. The present day industrial robot is easy to control because it is designed to be very heavy, rigid and slow. This, however, gives high weight to payload ratios which increase cost and decrease the speed of the robot. To improve this ratio, several researchers have proposed the use of lightweight robots with links that are allowed to flex during operation [33,34,35,36].

When compared with the traditional robot manipulators constructed from rigid links, flexible robot manipulators have many advantages; among them are [30]; the moving of larger payloads without increasing the mass of the linkages, requirements for less material and smaller actuators, less link weight, less power consumption, and the machines are more maneuverable and transportable. Flexible robot manipulators are not presently used in production industries because robot manipulators are required to have a reasonable accuracy in the response of the manipulator end-effector to the input command from its control system. The experiments described in [30,31,37,38] were directed towards developing controller for flexible robot manipulators. Building this type of controller is a difficult and very challenging task. One major step in making this controller is to analyze the dynamic behavior of the system. Computer simulation is necessary to evaluate whether the derived dynamics of the system are correct. It is

therefore most appropriate to study analogues of such systems. The flexible pole-cart system provides such an analogue.

This chapter presents a rule based control system for the flexible pole-cart balancing problem (the inverted pendulum using an elastic pole) that operates on a simulation of the system. The task of this system is to balance an elastic pole that is hinged on a movable cart. It is assumed that the hinge is frictionless. The cart is allowed to move along a track with limited length and that has friction. Forces of different magnitude are applied to the cart in either a left or right direction. The initial angle of the pole can be varied up to 30 degrees. This is more difficult than the conventional rigid pole-cart system because of the complexity in its dynamics. The deflection of the elastic pole gives additional degrees of freedom to this system. A computer simulation of the use of the cart to balance a flexible pole under first mode of vibration is presented here. The dynamic equations of the system were derived using Newton's laws, Bernoulli-Euler analysis, and beam theories. The system was analyzed with the presence of friction. Numerical integration using fourth order Runge-Kutta was conducted. Computer graphics of the cart balancing the pole along the track in real time have been made and are shown. Results on the analysis of the behavior of the system under various conditions has also been obtained in order to explore the practicality of attempting to control such a system.

The chapter begins by presenting the mechanics of the system. It then continues by describing a simulation of these mechanics. The chapter closes by describing the operation of the rule based controller on the simulated dynamics. The code for simulations program and the controller are presented in full in Appendix A.

3.2. Mechanics

This section discusses the dynamics of the flexible pole-cart balancing system. The analysis of the system is based on the dynamics of the rigid pole-cart, together with beam theories.

3.2.1. Diagram of the Flexible Pole-Cart Balancing System.

Figure 3.1 shows the dynamics of the system. The free body diagrams of the system are shown in figures 3.2 and 3.3.

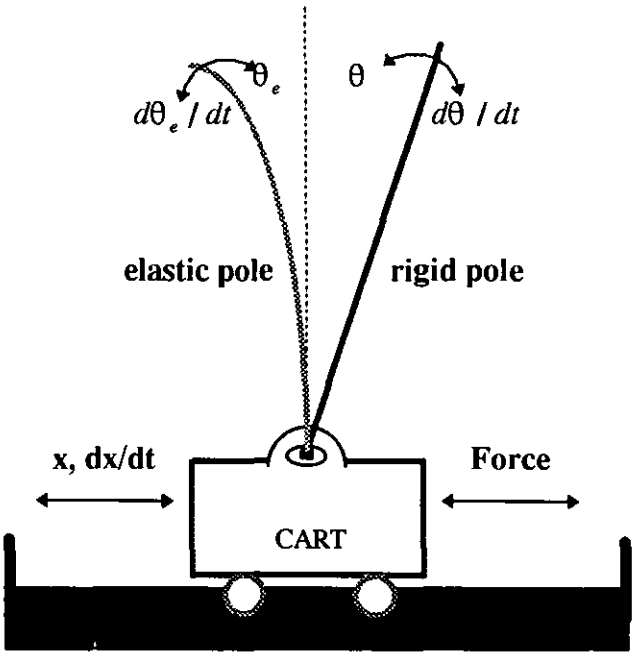


Figure 3.1
The flexible pole-cart balancing system

FREE BODY DIAGRAM

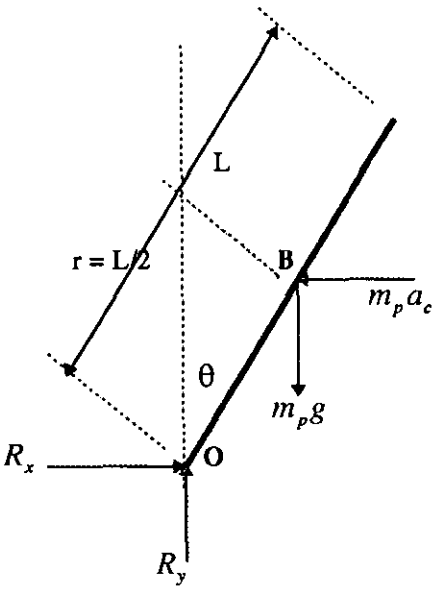


Figure 3.2
Forces acting on the pole

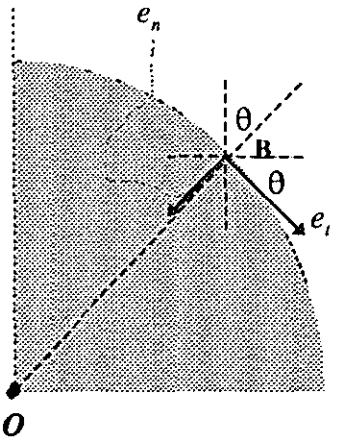


Figure 3.3
Tangential and normal forces acting on the center of the pole

3.2.2. Derivation of the Equations

3.2.2.1. Solution for Rigid Pole Angle, Velocity, Acceleration, and Cart Velocity.

This section presents the analysis that relates the motion of the cart to the motion of the pole. It begins by considering the dynamics of the rigid pole.

Let

m_p = mass of the pole.

m_c = mass of the cart.

a_c = acceleration of the cart.

L = total length of the pole.

θ = the angle of the pole from y axis.

$\dot{\theta}$ = angular velocity of the pole.

$\ddot{\theta}$ = angular acceleration of the pole.

g = acceleration due to gravity

Applying Newton's law to the rigid pole:

Summation of forces at the center of the pole = (mass)(acceleration of pole)

$$\sum \hat{F} = m_p a^B \tag{3.1}$$

Using definitions from figures 3.2 and 3.3.

e_t = a unit vector for tangential component.

e_n = a unit vector for normal component.

a_t = tangential acceleration.

a_n = normal acceleration; then

$$e_t = \cos(\theta) * e_x - \sin(\theta) * e_y \tag{3.2}$$

$$e_n = -\sin(\theta)^* e_x - \cos(\theta)^* e_y \tag{3.3}$$

$$a_t = r\alpha = \frac{L}{2} * \ddot{\theta} \tag{3.4}$$

$$a_n = rw^2 = \frac{L}{2} * \dot{\theta}^2 \tag{3.5}$$

and using equations 3.2 to 3.5:

$$a^B = a_t * e_t + a_n * e_n$$

$$a^B = \frac{L}{2}\ddot{\theta}(\cos(\theta)e_x - \sin(\theta)e_y) + \frac{L}{2}\dot{\theta}^2(-\sin(\theta)e_x - \cos(\theta)e_y) \tag{3.6}$$

then from figure 3.2:

$$\sum \hat{F} = (-m_p a_c + R_x)e_x + (R_y - m_p g)e_y = m_p a^B$$

$$\left(-m_p a_c + R_x\right)e_x + \left(R_y - m_p g\right)e_y = m_p\{\frac{L}{2}\ddot{\theta}(\cos(\theta)e_x - \sin(\theta)e_y) +$$

$$\frac{L}{2}\dot{\theta}^2(-\sin(\theta)e_x - \cos(\theta)e_y)\}$$

Equating terms of the e_x and e_y components:

$$-m_p a_c + R_x = m_p\{\frac{L}{2}\ddot{\theta}\cos(\theta) + \frac{L}{2}\dot{\theta}^2(-\sin(\theta))\}$$

$$-m_p a_c + R_x = m_p\{\frac{L}{2}\ddot{\theta}\cos(\theta) - \frac{L}{2}\dot{\theta}^2\sin(\theta)\}$$

$$-m_p a_c + R_x = m_p\frac{L}{2}\{\ddot{\theta}\cos(\theta) - \dot{\theta}^2\sin(\theta)\} \tag{3. 8}$$

$$R_y - m_p g = m_p\{\frac{L}{2}\ddot{\theta}(-\sin(\theta) - \frac{L}{2}\dot{\theta}^2\cos(\theta)\} \tag{3.9}$$

From the Euler equations, the summation of the moments at point O is equal to the product of the moment of inertia (I) and the angular acceleration (α) of the pole.

$$\sum M^{o} = I^{o}\alpha \tag{3.10}$$

where:

$$I^{o} = \frac{1}{3} m_p L$$

$$\alpha = \ddot{\theta}$$

Therefore using figure 3.2 apply equation 3.10:

$$m_p g \frac{L}{2}\sin(\theta) - m_p a_c \frac{L}{2}\cos(\theta) = I^{o}\ddot{\theta} = \frac{1}{3} m_p L^2 \ddot{\theta}$$

$$a_c \frac{L}{2}\cos(\theta) = \frac{1}{3} L^2 \ddot{\theta} - g\frac{L}{2}\sin(\theta)$$

since $L/2 = r$

$$a_c r\cos(\theta) = \frac{4}{3} r^2 \ddot{\theta} - gr\sin(\theta) \tag{3.11}$$

Equation 3.11 shows the relationship of the cart acceleration to the angular acceleration of the pole.

To establish the forces acting on the cart it is assumed that the mass of the wheels is very small compared to the mass of the cart and the pole. Figure 3.4 shows the free body diagram of the cart.

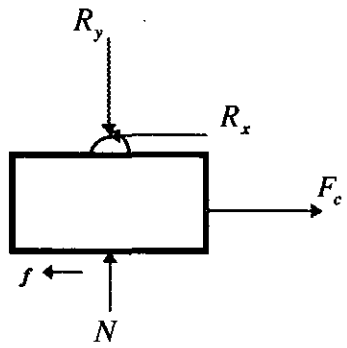


Figure 3.4
Forces acting on the cart

Applying Newton's law to figure 3.4:

$$\sum F = m_c a_c$$

$$F_c - R_x - f = m_c a_c$$

$$R_y = N \quad ; \quad f = \mu N$$

$$F_c = m_c a_c + R_x + \mu R_y \tag{3.12}$$

This is the force needed to move the cart.

Then substituting equations 3.8 & 3.9 into 3.12

$$F_c = m_c a_c + m_p a_c + m_p \frac{L}{2}(\ddot{\theta}\cos\theta - \dot{\theta}^2 \sin\theta) \; +$$

$$\mu(m_p g + m_p \frac{L}{2}(-\ddot{\theta}\sin\theta - \dot{\theta}^2 \cos\theta))$$

$$F_c = (m_c + m_p)a_c + \mu m_p g + \ddot{\theta}\, m_p \frac{L}{2}(\cos\theta - \mu\sin\theta) \; +$$

$$\dot{\theta}^2 m_p \frac{L}{2}(-\sin\theta - \mu\cos\theta)$$

$$a_c = \frac{F_c - \{\mu m_p g + \ddot{\theta} m_p \frac{L}{2}(\cos\theta - \mu\sin\theta) + \dot{\theta}^2 m_p \frac{L}{2}(-\sin\theta - \mu\cos\theta)}{(m_p + m_c)}$$

Let

$$m = m_p + m_c$$

$$r = L/2$$

then

$$a_c = \frac{F_c - \{\mu m_p g + \ddot{\theta} m_p r(\cos\theta - \mu\sin\theta) + \dot{\theta}^2 m_p r(-\sin\theta - \mu\cos\theta)}{m} \qquad (3.13)$$

and substituting 3.13 to 3.11

$$(r\cos\theta)\frac{F_c - \{\mu m_p g + \ddot{\theta} m_p r(\cos\theta - \mu\sin\theta) + \dot{\theta}^2 m_p r(-\sin\theta - \mu\cos\theta)}{m} =$$

$$\frac{4}{3}r^2\ddot{\theta} - gr\sin\theta$$

$$\frac{4}{3}r\ddot{\theta} - \ddot{\theta}\frac{m_p r\cos\theta(\cos\theta - \mu\sin\theta)}{m} = g\sin\theta -$$

$$\frac{\cos\theta}{m}\{F_c - [\mu m_p g - \dot{\theta}^2 m_p r(\sin\theta + \mu\cos\theta)]\}$$

$$\ddot{\theta}\left[\frac{4}{3}mr - m_p r\cos^2\theta + \mu m_p r\cos\theta\sin\theta\right] = mg\sin\theta -$$

$$\cos\theta\{F_c - [\mu m_p g - \dot{\theta}^2 m_p r(\sin\theta + \mu\cos\theta)]\}$$

$$\ddot{\theta} = \frac{mg\sin\theta - \cos\theta\{F_c - [\mu m_p g - \dot{\theta}^2 m_p r(\sin\theta + \mu\cos\theta)]\}}{\frac{4}{3}mr - m_p r\cos^2\theta + \mu m_p r\cos\theta\sin\theta} \quad (3.14)$$

This represents the angular acceleration of the rigid pole hinge root on top of the cart that move dependent on the magnitude and direction of the applied force F_c.

3.2.2.2. Solution for Elastic Pole Angle, Velocity, and Acceleration.

It is now necessary to extend this analysis to include the elastic pole.

Let

θ_t = total elastic pole's angle from the vertical axis.

$\dot{\theta}_t$ = elastic pole's velocity.

$\ddot{\theta}_t$ = elastic pole's acceleration

For the elastic pole, it is assumed that the total angle θ_t (the pole's actual position with respect to vertical axis) is equal to the actual angle of the rigid pole θ plus the pole's angle due to its elastic deflection θ_e.

$$\theta_t = \theta + \theta_e \quad (3.15)$$

To find θ_e it is assumed that the pole behaves as a cantilever with uniform distributed load as is shown in figure 3.5. From [40]

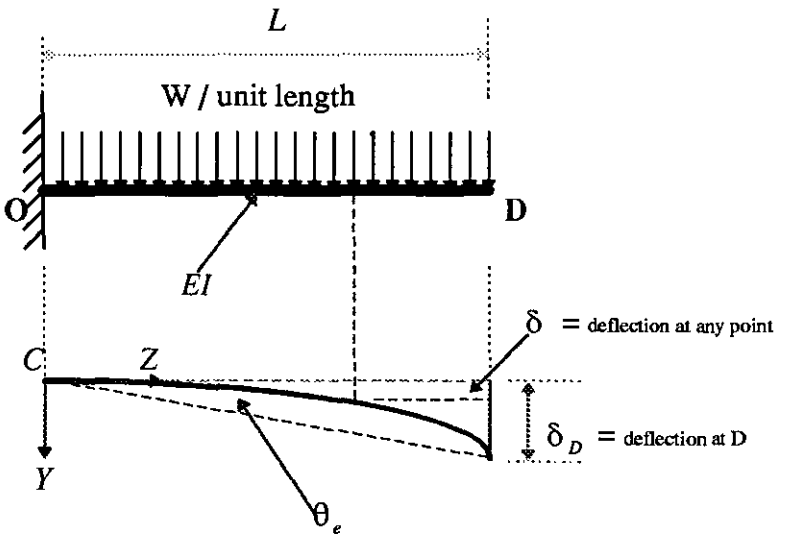


Figure 3.5

Cantilever carrying a uniformly distributed load

E = Young's modulus of elasticity.

I = Second moment of area

= $BD^3/12$

B = Breadth of the beam

D = Depth of the beam

W = Weight per unit length

The bending moment at a distance Z from C is :

$$M = \frac{-1}{2}W(L-Z)^2$$

but

$$-M = EI\frac{d^2\delta}{dZ^2}$$

hence;

$$EI\frac{d^2\delta}{dZ^2} = \frac{1}{2}(L-Z)^2 = \frac{1}{2}W(L^2 - 2LZ + Z^2)$$ Integrating this equation gives

$$EI\frac{d\delta}{dZ} = \frac{1}{2}W(L^2Z - LZ^2 + \frac{1}{3}Z^3 + A)$$ further integrating this equation gives

$$EI\delta = \frac{1}{2}W(\frac{1}{2}L^2Z^2 - \frac{1}{3}LZ^3 + \frac{1}{12}Z^4 + AZ + B)$$

At the built in end, Z = 0, and we have

$$\frac{d\delta}{dZ} = 0 \quad \text{and } \delta = 0\,, \text{ Thus A = B = 0}$$

Then

$$EI\delta = \frac{1}{24}W(6L^2Z^2 - 4LZ^3 + Z^4)$$

$$\delta = \frac{1}{24}W(6L^2Z^2 - 4LZ^3 + Z^4)/(EI) \qquad \textbf{(3.16)}$$

At the free end D, $Z = L$

$$\delta_D = \frac{WL^4}{8EI} \qquad \textbf{(3.17)}$$

It is now necessary to fit the cantilever analysis to the flexible pole. To find the value of W within the cantilever analysis, the pole can be analyzed by considering both the concentrated and uniform load as shown in figures 3.6 and 3.7.

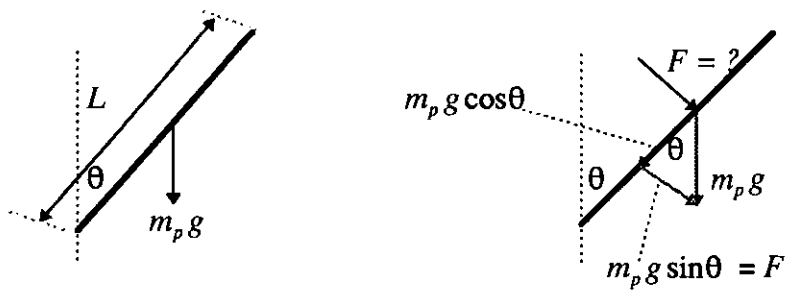


Figure 3.6
Concentrated load of the pole at any position

For the pole at uniform load using F at figure 3.6:

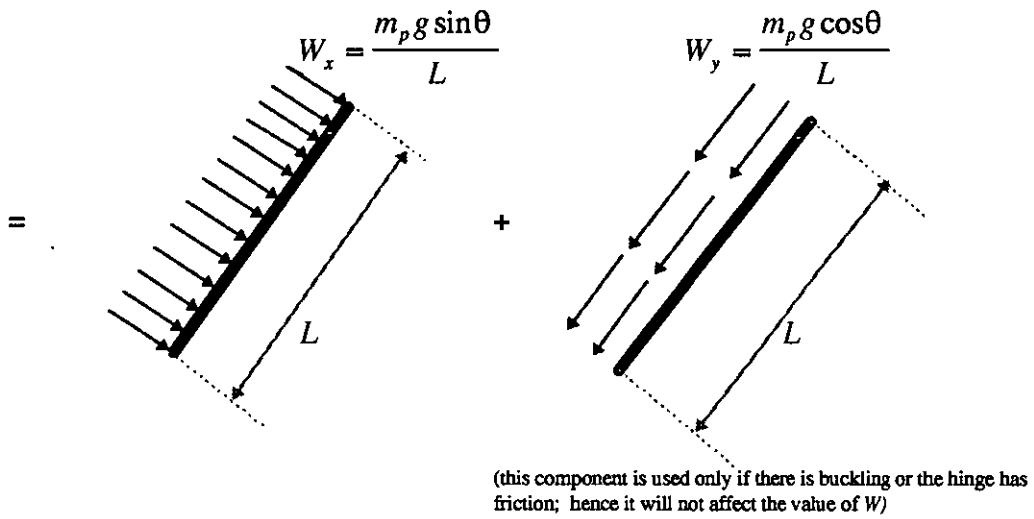


Figure 3.7
Uniform load of the pole at any position

Therefore :

$$W = \frac{m_p g \sin\theta}{L} \tag{3.18}$$

To determine the total elastic pole angle θ_t, at any time, consider figure 3.8 below.

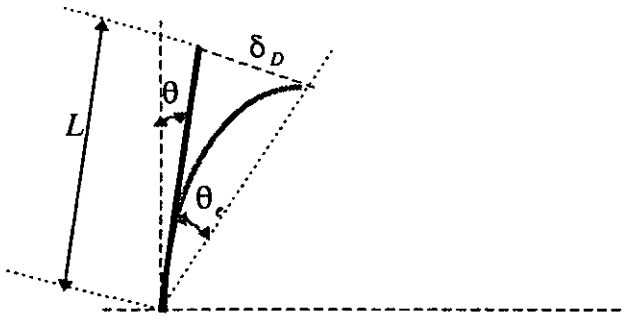


Figure 3.8
Position of the rigid and elastic pole at any time t

$$\tan\theta_e = \frac{\delta_D}{L} \quad ; \quad \text{from equation 3.17}$$

$$\tan\theta_e = \frac{WL^4}{8EIL} = \frac{WL^3}{8EI} \quad ; \text{ from equation 3.18}$$

$$\tan\theta_e = \frac{L^2 m_p g \sin\theta}{8EI} \quad ; \quad \text{but} \quad I = BD^3/12$$

$$\tan\theta_e = \frac{12L^2 m_p g \sin\theta}{8EBD^3} \; ; \text{ let } K = \frac{12L^2 m_p g}{8EBD^3}$$

$$\tan\theta_e = K\sin\theta \; ; \text{ hence}$$

$$\theta_e = \tan^{-1}(K\sin\theta) \qquad \textbf{(3.19)}$$

Substitute equation 3.19 into equation 3.15

$$\theta_t = \theta + \tan^{-1}(K\sin\theta) \qquad \textbf{(3.20)}$$

To find the elastic pole's angular velocity, differentiate 3.20:

$$\dot{\theta}_t = \dot{\theta} + \left(\frac{K\cos\theta}{1+(K\sin\theta)^2}\right)\dot{\theta}$$

$$\dot{\theta}_t = \dot{\theta}\left\{1 + \left(\frac{K\cos\theta}{1+(K\sin\theta)^2}\right)\right\} \qquad \textbf{(3.21)}$$

To find the elastic pole's angular acceleration, differentiate 3.21.

$$\ddot{\theta}_t = \ddot{\theta}\left[1 + \frac{K\cos\theta}{1+(K\sin\theta)^2}\right] +$$

$$\dot{\theta}\left[\frac{(-K\sin\theta)(\dot{\theta})(1+(K\sin\theta)^2) - (K\cos\theta)(2K\sin\theta)(K\cos\theta)(\dot{\theta})}{(1+(K\sin\theta)^2)^2}\right]$$

$$\ddot{\theta}_t = \ddot{\theta}\left[1+\frac{K\cos\theta}{1+(K\sin\theta)^2}\right]+$$

$$\dot{\theta}^2\left[\frac{(-K\sin\theta)(1+(K\sin\theta)^2)-(K\cos\theta)(2K\sin\theta)(K\cos\theta)}{(1+(K\sin\theta)^2)^2}\right]$$

$$\ddot{\theta}_t = \ddot{\theta}\left[1+\frac{K\cos\theta}{1+(K\sin\theta)^2}\right]+ \dot{\theta}^2\left[\frac{K\sin\theta(-1-K^2(\sin\theta)^2-2K^2(\cos\theta)^2)}{(1+(K\sin\theta)^2)^2}\right]$$

$$\ddot{\theta}_t = \ddot{\theta}\left[1+\frac{K\cos\theta}{1+(K\sin\theta)^2}\right]+ \dot{\theta}^2\left[\frac{-K\sin\theta\{1+K^2((\sin\theta)^2+2(\cos\theta)^2)\}}{(1+(K\sin\theta)^2)^2}\right]$$

$$\ddot{\theta}_t = \ddot{\theta}\left[1+\frac{K\cos\theta}{1+(K\sin\theta)^2}\right]+ \dot{\theta}^2\left[\frac{-K\sin\theta(1+K^2(1+(\cos\theta)^2)}{(1+(K\sin\theta)^2)^2}\right] \quad \textbf{(3.22)}$$

This presents the angular acceleration of the flexible pole hinge on top on the moving cart.

3.2.2.3. Solution for Cart Acceleration and Displacement due to Balance the Elastic Pole.

This section discusses the mechanics to find the displacement of the cart due to the applied force in order to balance the elastic pole. The acceleration of the cart in order to balance the pole is derived from equation 3.13 with $\ddot{\theta}$, $\dot{\theta}$, θ replaced by $\ddot{\theta}_t$, $\dot{\theta}_t$, θ_t respectively. Hence,

$$a_c = \frac{F_c - \{\mu m_p g + \ddot{\theta}_t m_p r(\cos\theta_t - \mu\sin\theta_t) + \dot{\theta}_t^2 m_p r(-\sin\theta_t - \mu\cos\theta_t)}{m} \quad \textbf{(3.23)}$$

From the computer simulation using numerical integration (fourth order Runge-Kutta) the value of the acceleration a_{ce} is found to be a cosine function (refer to section 3.3.4 figures 3.17c, 3.18c, and 3.19c). The reason for this is that the acceleration of the cart is dependent upon the force applied to it. This force is being controlled in order to balance the pole and it is experimentally observed to be periodic. At time t equal to zero initial force is already applied to the cart. Because of this, at this point in time, the cart is already accelerating at a magnitude equivalent to force/mass.

Thus, the acceleration of the cart for balancing the flexible pole at any time t is:

$$a_{ce} = k\cos\omega t \tag{3.24}$$

The velocity of the cart at any time t for balancing the flexible pole is obtained by integrating equation 3.24.

$$v_{ce} = \int_0^t a_{ce} = \int_0^t k\cos wt = \frac{k}{w}\sin wt \tag{3.25}$$

The displacement of the cart at any time t for balancing the flexible pole is obtained by integrating equation 3.25.

$$x_{ce} = \int_0^t v_{ce} = \int_0^t \frac{k}{w}\sin wt = \frac{-k\cos wt}{w^2}$$

but from equation 3.24 $a_{ce} = k\cos\omega t$, hence

$$x_{ce} = \frac{-a_{ce}}{\omega^2}$$

and

$$\omega = 2\pi f$$

f = *average frequency*

$$x_{ce} = \frac{-a_{ce}}{4\pi^2 f^2} \tag{3.26}$$

To find the frequency f it is necessary to obtain the total number of cycles during the total time of pole balancing. Below is the algorithm to determine this.

1. Determine the highest value of the acceleration for the entire time of balancing excluding the first one (a_{ch}).
2. Starting from time t > 0.0 record the value of time for the first a_{ch} ($Time_{ach1}$).
3. Record the time it takes to have another acceleration approximately equal to a_{ch} ($Time_{ach2}$)
4. Record the frequency from ($Time_{ach1}$) to ($Time_{ach2}$) = one cycle.
5. Repeat process 2 & 3 by substituting $Time_{ach2}$ to $Time_{ach1}$ for *I to N*.
   I & N can be any value of time from t > 0.0 to the final time of balancing the pole. Record the total number of cycles for this process (*tot_cycles*).
6. Get the sum of the time recorded from $Time_{ach1}$ to $Time_{achN}$ (*tot_time*).

$$tot\_time = \sum_{I=1}^{N} (Time_{achI} + Time_{achI+1})$$

7. Average frequency is

$$f = tot\_cycles \,/\, tot\_time \tag{3.27}$$

3.2.2.4. Solution for the Location of the Pole at Any Time in XY Plane.

This section discussed the mechanics to find the coordinates of the flexible pole on XY plane at any angle. See figure 3.9. The equations derived from this analysis are very important in displaying the pole graphically. Every point of the pole is plotted. This analysis uses equation 3.16 as its starting point.

Let

(x1,y1) = the coordinate at any point of the pole without elastic deflection (say **p1**).

(x2,y2) = the new coordinate of **p1** due to elastic deflection.

The deflection of the elastic pole at any point is derived from equation 3.16.

$$\delta = \frac{1}{24} W(6L^2Z^2 - 4LZ^3 + Z^4)/(EI)$$

From figure 3.9

The value of $L1$ is from 0.0 to L.

$$x1 = (\sin\theta)(L1) \tag{3.28}$$

$$y1 = (\cos\theta)(L1) \tag{3.29}$$

$$L2 = \sqrt{(L1)^2 + \delta^2} \tag{3.30}$$

$$\theta_e = \tan^{-1}\left(\frac{\delta}{L1}\right) \tag{3.31}$$

$$x2 = \sin(\theta + \theta_e)(L2) \tag{3.32}$$

$$y2 = \cos(\theta + \theta_e)(L2) \tag{3.33}$$

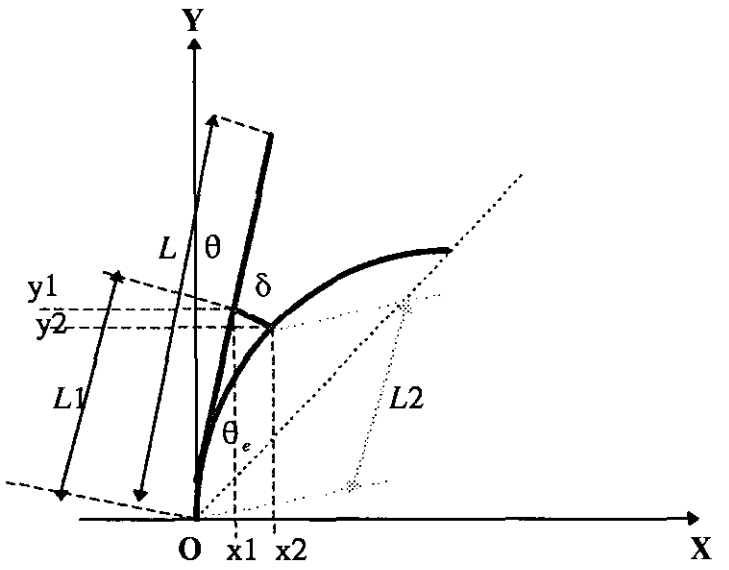


Figure 3.9

The coordinates of the elastic pole at any point in xy plane

3.3. Software Simulation

3.3.1. The Program

MAYMAY is a computer program that will simulate the pole-cart balancing system. This program is written in the Turbo Pascal language and implemented on an IBM PC machine. MAYMAY can simulate both rigid and elastic pole-cart balancing with or without friction. The program is a menu driven. The user has nine options to choose from the main menu (see figure 3.10).

Option number one is to simulate rigid pole-cart balancing and option number two is to simulate elastic pole-cart balancing. Both options one and two can be carried out either with or without friction. The user must enter data for these options. These data represent the characteristics of the pole, the system initial condition, and the simulation time (see tables 3.1 and 3.2). Once the data are entered into the computer, the program, using numerical integration (fourth order Runge-Kutta, see figure 3.11) calculates the values of the derived dynamic equations presented previously. The outputs of this process are the values at any given time of the angle of the pole from the vertical axis, the angular velocity and acceleration of the pole, the force applied to the cart, the velocity of the cart, the acceleration of the cart, and the displacement of the cart. All of this data are stored in an external file for future use. After calculating these values, the process then will go back to the main menu.

Option number four plots the behavior of the rigid pole at any given time. This is a graphic representation of the rigid pole's angle versus time. Option number five is similar to number four, but extends the simulation to an elastic pole. Option number six is a real time graphical pole-cart simulation. This process will display the cart moving along the track (forward and backward), and balancing the pole that is hinged at its root. Option number seven plots the graph of the displacement of the cart versus time. Option number eight presents the graph of the cart's acceleration versus time, and option number nine presents the graph of the cart's velocity versus time. The data for option 4 is

taken from the result of process 1, while for options 5, 6, 7, 8, and 9 it is taken from the result of process 2. Finally, to exit from the program, the user should choose option number zero.

Table 3.1

(The data that the user must enter for option 1 - rigid pole)
Mass of the pole (in kilograms)
Total mass of the pole and the cart (in kilograms)
Total length of the pole (in meters)
Initial force applied (in Newton)
Coefficient of friction
Step size (H) (the increment of integral calculation)
Upper limit of integration (tmax)
Freq. intermediate printouts (Ifreq) (display result for given increment)
Initial pole angle (theta in degrees)
Limitations of pole angle (in degrees)

Table 3.2

(The data that the user must enter for option 2 - elastic pole)
Mass of the pole (in kilograms)
Total mass of the pole and the cart (in kilograms)
Total length of the pole (in meters)
Breadth of the pole (in meters)
Depth of the pole (in meters)
Young's modulus - elasticity of the pole (in Newton/square meter)
Initial force applied (in Newton)
Coefficient of friction
Step size (H) (the increment of integral calculation)
Upper limit of integration (tmax)
Freq. intermediate printouts (Ifreq) (display result for given increment)
Initial pole angle (theta in degrees)
Limitations of pole angle (in degrees)

3.3.2. The Algorithm

This section presents the algorithm of program MAYMAY. Figure 3.10 is the complete structure of the program. Detailed algorithms of every option are shown in figures 3.11 to 3.15. The code is in appendix A.

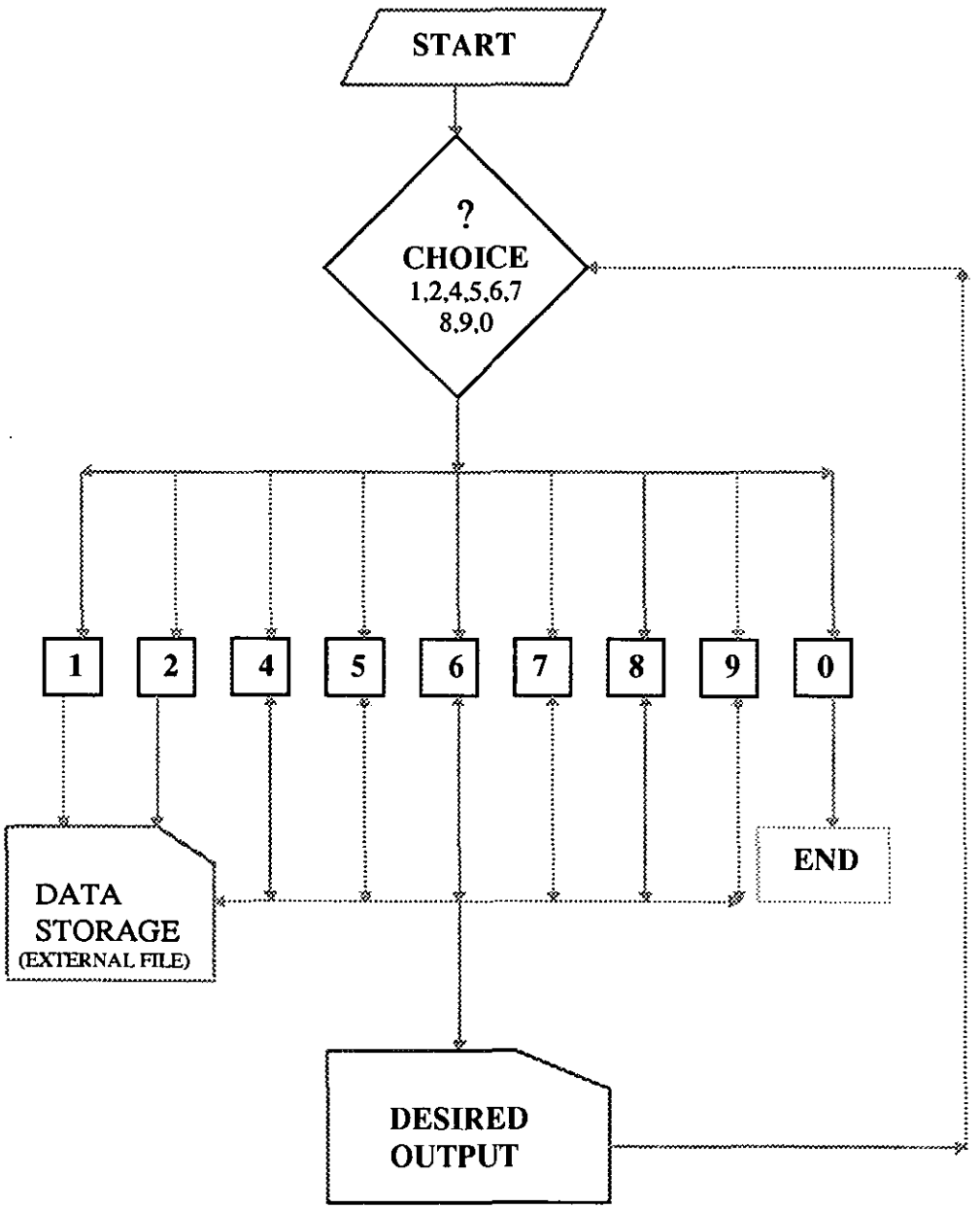


Figure 3.10
(Program MAYMAY)
Elastic pole-cart balancing system computer simulation

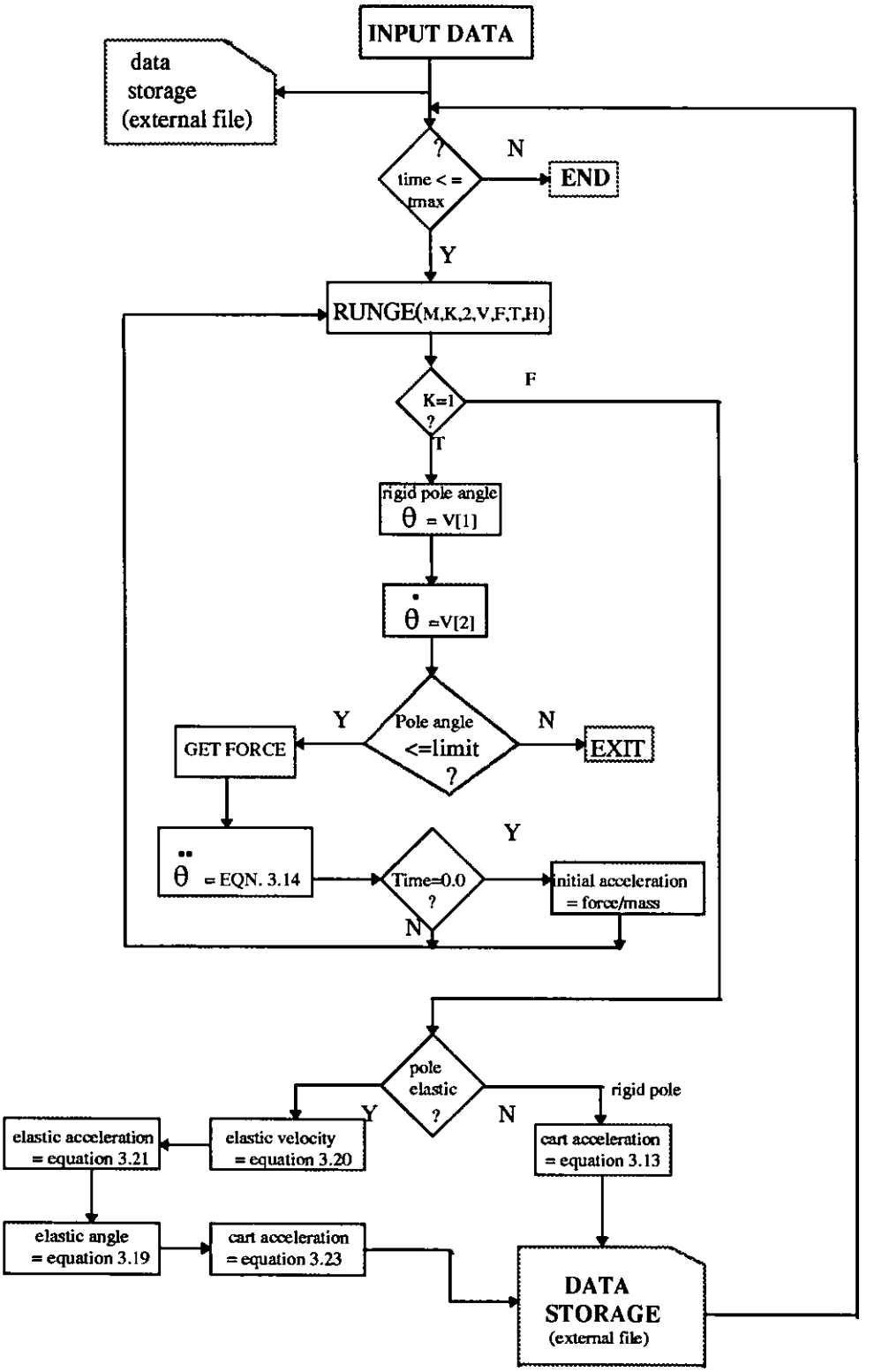


Figure 3.11
option 1 & 2
Numerical integration to find the behavior of elastic/rigid pole-cart balancing system

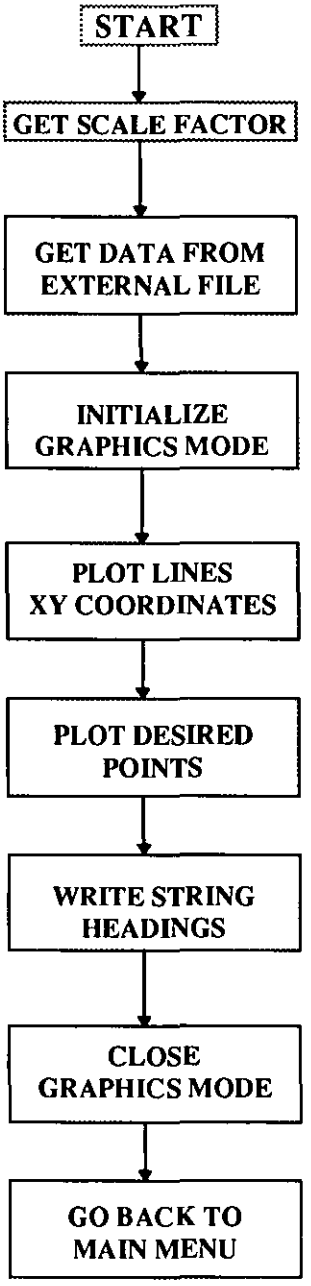


Figure 3.12
options 4, 5, 7, 8, and 9
Graphical representation of the behavior of the pole-cart balancing system

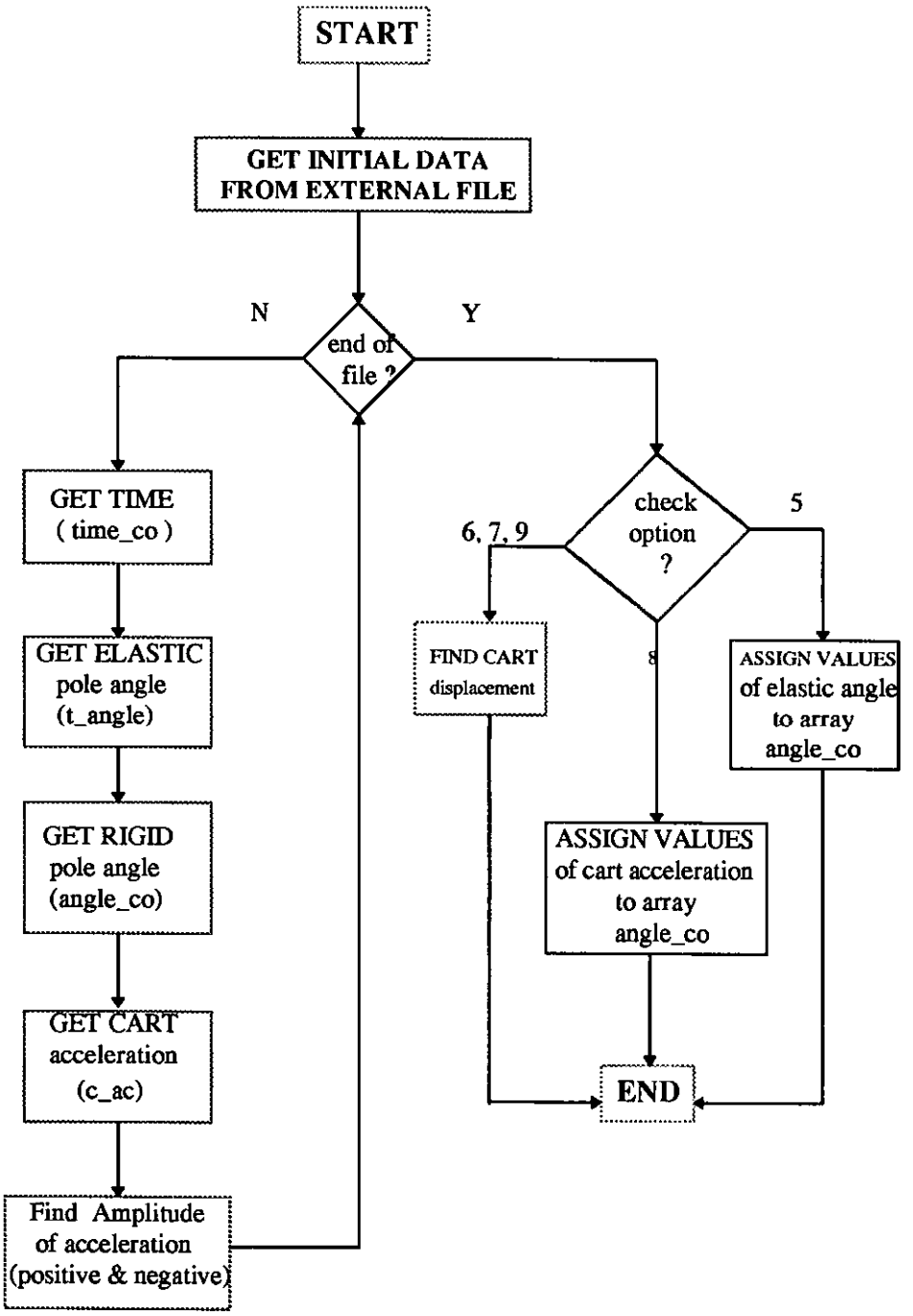


Figure 3.13
Process get data from external file

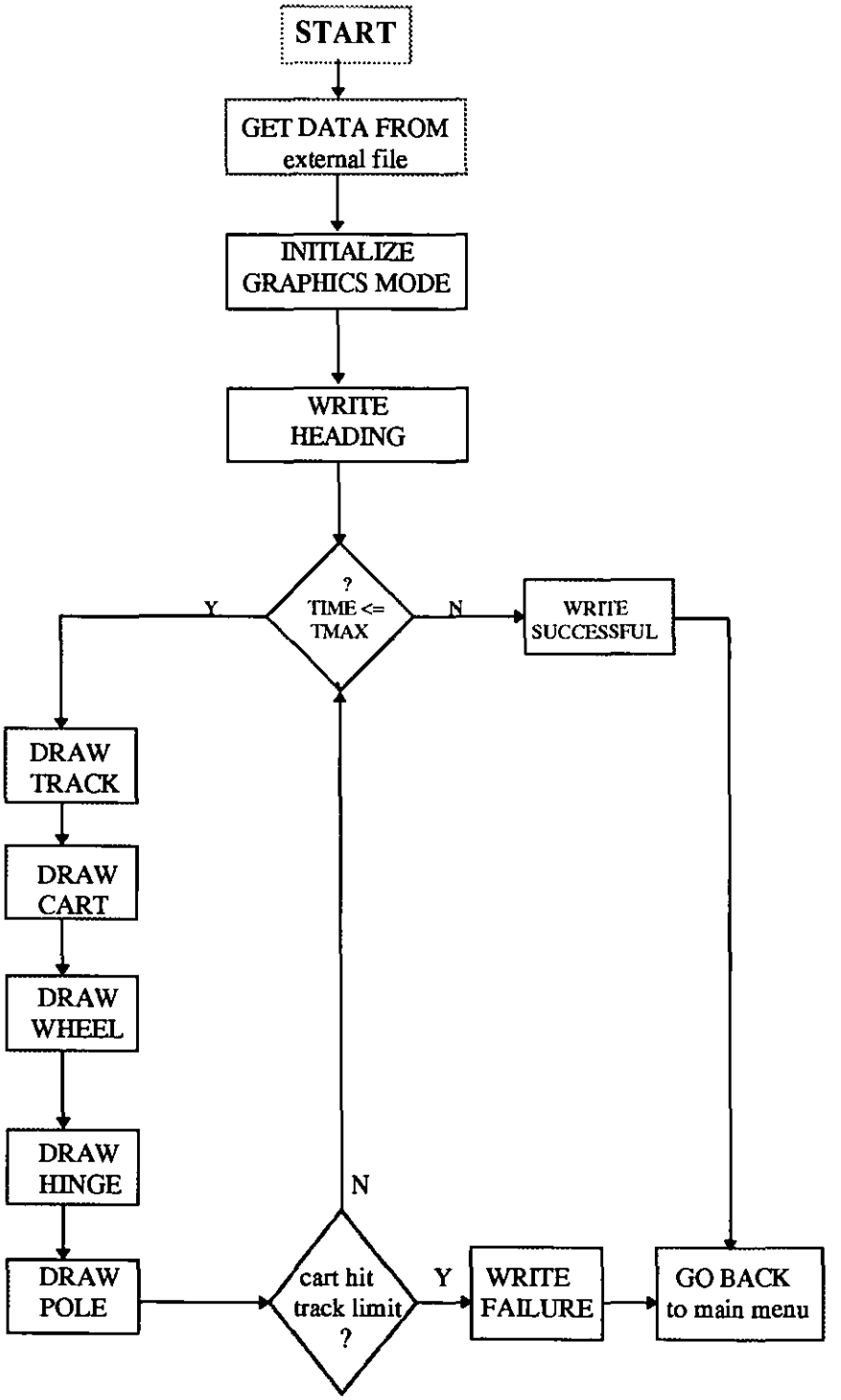


Figure 3.14
option 6
Real time simulation of the cart balancing the pole along a track

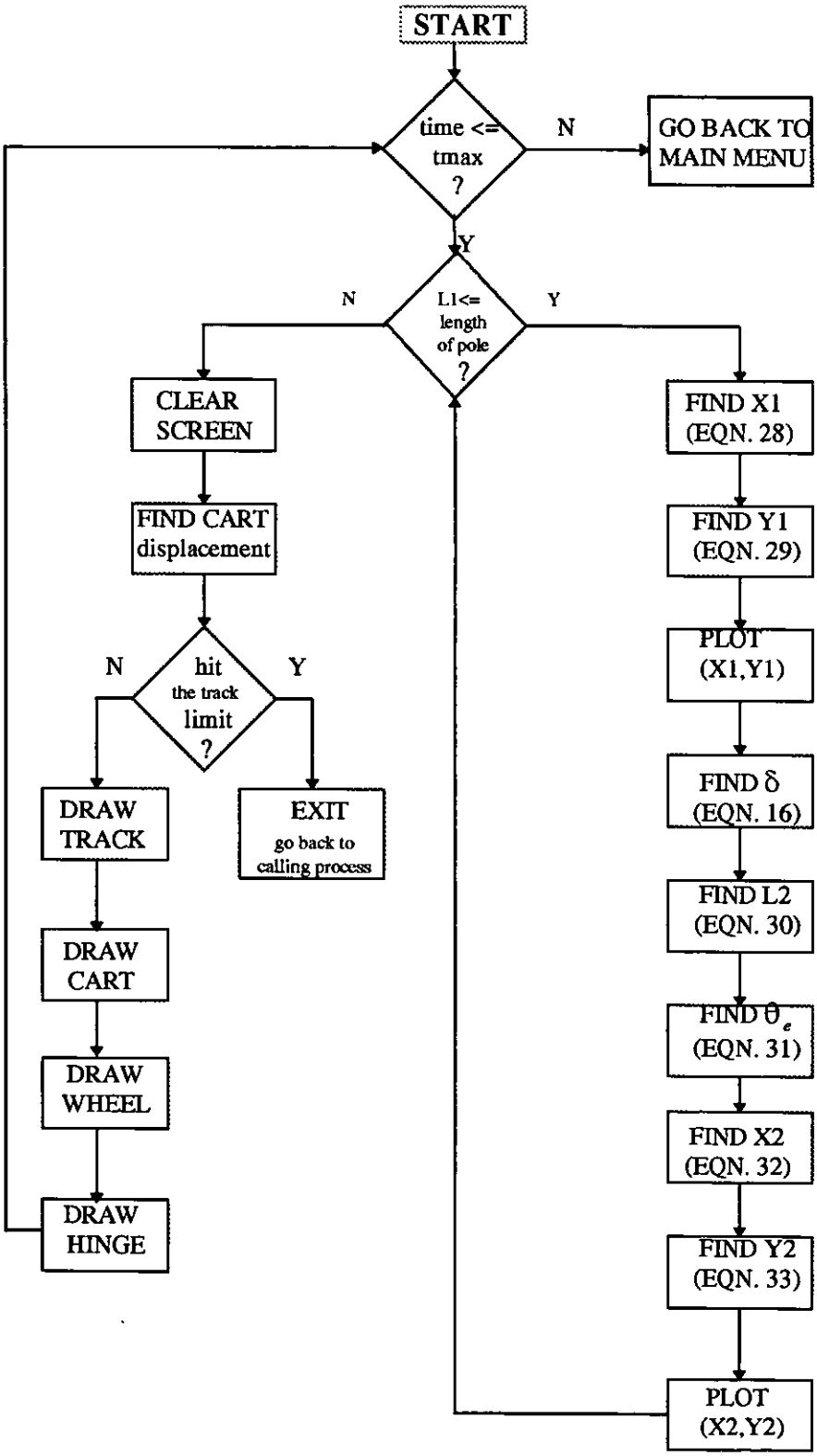


Figure 3.15

Process in drawing the pole at any given time

3.3.3. The Controller

The task of the controller is to balance the flexible pole on top of the cart moving along a limited track for a given time. The algorithm for this process is shown in figure 3.11 section 3.3.2. This process uses numerical_integration. Subprocess RUNGE( ) will calculate the values of the pole's angle, velocity, and acceleration for every time step set by the user. These values are taken from the parameters V and F of RUNGE( ). For every increment of the time step, the controller will check if it has exceeded the total simulation time. If the total simulation time has been attained process numerical_integration will end otherwise subprocess RUNGE( ) will be executed.

The motion of the cart is dependent on the force applied to its body. Hence, it is necessary to control the magnitude and direction of this applied force. However, this force is directly proportional to the angle of the pole from the vertical axis and the total mass of the cart and pole. The angle of the pole is obtained from equation 3.14 by applying numerical integration using fourth order Runge-Kutta. This is the subprocess RUNGE( ).

To determine the actual magnitude and direction of the force, the controller will check first the magnitude and direction of the pole's angle. This is the subprocess check_pole_angle. If the angle of the pole exceeds the prescribed limit then it will report a failure and go back to the main menu, otherwise the process will continue. If the inclination of the angle of the pole is going left (negative) then the direction of the force applied to the cart is going left (also negative), otherwise it is in the opposite direction. The magnitude of the force is chosen using a simple rule-based system in a manner of a look up table (see Appendix A). For 0.0009 to 0.001 degrees inclination this corresponds to 0.1 Newton of force applied to the cart. Above this value an increment of 0.003 degree angle will correspond to an increase of 0.1 Newton in the applied force. The controller can apply a maximum force of 15 Newton's, although the user may enter an initial force greater than this. The author conducted a number of experiments in simulation the values relating the applied forces on the cart and the angles of the pole, to determine satisfactory

values of the parameter. The control algorithm can be modified to accommodate changes in mass. The above controller is set for a total mass of 1.1 kilograms. Figure 3.16 show the results of the flexible pole cart balancing controller operating with different parameters.

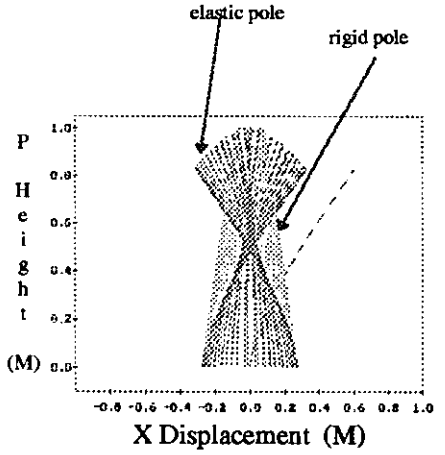
3.3.4. Computer Simulation Results

The figures that follow show the behavior of the pole cart simulation programme under varying conditions:

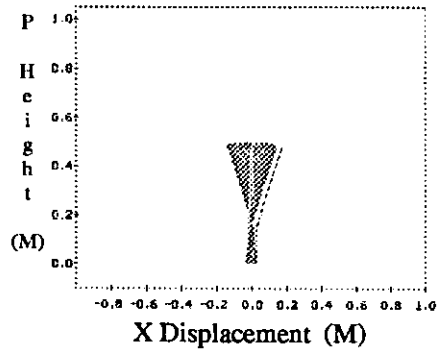
Section 3.3.4.1 to 3.3.4.3 described the behavior of the system for a number of conditions. Figure 3.16 shows the animation of the system when parameters are changed. Examples 1, 2, and 3 present the oscillations of the displacement of the cart, velocity of the cart, acceleration of the cart, positions of the pole, and the graphical representation of the motion of the entire system.

Figure 3.16
Animation of the flexible pole-cart balancing system when parameters are changed

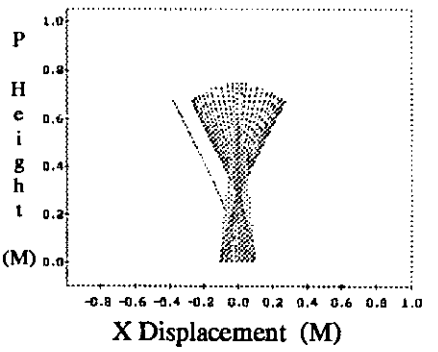
a) Initial angle = +10 degrees
Initial force = 10 Newtons



b) Initial angle = 5 degrees
Initial force = 1 Newton



c) Initial angle = -10 degrees
Initial force = 7 Newtons



3.3.4.1. Example 1:

A computer simulation of the flexible pole-cart balancing system using the following data.

Mass of the pole	= 0.1000 kg
Total mass of the pole and the cart	= 1.1000 kg
Total length of the pole	= 1.0000 meters
Breadth of the pole	= 0.0300 meters
Depth of the pole	= 0.0050 meters
Elasticity of the pole	= 0.1800 Pascal
Initial force applied	= 7.0000 Newton
Coefficient of friction	= 0.0000
Step size (H)	= 0.0010
Upper limit of integration (tmax)	= 10.0000
Freq. intermediate printouts (Ifreq)	= 50
Initial time (t sec)	= 0.0
Initial pole angle (theta in deg)	= 10.000 degrees
Limitations of pole angle	= 50.000 degrees
Initial pole velocity (theta/dt)	= 0.0
Acceleration due to gravity	= 9.81 m/sq sec.

Figures 3.17a to 3.17e show the dynamic behaviour of the flexible pole-cart balancing system for example 1.

Figure 3.17a

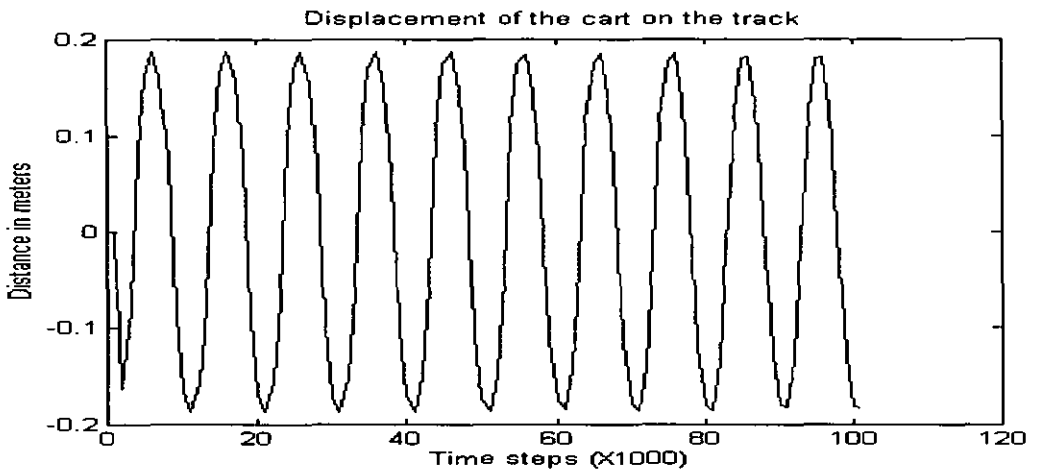


Figure 3.17b

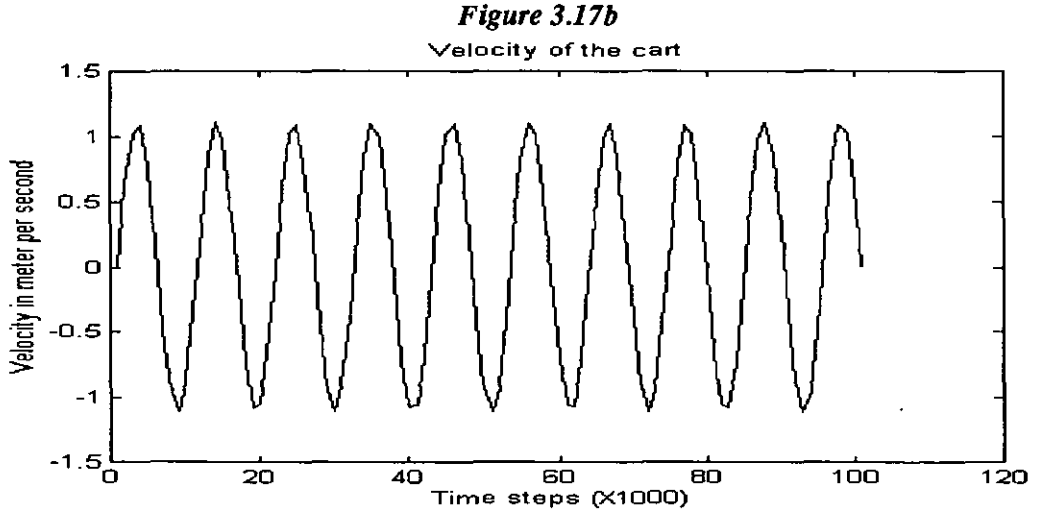


Figure 3.17c

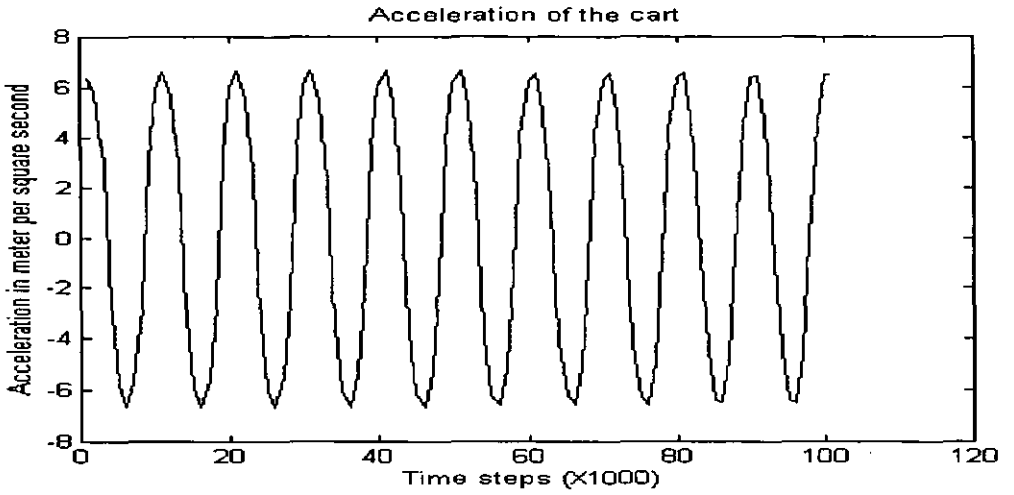


Figure 3.17d

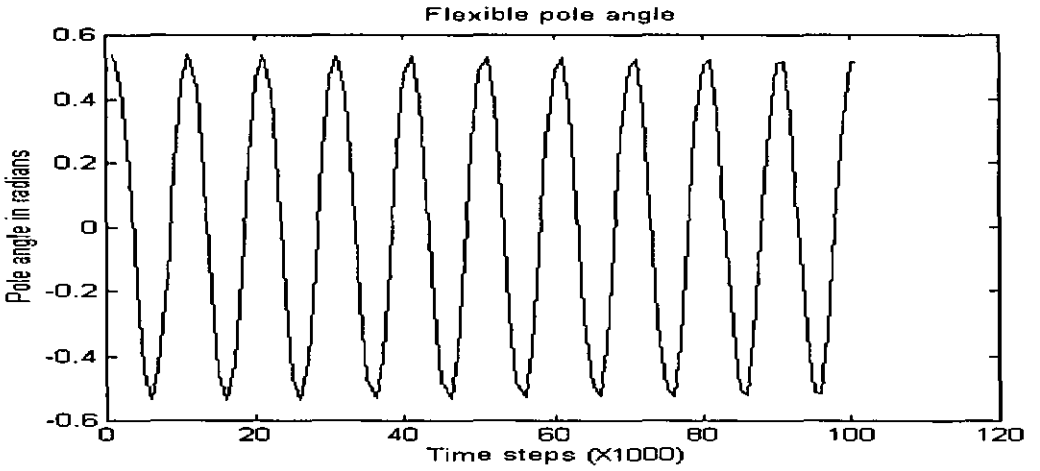
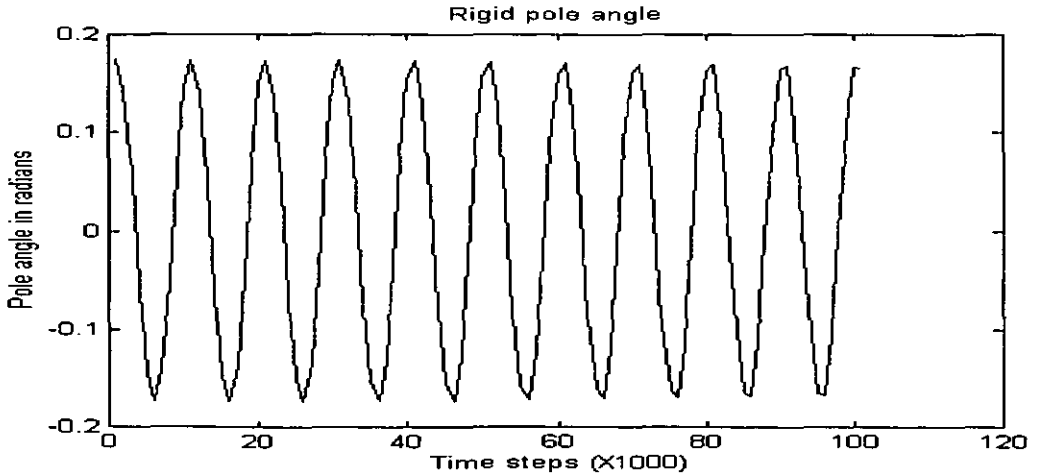
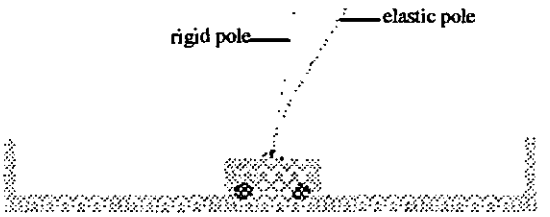


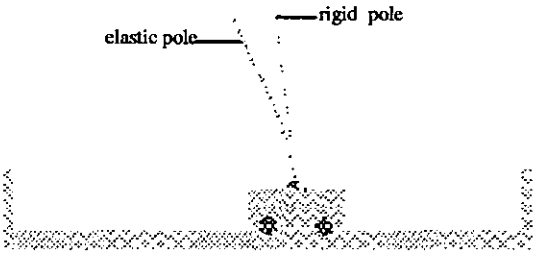
Figure 3.17e



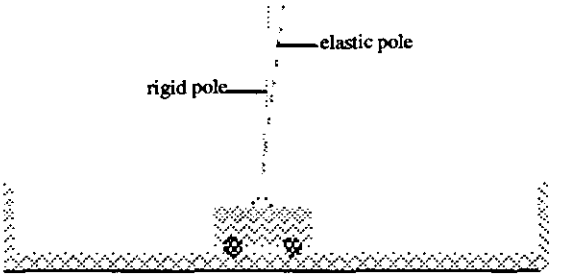
The figures below represent the real time movement of the cart and the whip of the pole for example 1.



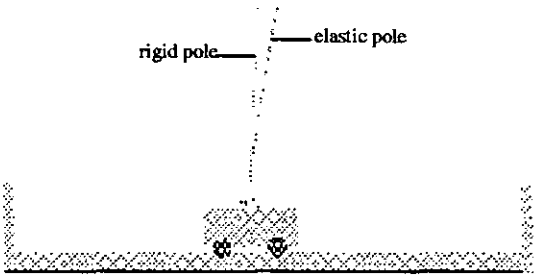
Time = 0.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



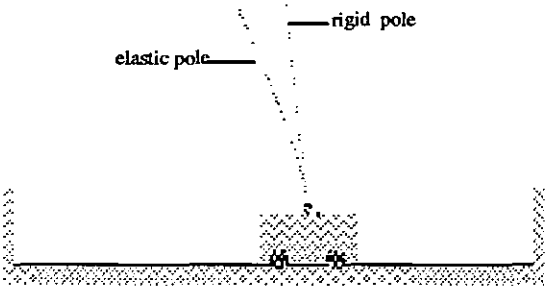
Time = 0.6 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



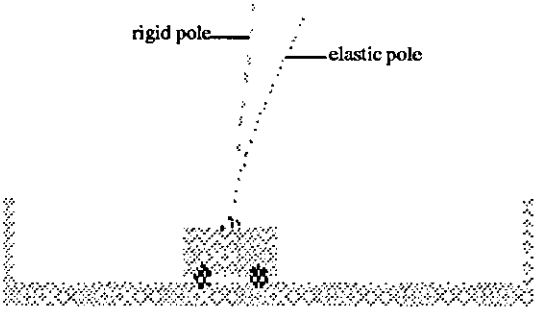
Time = 1.2 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



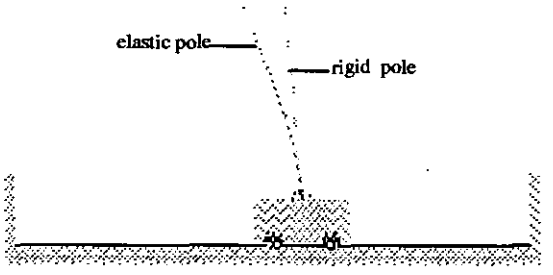
Time = 1.8 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



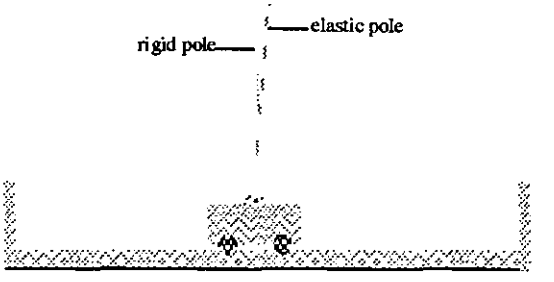
Time = 3.4 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 4.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 4.6 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 5.2 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation

3.3.4.2. Example 2:

In this simulation the values of the size of the cart, the length of the pole, and the initial force were changed. The following data is as below.

Mass of the pole	= 0.0500 kg
Total mass of the pole and the cart	= 0.5050 kg
Total length of the pole	= 0.5000 meters
Breadth of the pole	= 0.0150 meters
Depth of the pole	= 0.0025 meters
Elasticity of the pole	= 0.1800 Pascal
Initial force applied	= 1.0000 Newton
Coefficient of friction	= 0.0800
Step size (H)	= 0.0010
Upper limit of integration (tmax)	= 10.0000
Freq. intermediate printouts (Ifreq)	= 50
Initial time (t sec)	= 0.0
Initial pole angle (theta in deg)	= 5.0000 degrees
Limitations of pole angle	= 50.000 degrees
Initial pole velocity (theta/dt)	= 0.0
Acceleration due to gravity	= 9.81 m/sq sec.

Figures 3.18a to 3.18e show the dynamic behaviour of the flexible pole-cart balancing system for example 2

Figure 3.18a

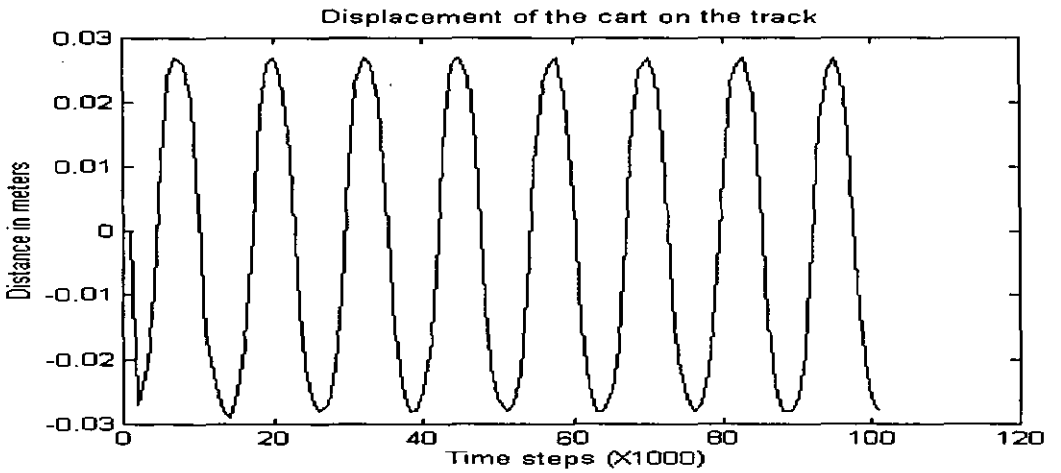


Figure 3.18b

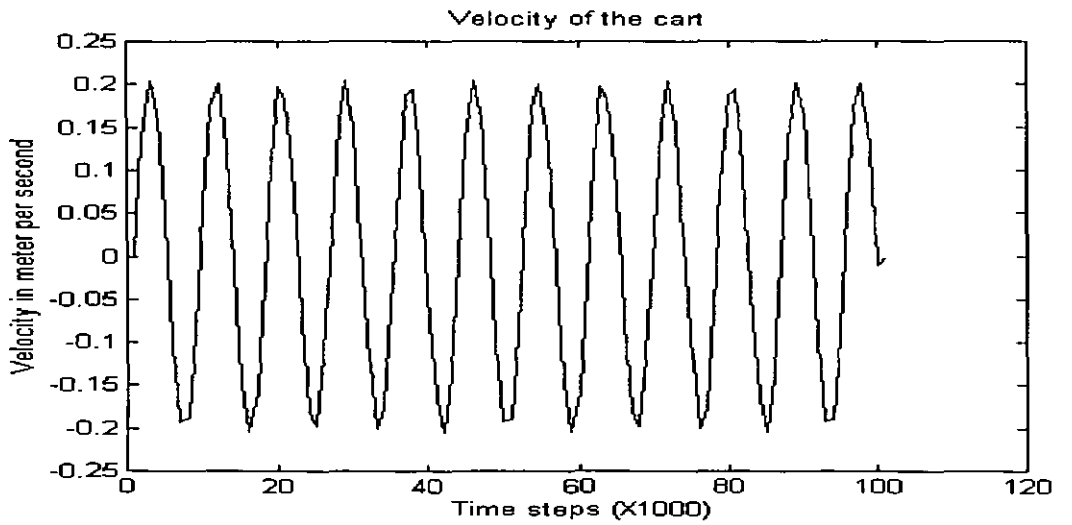


Figure 3.18c

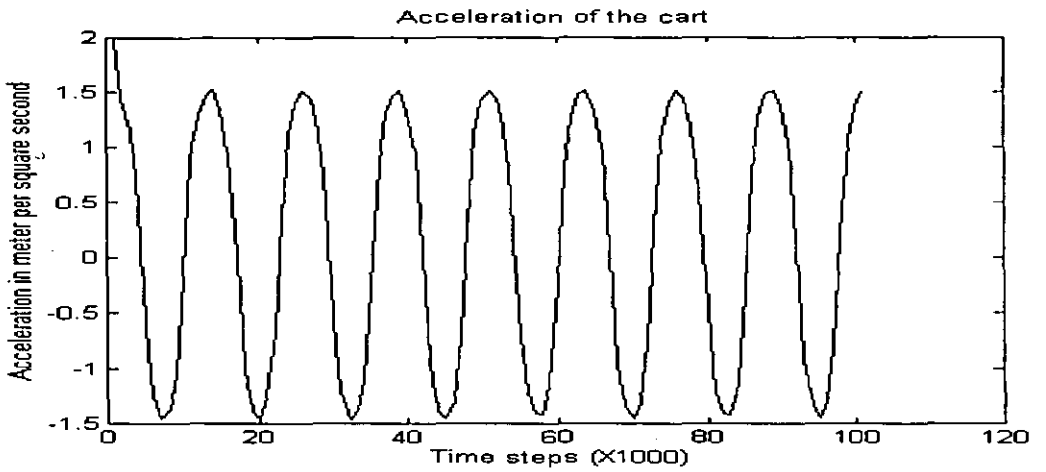


Figure 3.18d

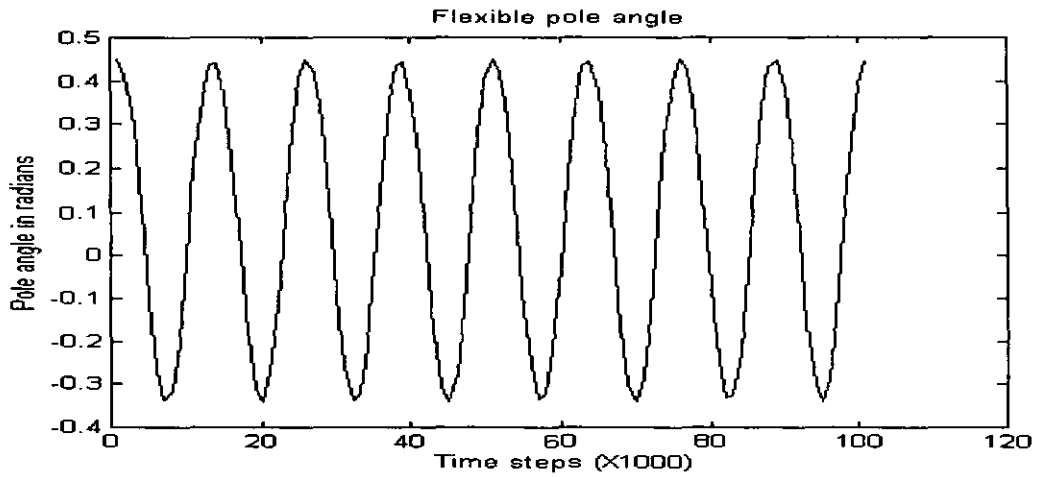
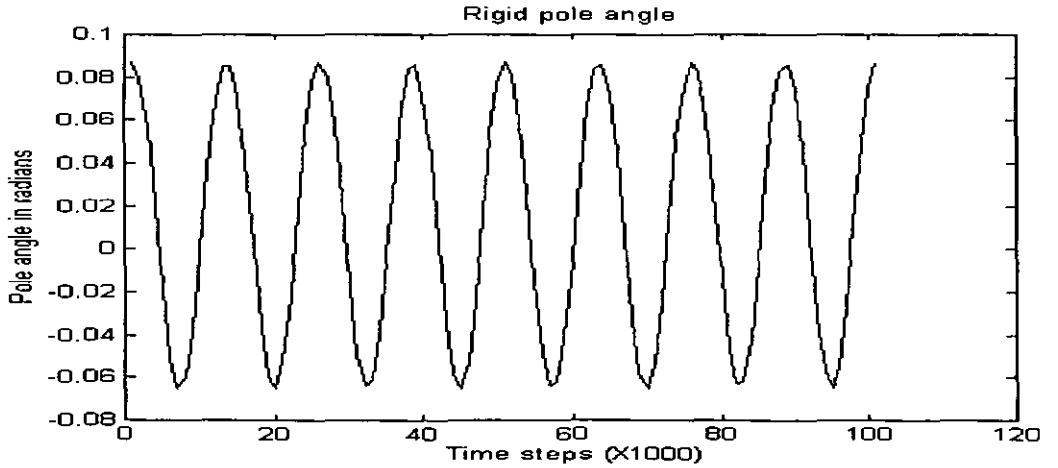
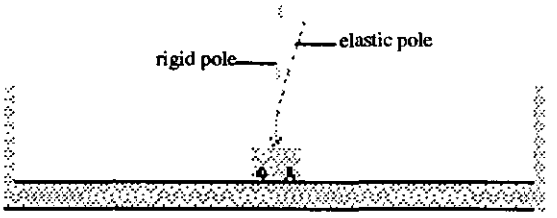


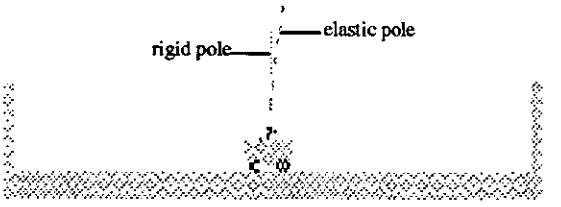
Figure 3.18e



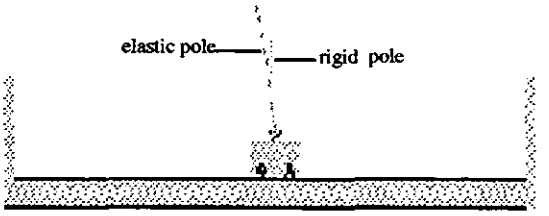
The figures below represent the real time movement of the cart and the whip of the pole for example 2.



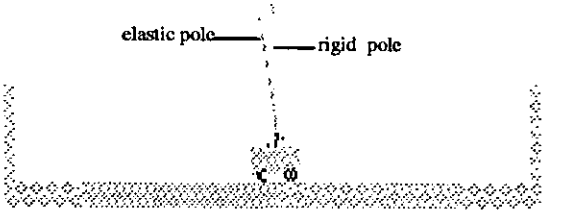
Time = 0.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



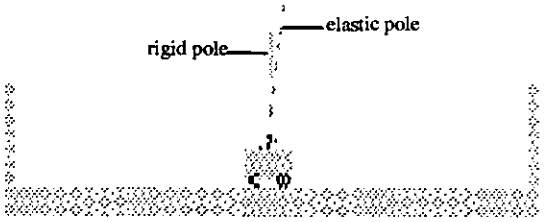
Time = 1.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



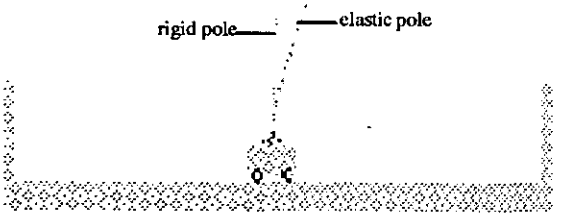
Time = 2.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



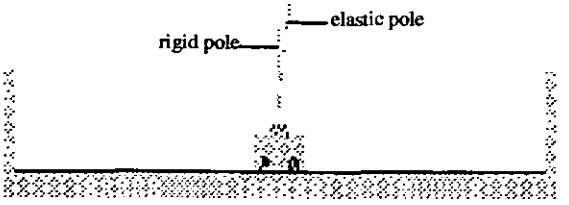
Time = 3.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



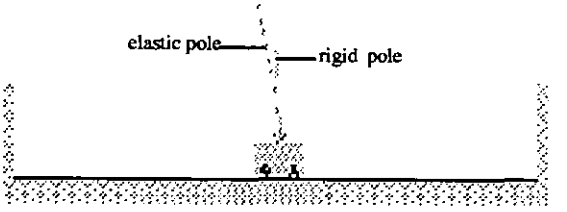
Time = 4.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 5.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 6.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 7.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation

Example 3:

In this simulation the initial values of the pole angle and the force applied to the cart were changed. The data are as below.

Mass of the pole	= 0.1000 kg
Total mass of the pole and the cart	= 1.1000 kg
Total length of the pole	= 1.0000 meters
Breadth of the pole	= 0.0300 meters
Depth of the pole	= 0.0050 meters
Elasticity of the pole	= 0.1800 Pascal
Initial force applied	= 5.0000 Newton
Coefficient of friction	= 0.0000
Step size (H)	= 0.0010
Upper limit of integration (tmax)	= 10.0000
Freq. intermediate printouts (Ifreq)	= 50
Initial time (t sec)	= 0.0
Initial pole angle (theta in deg)	= 5.0000 degrees
Limitations of pole angle	= 50.000 degrees
Initial pole velocity (theta/dt)	= 0.0
Acceleration due to gravity	= 9.81 m/sq sec.

Figures 3.19a to 3.19e show the dynamic behaviour of the flexible pole-cart balancing system for example 3

Figure 3.19a

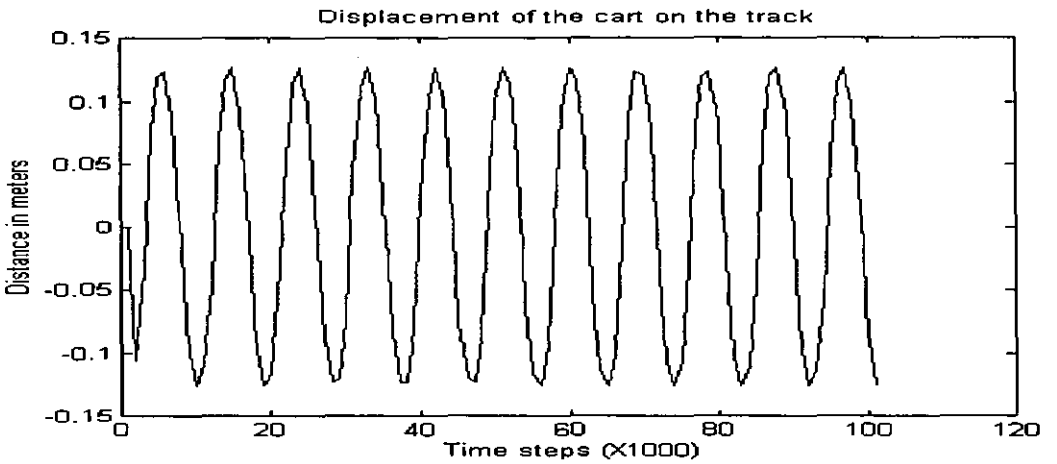


Figure 3.19b

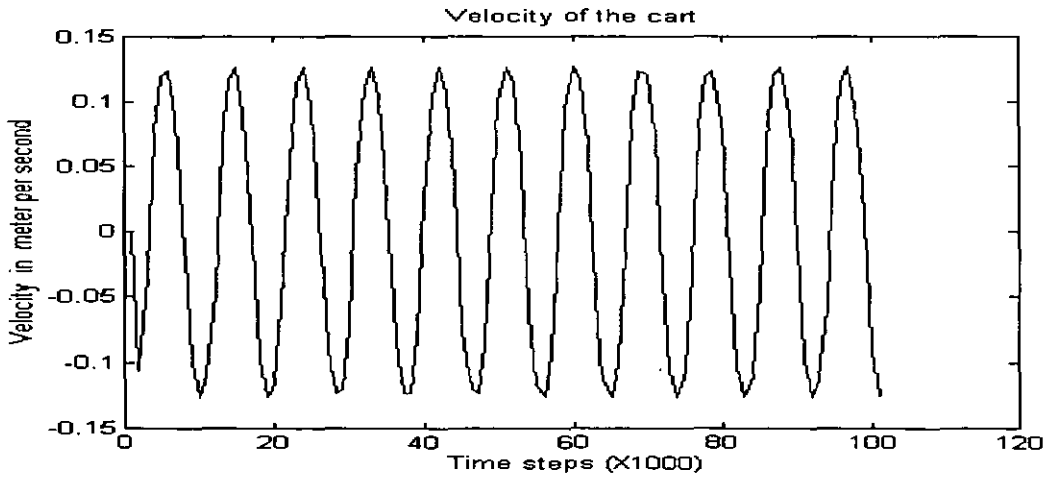


Figure 3.19c

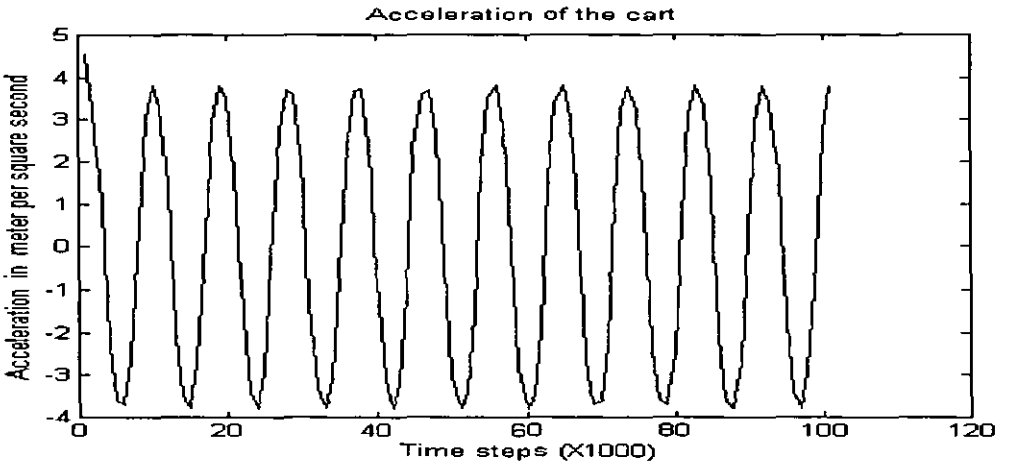


Figure 3.19d

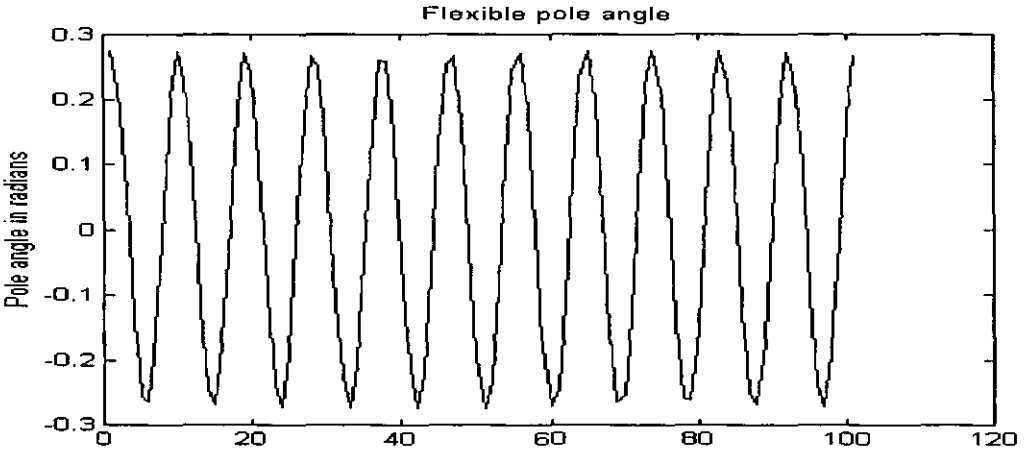
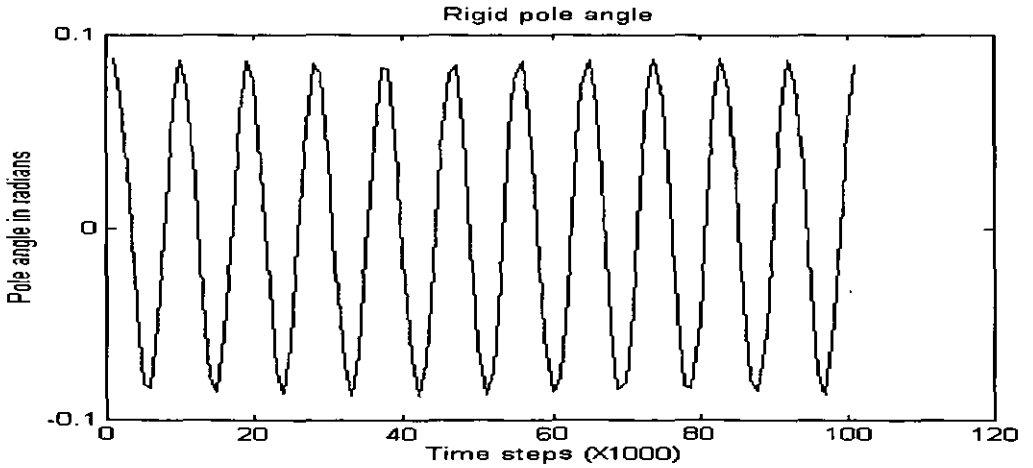
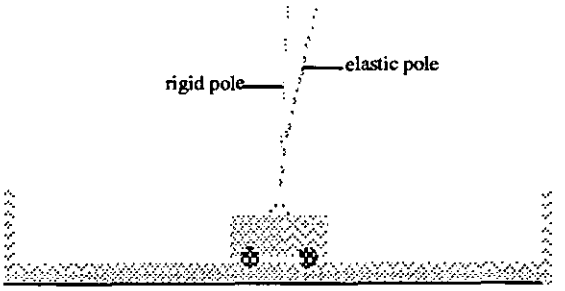


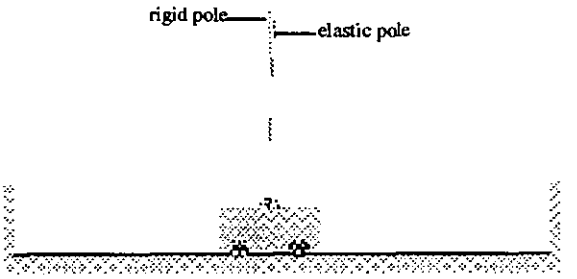
Figure 3.19e



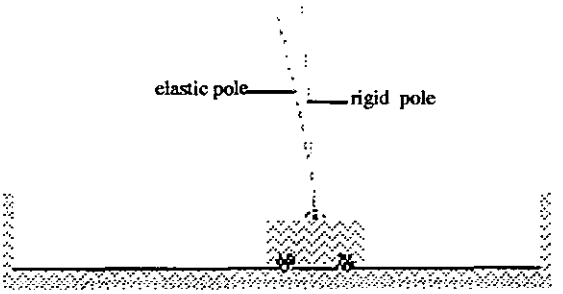
The figures below represent the real time movement of the cart and the whip of the pole for example 3.



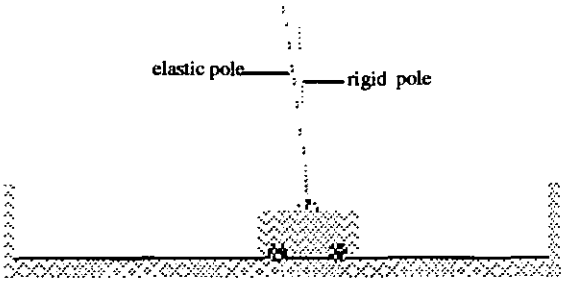
Time = 0.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



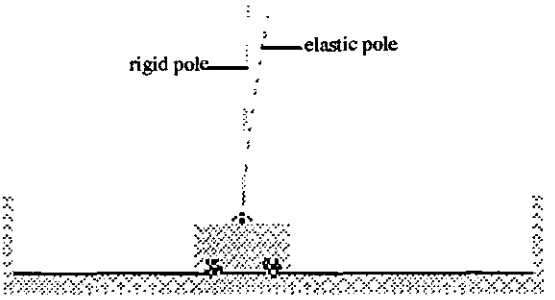
Time = 0.2 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



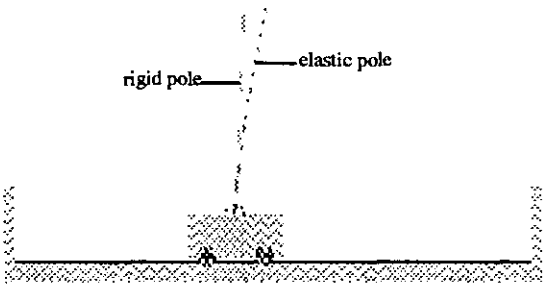
Time = 0.4 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



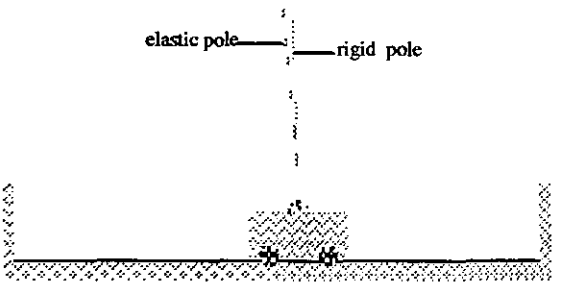
Time = 0.6 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



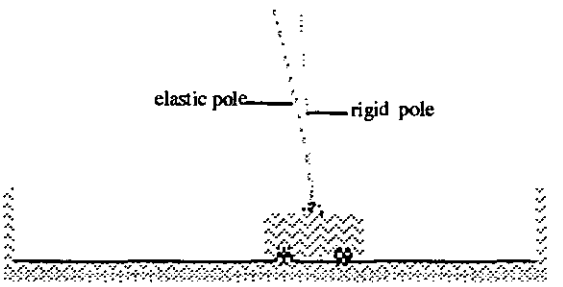
Time = 0.8 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 1.0 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 1.2 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation



Time = 1.4 (X1000 steps)
CART POLE BALANCING SYSTEM
Real time simulation

3.4. Analysis of Results

The program MAYMAY has been used to simulate the mathematical equations derived in Section 3.2. The output of this program indicates that this analysis is qualitatively correct. This needs to be verified by experiment. In order to balance the pole on top of the cart it is necessary to control the force that is applied to the cart. The magnitude of this force is directly proportional to the angle of the pole from the vertical axis and the total weight of the cart and pole. If the angle of the pole increases or the total weight of pole and cart increases then the magnitude of the force also increases. The direction of the force is dependent on the direction of the angle. If the pole is inclined to the left (angle is negative) then the direction of the force is also to the left, otherwise it is the opposite. The value of the force used is selected based upon experiment within the simulation. Figure 3.11 section 3.3.2 shows the algorithm of the controller for this.

The initial condition of the program is, at time t equal to zero, the velocity and displacement of the cart are zero. The force and the pole angle can be initialised to any value (positive or negative). However, since there is a limitation to the length of the track, the value of the pole angle is limited to plus and minus forty degrees. The movement of the cart is dependent on the rate of change of the magnitude of the force and direction. The faster the applied force changes in one direction, the more oscillation there is in of the movement of the cart. Furthermore, the larger magnitude of force applied to the cart the further it travels.

In order to centre the cart on the track, the distance it travels should be controlled. This is derived and explained in section 3.2.2.3. The average frequency of the cart's acceleration is very important here. The frequency should not be small in order to prevent the cart from hitting the track limit. The frequency should not also be too large, in this case the cart will not move sufficiently far (see equation 26).

The strength, efficiency and capability of program MAYMAY has been tested by running it under various conditions to indicate that it appears that the flexible pole cart problem can be controlled.

3.5. Summary

A computer simulation of the simple rule based control of a cart balancing a flexible pole under its first mode of vibration was presented. The appropriate dynamic equations of the system have been derived using Newton's laws, Euler analysis, and Beam theories. The system can be assumed to be with or without friction. Numerical integration using fourth order Runge-Kutta was implemented. A real time graphics representation of the cart balancing the flexible pole on a limited track can be displayed. The behaviour of the system can be analysed and observed by viewing the graphs of the pole's angle versus time (either with or without friction), the acceleration of the cart versus time, the velocity of the cart versus time, and the displacement of the cart versus time. The Turbo Pascal language has been used to implement the computer program on an IBM PC machine.

The simulation program indicated that it was likely to be possible to balance a flexible pole-cart system. It was therefore decided to proceed to demonstrations on a real system without the necessary simplifications made to the model system and to explore the applicability of non-conventional control techniques to the system..

The next chapters of this thesis are therefore focused on the development and testing of on line and off line intelligent controllers using neural network algorithms and fuzzy logic systems on a real system, and the simulation of the application of genetic algorithms as an extension of the non-conventional control approaches.

CHAPTER 4

Off Line Application of Neural Networks to the Flexible Pole-Cart Balancing Problem

4.1. Introduction

Over recent years, neural networks have received a great deal of attention and are being proposed as powerful computational tools [46]. The structures of neural networks are roughly based on our present understanding of the biological nervous system. The potential benefits of neural networks extend beyond the high computation rates provided by massive parallelism. The application phase of neural networks takes relatively little time compared to its training phase and therefore offers potentially faster solutions for problem solving. The basic architecture of a neural network is presented in section 4.2.

This work presents a simulation of the flexible pole-cart balancing problem as a test bed for neural network applications. As has been discussed earlier this type of problem is more complex and highly nonlinear when compared to the classical rigid pole-cart balancing problem because it gives an additional degree of freedom to the classical system, e.g. its transverse displacement. The author has derived (see section 3.2) the mathematical equations of the dynamics of this system and used computer simulation to test the validity of the mathematical model. The results of this computer simulation have been used as the training data for the neural network.

The objective of the work presented in this chapter is to develop and test neural network based software that learns to predict the value of the force applied to the cart at any given time in order to balance the flexible pole hinged at its root on the top of the cart. The Backpropagation neural network architecture and Kohonen's self organizing map have been used to test the capability of neural networks to control the flexible pole-cart balancing problem in simulation. A Backpropagation neural network has been trained by

supervised learning. The network was presented with training data set made up of pairs of patterns i.e, an input pattern paired with a target output. Upon the presentation of this data, weights within the network were adjusted to decrease the difference between the network's output and the target output (see section 4.2.2). The inputs are the elastic pole angle, rigid pole angle, velocity of the cart, and the displacement of the cart, while the output is the force applied to the cart.

A Kohonen's self organizing map neural network is trained by unsupervised learning. It modifies the connection strengths based only on the characteristics of the input pattern presented to the network. It does not require any feedback (see section 4.2.3). In the previous chapter the author used a rule based system to determine the force to be applied to the cart using only the value of the pole's angle. All this data were presented to the neural network which learned to imitate these values using competitive learning.

This chapter begins with the discussion of neural network architectures and it continues to the processes needed for the application of this neural network to the flexible pole cart balancing problem. Results of the experiments conducted using neural network controllers are presented.

4.2. Neural Networks

A neural network is an information processing system that is nonalgorithmic, nondigital, and intensely parallel [44]. It consists of groups of very simple and highly interconnected processors called neurons or processing elements (PE). PE's are analogue of the biological neural cells in the brain. A subgroup of PE's is called a layer in the network. The first layer is the input layer and the last layer is the output layer. The layers that are placed between the input and the output layer are called hidden layers. The PE are connected by a large number of weighted links, over which signals can pass. Each PE typically receives many signals over its incoming connections. These signals may arise from other PE or from the external environment. A PE in a neural network receives input

stimuli along its input connections and translates those stimuli into a single output response, which is then transmitted along the PE's output connections. The mathematical expression that describes the translation of input stimulus pattern to output response signal is called the transfer function of the PE [45].

Figure 4.1 is a typical neural network architecture. The circular nodes represent PE's. There are three layers, an input layer, a hidden layer, and an output layer. The directed graph shows the connections from layer to layer. Although there may be more than one incoming connection, there is never more than one outgoing line from each PE. The outgoing connection often branches to carry the PE's single output signal to many destinations.

Figure 4.2 summarizes how a PE works. Each PE has a number of inputs (*Xi*), each of which must store a connection weight (*Wi*) and compute one and only one output signal (*Yj*). This output is a function (*f*) of the weighted sum $\sum W_i X_i$. The function (*f*) maybe a sigmoid function, sine function, hyperbolic tangent function or various threshold and linear functions. Weights (*Wi*) are variables and can be adjusted dynamically to produce (*Yj*).

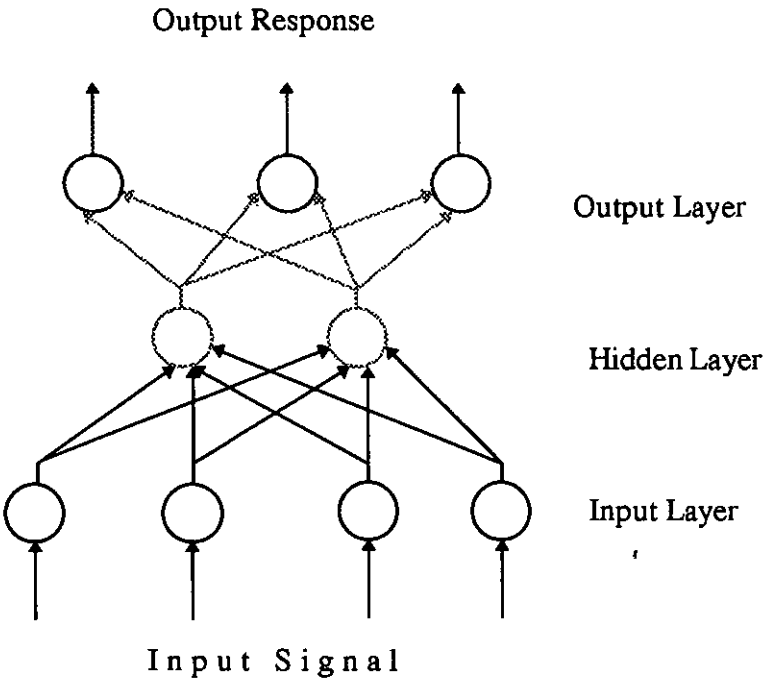


Figure 4.1
A typical neural network architecture

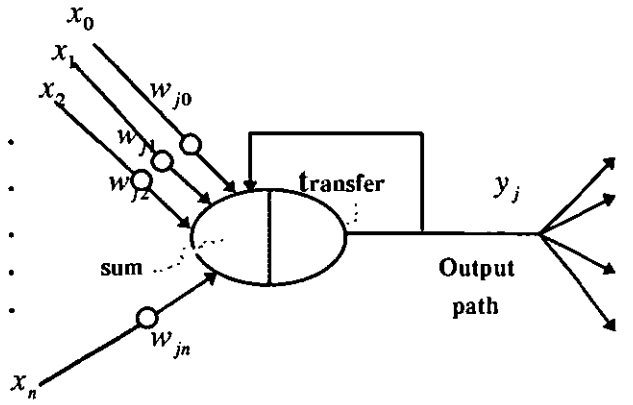


Figure 4.2
The job of a Processing Element

$I = \sum w_{ji} x_j$ Summation ; $y_j = f(I)$ Transfer

x_n = Inputs; w_{jn} = Weights

4.2.1. Training and Learning in Neural Network

Training and learning are fundamental to nearly all neural networks. A network in which learning is employed must be trained. Training is an external process or regimen. It is the procedure by which the network learns. Learning is the result that takes place internal to the network. It is the process by which a neural network modifies its weights in response to external inputs. Weights are changed when the output(s) are not what is expected.

Training is done using examples, and it can take place in three distinct ways [44], namely; supervised, reinforcement, and unsupervised. In supervised training the network is provided with an input stimulus pattern along with the corresponding desired output pattern. The learning law for such a network typically computes an error, that is, how far from the desired output the network's actual output really is. This error is then used to modify the weights on the interconnections between the PE's. Initial weights can be set randomly. Using this technique, a network can do things like make decisions, map associations, "memorize" information, or generalize.

Reinforcement training is similar to supervised training except that the exact desired output is not provided; only a "grade" of how well the network is working. In this type of training the neural network only receives feedback indicating the value of the system's action. The weights are reinforced for properly performed actions and punished for inappropriate ones. This technique is useful in those cases where supervisory information is not available.

Unsupervised training is sometimes called self organization training. In this type of training the network is presented only with a series of input patterns and is given no information or feedback at all about its performance levels. The network uses no external influences to adjust its weights. It looks for regularities or trends in the input signals, and makes adaptations according to the function of the network. Even without being told whether it's right or wrong, the network still must have some information about how to organize itself. Competition between PE's can also form the basis for learning. Training

of competitive clusters can amplify the responses of specific groups to specific stimuli and associate those groups with each other. For example, processing elements could be organized to discriminate between various pattern features, such as vertical edges or left-hand and right-hand edges.

4.2.2. The Backpropagation Neural Network Architecture

The backpropagation neural network is one of the most important historical developments of neurocomputing [47]. It is a powerful mapping network that has been successfully applied to a wide variety of problems ranging from credit application scoring to image compression. It was originally introduced by Paul Werbos in 1974 [48], and extended by David Parker [49], and by David Rumelhart [50] in 1986.

The architecture of the backpropagation neural network is a hierarchical design consisting of fully interconnected layers or rows of processing units (see figure 4.3). Each unit is itself comprised of several individual processing elements. This architecture does not have feedback connections, but errors are backpropagated during training. Errors in the output determine measures of hidden layer output errors, which are used as a basis for adjusting of connection weights between the input and hidden layers. Adjusting the two sets of weights between the pairs of layers and recalculating the outputs is an iterative process that is carried on until the errors fall below a tolerance level. Learning rate parameters scale the adjustments to weights. A momentum parameter can also be used in scaling the adjustments from a previous iteration and adding to the adjustments in the current iteration.

The backpropagation network undergoes supervised training, with a finite number of pattern pairs consisting of an input pattern and a desired output pattern. An input pattern is presented at the input layer. The PE's then pass the pattern digits to the next layer of PE's, the hidden layer. The outputs of the hidden layer PE's are obtained by using perhaps a bias, and a threshold function with the activations determined by the weights

and the inputs. These hidden layer outputs become inputs to the outer PE's, which also process using possibly a bias and a threshold function with their activations to determine the final output from the network. Once training is completed, the weights are set and the network can be used to find outputs for new inputs. The number of PE's in the input layer determines the dimension of the inputs, and the number of PE's in the output layer determines the dimension of the outputs.

4.2.3. The Kohonen's Self Organizing Map Neural Network

The self organizing map neural network was developed by Teuvo Kohonen of Helsinki University of Technology during the period 1979 - 1982 [51, 52]. It is employed only in unsupervised learning network applications, where no expected outputs are presented to a neural network. A network, by its self organizing properties, is able to infer relationships and learn more as more inputs are presented to it. One advantage to this scheme is that the system will change whenever the conditions and inputs vary.

In this technique the processing elements compete for the opportunity of learning. The processing element with the largest output is declared the winner and has the capability of inhibiting its competitors as well as exciting its neighbors. Only the winner is permitted as output, and only the winner plus its neighbors are permitted to adjust their weights. The size of this neighborhood can vary during the training period. Inputs are fed into each of the PE's in the Kohonen layer from the input layer (see figure 4.4). Each PE determines its output according to a weighted sum formula. The weights and the inputs are usually normalized, which means that the magnitude of the weight and input vectors are set from 0.0 to 1.0.

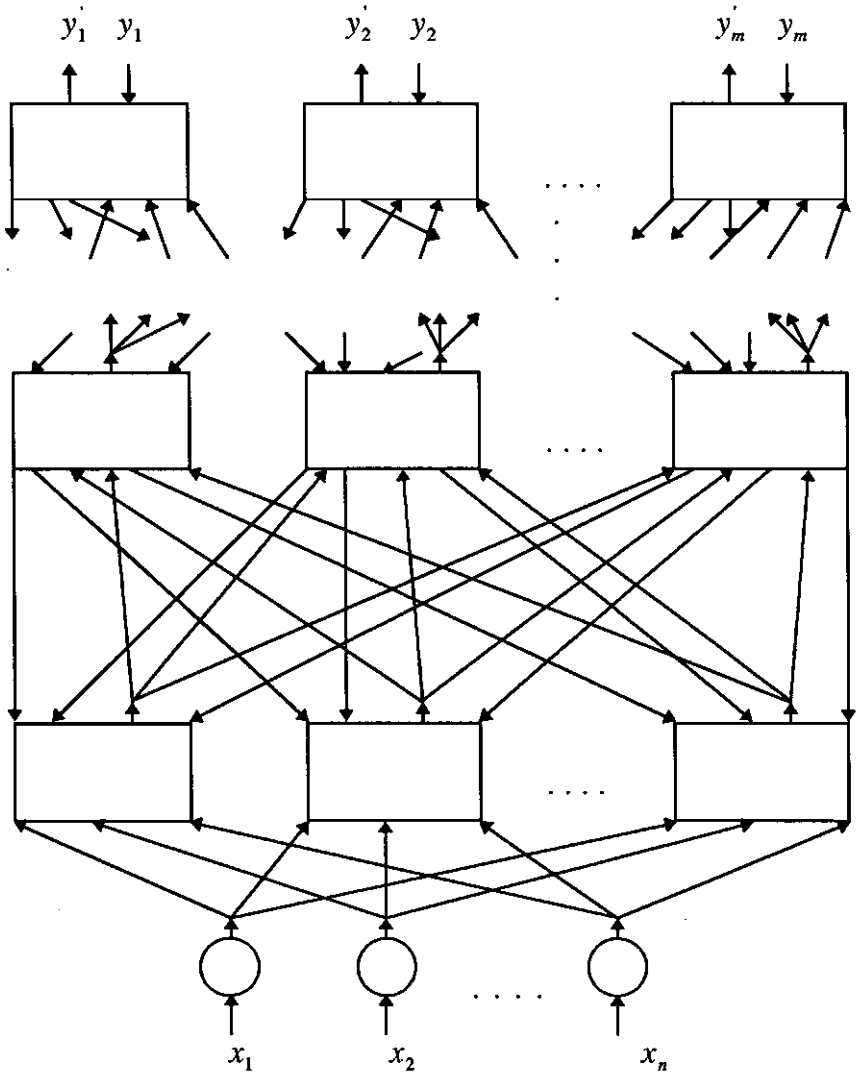


Figure 4.3

Macroscopic architecture of the backpropagation neural network. The boxes and circles are processing elements or neurons.

(x_1 ... x_n are inputs; y_1' ... y_m' are outputs; y_1... y_m are output errors)

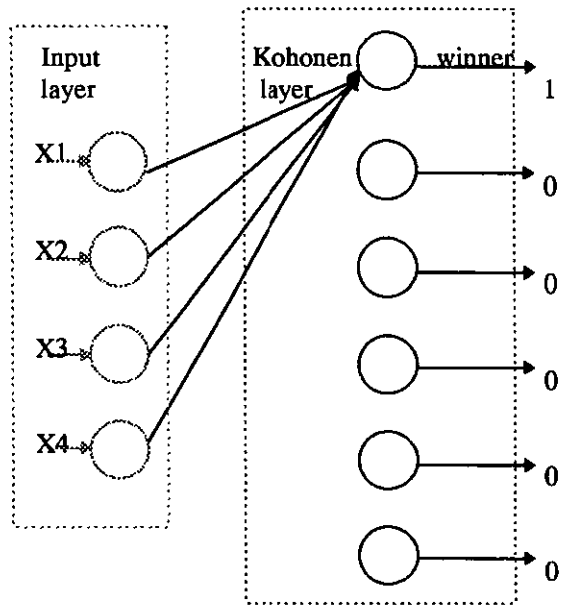


Figure 4.4
A Kohonen Network

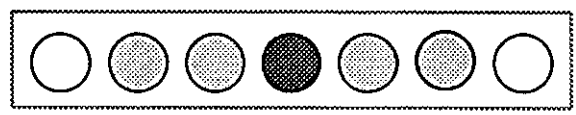


Figure 4.5
Winner PE with a neighborhood size of 2 for a Kohonen map

4.3. The Flexible Pole-Cart Balancing System

The task of the flexible pole-cart balancing system is to balance an elastic pole that is hinged on a movable cart. It is assumed that the hinge is frictionless. The cart is allowed to move along a track with limited length and that has friction. Forces of different magnitude are applied to the cart in either a left or right direction to balance the pole. The dynamics of this system have been shown in Chapter 3.

4.4. Processes Involved In The Formulation Of The Flexible Pole-Cart Balancing Control: Neural Network Perspective

This section describes the various processes involved in formulating the problem from a neural network perspective and provides an effective specification of the application of a neural network to the flexible pole-cart balancing system. These processes are briefly described below.

(1) The decision on how the information for presentation to the neural network should be represented is very important. Since neural networks are pattern matchers, the representation of the data contained in the training sets is critical to a successful neural network solution. Clear understanding of the problem is necessary. Writing a brief narrative description of what the neural network will do is known to support this. For this work, the goal is to develop a neural network that learns to predict the amount of force exerted on the cart to balance the pole given the position of the pole, displacement of the cart, and the velocity of the cart.

(2) It is important to have enough data to yield sufficient training and test sets to train and evaluate the performance of the neural network effectively. The architecture of the network, the training method, and the problem being addressed are dependent on the amount of data required for training a network. In this research, the data used to train

the network are elastic pole angle, rigid pole angle, velocity of the cart, displacement of the cart, and the force applied to the cart.

(3) The data sets in the input training set, as well as the desired output, should be as orthogonal as possible; that is, the variables contained in the data sets should be independent with no correlation.

(4) Generally, the majority of effort in developing a neural network goes into collecting data examples and preprocessing them appropriately. The standard process is to normalize the data. Here the requirement is that the input to each input processing elements should be in the interval between -1.0 to 1.0 and the output to each output processing element should be between 0.0 to 1.0. The following approaches have been adopted for normalizing the raw data to the pole balancing problem before using it in the neural network.

(A) For input values:

(i) el_ang_n = el_ang_r/max_el_ang;

where:

el_ang_n = normalized value of the flexible pole's angle.

el_ang_r = raw value of the flexible pole's angle.

max_el_ang = largest absolute value of the flexible pole's angle.

(ii) ri_ang_n = ri_ang_r/max_ri_ang;

where:

ri_ang_n = normalized value of the rigid pole's angle.

ri_ang_r = raw value of the rigid pole's angle.

max_ri_ang = largest absolute value of the rigid pole's angle.

(iii) cart_vel_n = cart_vel_r/max_car_vel;

where:

cart_vel_n = normalized value of the cart velocity.

cart_vel_r = raw value of cart velocity.

max_cart_vel = largest absolute value of cart velocity.

(iv) cart_dis_n = cart_dis_r/max_car_dis;

where:

cart_dis_n = normalized value of the cart displacement.

cart_dis_r = raw value of cart displacement.

max_cart_dis = largest absolute value of cart displacement.

(B) For output values:

force_n = force_r/max_force;

where:

force_n = normalized value of the force exerted to the cart.

max_force = maximum value of the force exerted to the cart.

= 15 Newton

Since the output range is from 0.0 to 1.0 the author has used two output vectors in order for the network to identify the direction of the force (negative & positive values) as well as its magnitude.

Example :

For a normalized force of -0.5 the corresponding output data are

0.5 and 0.0.

0.0 indicates that the force is going left (-).

For a normalized force of 0.5 the corresponding output data are

0.5 and 1.0

1.0 indicates that the force is going right (+).

(5) Experiments must be carried out to train and test the neural network. The "architecture" is a specification of the neural network topology, with other attributes of the neural network such as the learning rule; activation function; update function; learning and momentum factors. It should be kept in mind that the number of hidden layers and number of nodes in each layer are problem dependent and are empirically selected. It is necessary to vary the parameters used in the neural network such as the learning rate, error tolerance, momentum, etc. in order to get the fastest convergence.

4.5. Discussions: The Neural Network Simulator

This section describes the construction of the neural network software for the flexible pole-cart balancing problem. There are two algorithms used as a representative networks; the Backpropagation, and the Kohonen's Self Organizing Map. The code is presented as Appendix C.

4.5.1. The Backpropagation Model

The architecture of backpropagation neural network has been discussed in section 4.2.2. For the problem of interest the input layer consists of four PE's because there are four input variables to the network. The output layer has two PE's since the neural network needs two outputs in order to identify the direction and magnitude of the force. In this program the best result was obtained by using two hidden layers, each layer consisting of eight PE's (see figure 4.6 for the complete structure). The following equations [53] are used in the program;

O = desired output pattern.

x = output of input layer

γ = momentum parameter.

λ = learning rate parameter for the hidden layer.

μ = learning rate parameter for the output layer.

$y_j = f((\sum_i x_i W_1[i][j]) + \theta_j)$ = output of jth hidden layer PE.

$z_j = f((\sum_i y_i W_2[i][j]) + \tau_j)$ = output of jth output layer PE.

$O_i - z_i$ = ith component of vector output difference.

$e_i = z_i(1-z_i)(O_i - z_i)$ = ith component of output error at the output layer.

$t_i = y_i(1-y_i)(\sum jW_2[i][j]e_j)$ = ith component of output error at the hidden layer.

$\Delta W_2[i][j] = \mu y_i e_j + \gamma \Delta W_2[i][j](t-1)$ = adjustment for weight between ith PE in hidden layer and jth output PE.

$\Delta W_1[i][j] = \lambda x_i t_j + \gamma \Delta W_1[i][j](t-1)$ = adjustment for weight between ith input PE and jth PE in hidden layer.

$\Delta \tau_j = \mu e_j$ = adjustment to the threshold value or bias for the jth output PE.

$\Delta \theta_j = \lambda e_j$ = adjustment to the threshold value or bias for the jth hidden layer PE.

$f(x) = \frac{1}{(1+e^{-x})}$ = thresholding function.

The program needs the following information from the user:

(a) Error tolerance - this is the difference between desired output and networks computed output. If this is attained the program simulation will stop.

(b) Learning parameter - used in scaling the adjustment to weights.

(c) Maximum number of cycles - a cycle is one pass for the whole training data. This will insure the program stops even if the error tolerance is not attained.

(d) The total number of layers.

(e) The total number of processing elements for every layer.

(f) Momentum parameter - used in scaling the adjustments from the previous iteration and adding the adjustments in the current iteration.

(g) Noise - a random number added to each input component of the input vector as it is applied to the network. This will avoid getting stuck to local minima.

The momentum and noise terms are described more fully in section 4.5.2. There are two major processes to be undertaken to construct the backpropagation network. The first one is the training process and the second one is the testing process. All of this processes use external files for data storage. The training process uses files input.dat, weights.dat, and results.dat. File input.dat contains exemplar pairs, or patterns. Each pattern has four input variables and two output variable (see table 4.1). Once the training process reaches the error tolerance or the maximum number of cycles, the program keeps the state of the network, by saving all its weights in file weights.dat. Results of the last pattern are stored in file results.dat. In the testing process the user will enter only the

number of layers of the network and the processing elements for each layer. The program has assumed that the network has already been trained. External files testing.dat, weights.dat and results.dat are used in this process. File testing.dat contains only the input patterns. When this file is presented to the network it then uses the weights from file weights.dat to evaluate the output. The outputs from the network for all input patterns are then generated and stored in the file results.dat.

4.5.2. The Momentum and Noise Terms

Addition of the momentum term to the training law is a simple change that sometimes results in much faster training process. The weight change, in the absence of error, would be a constant multiple of the previous weight change, i.e. the weight change continues in the direction it was heading [53]. The momentum term is an attempt to try to keep the weight change process moving, and thereby not get stuck in a local minima. The training law for backpropagation as implemented in this simulator is:

$$Weight\_change = learning\_rate * input * error\_ouput + momentum\_parameter * previous\_weight\_change$$

The second term in this equation is the momentum term. The momentum term could be implemented either using the weight change for the previous pattern or using the weight change accumulated over the previous cycle. Although both of these implementations are valid, the second is particularly useful, since it adds a term that is significant for all patterns, and hence would contribute to global error reduction.

Another approach to avoid local minima is to introduce some noise in the input during training. A random number is added to each input component of the input vector as it is applied to the network. This is scaled by an overall noise factor, which has a value from 0 to 1. The noise factor is reduced at regular intervals because as the solution is closer and have reached a satisfactory minimum, it is not needed to interfere with

convergence to the minimum. In this simulation, noise factor decreases as the number of cycles increases.

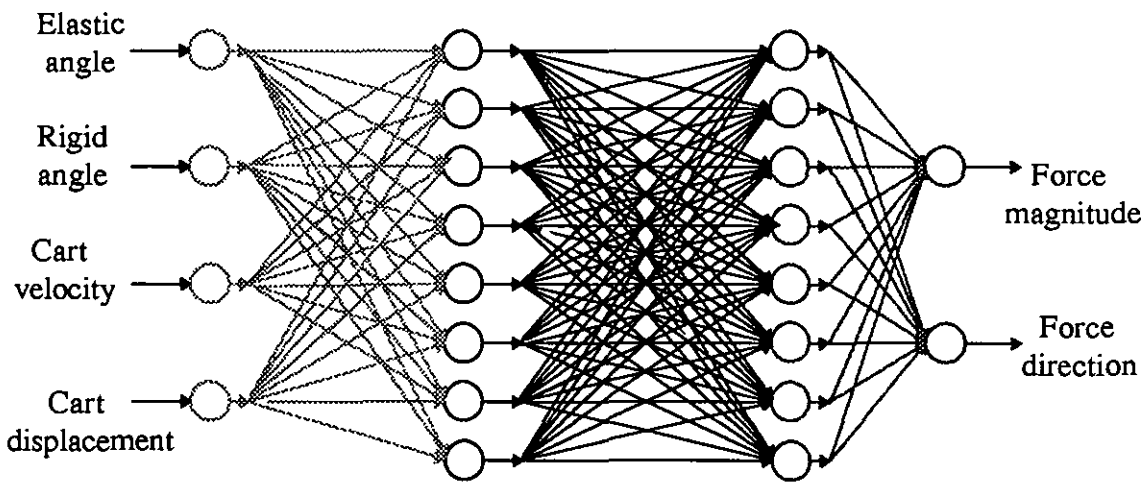


Figure 4.6
Backpropagation neural network model for flexible pole-cart balancing system

4.5.3. Kohonen's Self Organizing Map Model

The Kohonen model is composed of two layers. The input layer and the Kohonen layer (see figure 4.4). Information from the external environment is fed into the input layer. In this program the external information is solely the position of the pole. A rule based system is then used to search for the range of this position (maximum and minimum angle) and the equivalent force applied to the cart for this position. This inputs are fed into each of the processing elements in the Kohonen layer. The Kohonen layer uses a winner-take-all strategy. The processing elements with the highest output is the winner. Each processing element determines its output according to a weighted sum formula [53]:

$$output = \sum W_{ij} x_i .$$

The weights and the inputs in this program are normalized, which means that the magnitude of the weight and input vectors are set equal to one. The reason for this is that the training law uses subtraction of the weight vector from the input vector and normalization reduces both vectors to unit-less status, and hence, makes the subtraction of like quantities possible. Normalization of a vector is obtained by dividing each component by the square root of the sum of squares of all the components.

Example: let a vector $V = k_1 x + k_2 y + k_3 z$; $sq = \sqrt{k_1^2 + k_2^2 + k_3^2}$

then the normalized vector is :

$$V_n = \frac{k_1 x}{sq} + \frac{k_2 y}{sq} + \frac{k_3 z}{sq}$$

The training law for the Kohonen model is straightforward. The change in weight vector for a given output neuron is given by the formula [10]:

$W_{new} = W_{old} + \alpha (I_v - W_{old})$; where α = gain constant between 0 and 1.

I_v = Input vector

The neighborhood size normally has an initial value and it will gradually be decreased as the input pattern cycles continue. The same is true for the gain constant α. This program uses two external files for data storage. The input patterns are stored in file

input.dat and the outputs of the network when the simulation finished are stored in file results.dat. In order to run the program the user must enter the following information.

a) Neighborhood size.

b) Gain constant α .

c) Maximum cycles for the simulation; a cycle is one iteration through the data set.

d) Period; this is the number of cycles after which the α and neighborhood size decrement.

e) The size of the input layer and the Kohonen layer.

4.6. Discussion of Results

The author conducted a number of different sets of experiments in this program. The training data consisted of 40 patterns (see table 4.1). Different methods were used to normalize the data and the best method being the one described in section 4.4. In the application of backpropagation algorithm a number of different layers and processing elements were tried. For a three layer architecture the simulation did not converge. Good results were obtained for four layers. The time of convergence depended on the number of processing elements in each hidden layer. The addition of momentum parameter and noise factor also helped the simulation to converge. In this program the best result was obtained using the following input parameters (see section 4.5.1 for parameter definition).

a) Error tolerance = 0.007039

b) Learning parameter = 0.01

c) Maximum number of cycles = 3050

d) Total number of layers = 4

e) Total number of processing elements for every layer

(input hidden hidden output = 4 8 8 2)

f) Momentum parameter = 0.01

g) Noise factor = 0.05

Samples of the results of the backpropagation simulation are shown in tables 4.1 and 4.2

The program for the Kohonen network is straightforward. Here the number of the processing elements of the Kohonen layer should be greater than the number of the processing elements of the input layer. During some training sessions the winning distance achieved an incorrect value, this was solved by reducing the initial value of the neighborhood size. In this program the best result was obtained using the following input parameters (see section 4.5.3 for parameter definition).

A) Alpha = 0.6

b) Neighborhood size = 10

c) Period = 40

d) Maximum cycle = 30.

Samples of the results of Kohonen program are shown in table 4.3.

Table 4.1: Examples of the results of the backpropagation simulation using 2 outputs.

Samples of the training data for backpropagation model with 2 outputs
(The first 4 columns are the input data and the last 2 are the desired output)
(This is the training data used for examples 1, 2, 3, & 4)

INPUT DATA				OUTPUT DATA	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.847458	1.0
0.647774	0.660336	0.685901	-0.694884	0.661017	1.0
0.402434	0.413959	0.845998	-0.457932	0.423729	1.0
0.121891	0.126050	0.952937	-0.165491	0.152542	1.0
-0.459220	-0.471551	0.368853	0.521115	0.474576	0.0
-0.186079	-0.192290	0.125581	0.230196	0.203390	0.0
0.105567	0.109182	-0.125581	-0.146711	0.135593	1.0
0.387285	0.398548	-0.368852	-0.440091	0.406780	1.0

Example 1

Inputs:

a) Error tolerance = 0.007039
b) Learning parameter = 0.01
c) Maximum number of cycles = 3050
d) Total number of layers = 4
e) Total number of processing elements for every layer
(input hidden hidden output = 4 8 8 2)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 1:

INPUT VECTORS				OUTPUT VALUES	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.841673	0.999996
0.647774	0.660336	0.685901	-0.694884	0.673766	0.999993
0.402434	0.413959	0.845998	-0.457932	0.384845	0.999978
0.121891	0.126050	0.952937	-0.165491	0.162709	0.999701
-0.459220	-0.471551	0.368853	0.521115	0.487629	0.000157
-0.186079	-0.192290	0.125581	0.230196	0.177145	0.002171
0.105567	0.109182	-0.125581	-0.146711	0.143045	0.995773
0.387285	0.398548	-0.368852	-0.440091	0.399558	0.999948

Example 2

Inputs:

a) Error tolerance = 0.0075
b) Learning parameter = 0.01
c) Maximum number of cycles = 2525
d) Total number of layers = 4
e) Total number of processing elements for every layer
(input hidden hidden output = 4 12 12 2)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 2:

INPUT VECTORS				OUTPUT VALUES	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.853668	0.999984
0.647774	0.660336	0.685901	-0.694884	0.693899	0.999962
0.402434	0.413959	0.845998	-0.457932	0.392296	0.999860
0.121891	0.126050	0.952937	-0.165491	0.167283	0.998675
-0.459220	-0.471551	0.368853	0.521115	0.483844	0.000078
-0.186079	-0.192290	0.125581	0.230196	0.176161	0.002150
0.105567	0.109182	-0.125581	-0.146711	0.131581	0.991830
0.387285	0.398548	-0.368852	-0.440091	0.382929	0.999813

Example 3

Inputs:

a) Error tolerance = 0.0075
b) Learning parameter = 0.01
c) Maximum number of cycles = 1360
d) Total number of layers = 4
e) Total number of processing elements for every layer
(input hidden hidden output = 4 16 16 2)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 3:

INPUT VECTORS				OUTPUT VALUES	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.861767	0.999988
0.647774	0.660336	0.685901	-0.694884	0.726439	0.999981
0.402434	0.413959	0.845998	-0.457932	0.411485	0.999946
0.121891	0.126050	0.952937	-0.165491	0.142722	0.999201
-0.459220	-0.471551	0.368853	0.521115	0.474354	0.000141
-0.186079	-0.192290	0.125581	0.230196	0.186945	0.003093
0.105567	0.109182	-0.125581	-0.146711	0.134045	0.990485
0.387285	0.398548	-0.368852	-0.440091	0.390488	0.999867

Example 4

Inputs:

a) Error tolerance = 0.0075
b) Learning parameter = 0.01
c) Maximum number of cycles = 1394
d) Total number of layers = 4
e) Total number of processing elements for every layer
(input hidden hidden output = 4 20 20 2)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 4:

INPUT VECTORS				OUTPUT VALUES	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.847876	0.999994
0.647774	0.660336	0.685901	-0.694884	0.705844	0.999988
0.402434	0.413959	0.845998	-0.457932	0.409977	0.999956
0.121891	0.126050	0.952937	-0.165491	0.153431	0.999167
-0.459220	-0.471551	0.368853	0.521115	0.486288	0.000050
-0.186079	-0.192290	0.125581	0.230196	0.183329	0.002801
0.105567	0.109182	-0.125581	-0.146711	0.116891	0.990446
0.387285	0.398548	-0.368852	-0.440091	0.389119	0.999927

Example 5

Inputs:

a) Error tolerance = 0.0075
b) Learning parameter = 0.01
c) Maximum number of cycles = 5000
d) Total number of layers = 3
e) Total number of processing elements for every layer
(input hidden output = 4 16 2)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 5:

(The simulation experience local minima at: error = 0.045711, max. cycles = 5000)

INPUT VECTORS				OUTPUT VALUES	
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude	Force direction
0.838203	0.846334	0.482706	-0.866357	0.709629	1.000000
0.647774	0.660336	0.685901	-0.694884	0.678095	1.000000
0.402434	0.413959	0.845998	-0.457932	0.571376	1.000000
0.121891	0.126050	0.952937	-0.165491	0.433940	0.999979
-0.459220	-0.471551	0.368853	0.521115	0.632237	0.000001
-0.186079	-0.192290	0.125581	0.230196	0.630291	0.000390
0.105567	0.109182	-0.125581	-0.146711	0.605158	0.998291
0.387285	0.398548	-0.368852	-0.440091	0.610883	0.999999

Table 4.2: Example of the results of the backpropagation simulation using 1 output.

Samples of the training data for backpropagation model with one output
(The first 4 columns are the input data and the last 1 is the desired output)
(This is the training data used in example 6)

INPUT DATA				OUTPUT DATA
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude and direction
0.838203	0.846334	0.482706	-0.866357	0.847458
0.647774	0.660336	0.685901	-0.694884	0.661017
0.402434	0.413959	0.845998	-0.457932	0.423729
0.121891	0.126050	0.952937	-0.165491	0.152542
-0.459220	-0.471551	0.368853	0.521115	-0.474576
-0.186079	-0.192290	0.125581	0.230196	-0.203390
0.105567	0.109182	-0.125581	-0.146711	0.135593
0.387285	0.398548	-0.368852	-0.440091	0.406780

Example 6

Inputs:

a) Error tolerance = 0.0075
b) Learning parameter = 0.01
c) Maximum number of cycles = 4000
d) Total number of layers = 4
e) Total number of processing elements for every layer
(input hidden hidden output = 4 16 16 1)
f) Momentum parameter = 0.01
g) Noise factor = 0.05

Outputs of example 6:

(The simulation experience local minima at: error = 0.082327, max. Cycles = 4000)
(When maximum cycles reached 4388 the weights blown up and the program stopped)

INPUT VECTORS				OUTPUT VALUES
Rigid pole angle	Elastic pole angle	Cart displacement	Cart velocity	Force magnitude and direction
0.838203	0.846334	0.482706	-0.866357	0.856467
0.647774	0.660336	0.685901	-0.694884	0.744855
0.402434	0.413959	0.845998	-0.457932	0.296580
0.121891	0.126050	0.952937	-0.165491	0.340751
-0.459220	-0.471551	0.368853	0.521115	0.000000
-0.186079	-0.192290	0.125581	0.230196	0.000000
0.105567	0.109182	-0.125581	-0.146711	0.067407
0.387285	0.398548	-0.368852	-0.440091	0.453112

Table 4.3: Examples of the results of the Kohonen's simulation.

	Actual angle	Max. Angle	Min. Angle	Force	Cycles
Input data	0.002000	0.003000	0.001001	0.200000	
Output pattern	0.002000	0.002996	0.001004	0.200011	7
Input data	0.003500	0.006000	0.003001	0.300000	
Output pattern	0.003499	0.005995	0.002995	0.300068	7
Input data	0.355000	0.357000	0.354000	12.000009	
Output pattern	0.355012	0.357029	0.354005	11.997595	1
Input data	0.200000	0.201000	0.198001	6.799996	
Output pattern	0.200040	0.201061	0.198270	6.794923	3

4.7. Summary

This chapter has demonstrated the use of neural networks in the control of a highly nonlinear system. A computer simulation of a neural network controlling a model of a cart balancing a flexible pole under its first mode of vibration has been presented. The backpropagation algorithm and Kohonen's self organizing map had been used as neural network examples. The networks learned from a set of training data taken from the results of a computer simulation of the derived dynamics of the flexible pole-cart balancing system (see chapter 4).

The next chapter of this thesis shows the application of a neural network based controller to a real physical flexible pole-cart balancing system.

CHAPTER 5

On Line Application of Neural Networks to the Flexible Pole-Cart Balancing Problem

5.1. Introduction

This chapter presents an on line neural-net based hybrid controller that controls a cart balancing a flexible pole under its first mode of vibration. The networks learned from a set of training data derived from a real system and were initially tested against a computer simulation of the derived dynamics of the flexible pole-cart balancing system and then applied to the real system. The architecture of the neural network is the same as that described in section 4.5.1 with the force output directly mapped to a voltage required by the actuator in controlling the motion of the cart (see figure 5.4). The controller developed had been tested on the physical system and it not only balances the elastic pole for infinite time but also brings the cart nearly to the centre of the track. The controller can still balance the system even if external disturbances are applied to the plant (i.e., pushing the pole in any direction, elevating and shaking the track on either side, etc.). The system can also be initialised anywhere on the track. The controller's action is sufficiently fast that it can balance the system at an initial angle of -19.8 degrees. This is superior to the performance of even rigid pole controllers such those of [3, 4, 5, 6, 9, 10, 11]. The real physical system was constructed by the Quanzer Consulting Company to the authors specification. It was the first such system to be built. Results of experiments on the system are shown in section 5.4.

This chapter begins with the discussion of the hardware architecture of the real flexible pole-cart balancing system and it follows with the application of the neural network. The results of the physical experiments at different conditions are presented.

5.2. The Physical Architecture of the Flexible Pole-Cart Balancing System

A photograph of the real system and the hardware architecture are shown in figures 5.1 and 5.2. The specifications of the physical system are given below. Appendix B also describes the proprietory control system.

- *Track length* = *91.4 cm*
- *Pole length* = *41.0 cm*
- *Mass of the cart & camera sensor* = *0.755 kg*
- *Additional load on the tip of the pole = 0.35 kg*
- *Period of the elastic pole = 2 seconds*
- *Camera system = coupled at the base of the pole and a light bulb is attached to the tip of the pole.*

In the real physical system a camera system is used to detect the deflection of the pole. This is coupled at the base of the flexible pole and will detect the light coming from the bulb attached to the tip of the pole. The deflection of this light corresponds to the deflection of the pole. A potentiometer is attached to the base of the elastic pole in order to obtain its angular position. To determine the distance travelled by the cart, another potentiometer is attached to the wheel that rolls on the track. The values of these sensors are then fed to the computer via an analog to digital / digital to analog converter (AD/DA converter) see figure 5.2. In order to make the problem more complex an additional load was attached on the tip of the pole equivalent to 0.35 kilograms. This has the effect of increasing the period of the elastic pole to 2 seconds.

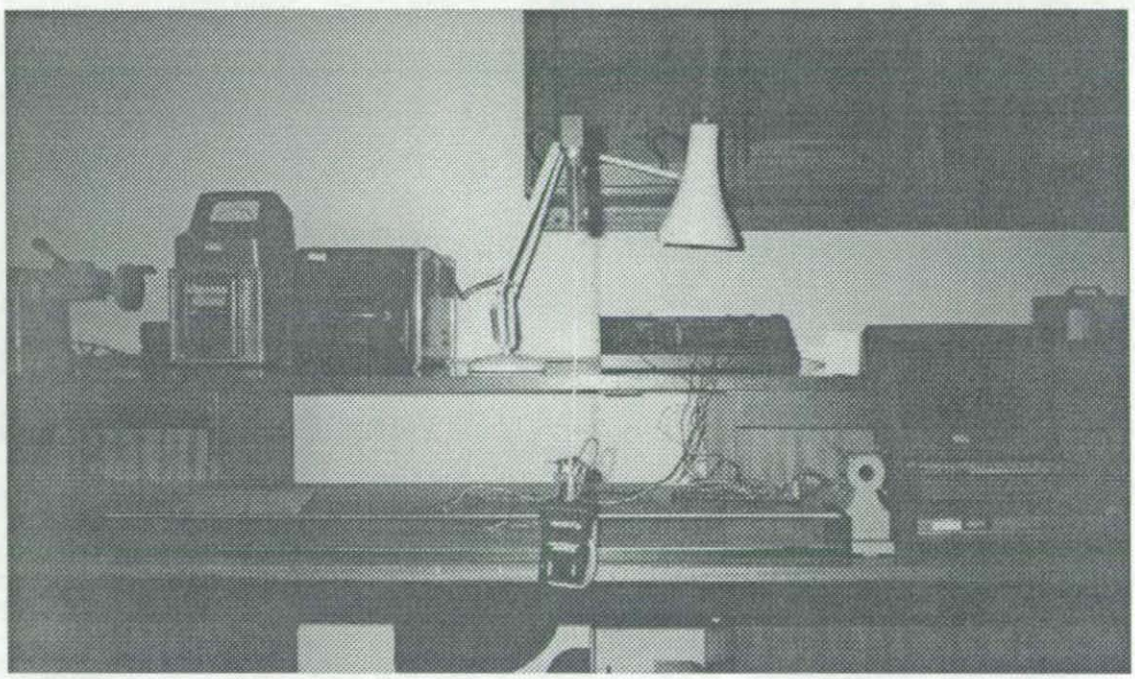


Figure 5.1
A photograph of the real flexible pole-cart balancing system

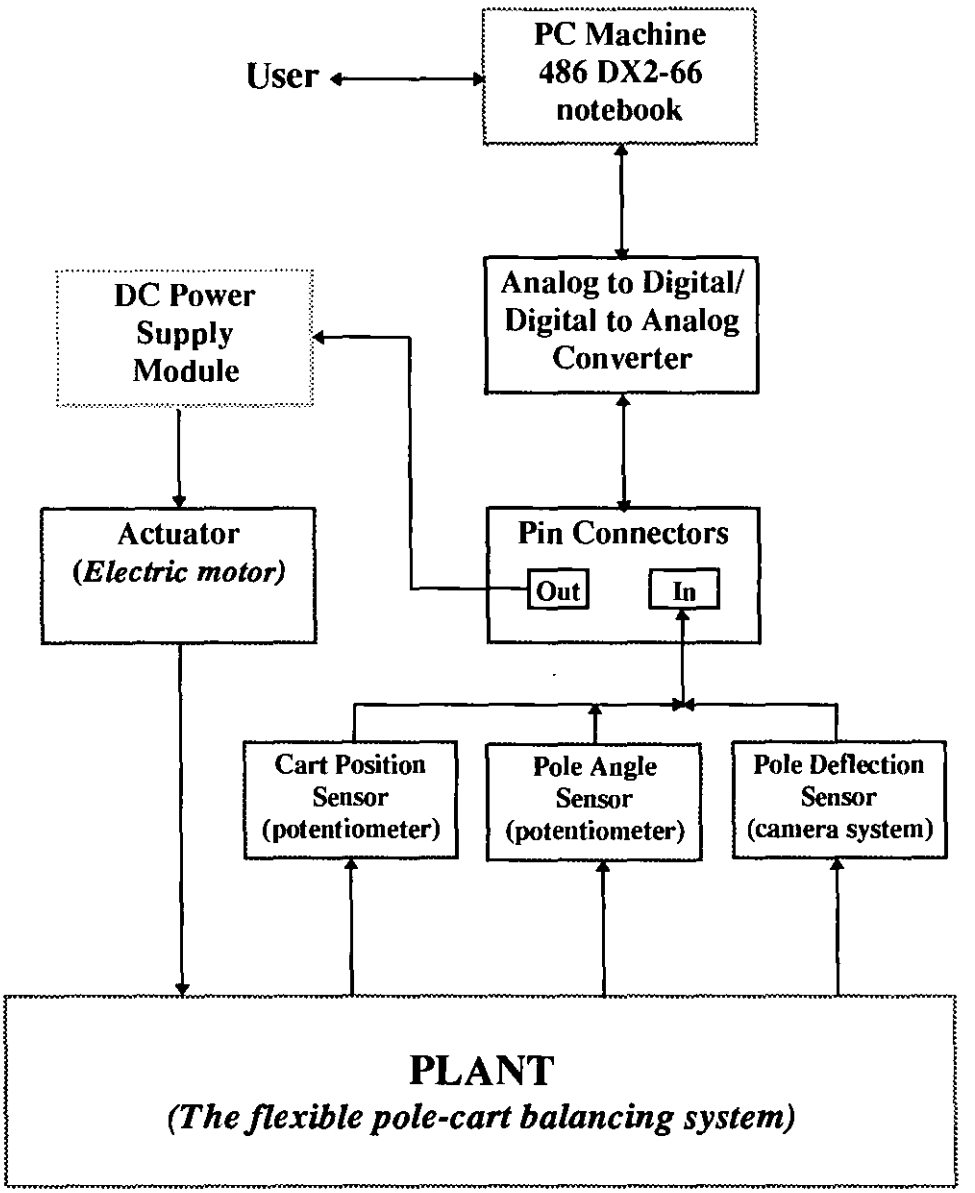


Figure 5.2
Hardware architecture of the real flexible pole-cart balancing system

5.3. Application of Neural Network Model to the Flexible Pole-Cart Balancing System

Figure 5.3 shows the on line hybrid controller block diagram for the flexible pole-cart balancing problem. The backpropagation controller described in section 4.2.2 was applied to the real system with the outputs directly mapped as voltages to the actuator (see figure 5.4). The training data was taken from observations of the inputs and outputs of the real system with its existing controller. Discussions of the training and testing process of this controller was similar to that shown in section 4.5. The backpropagation based controller successfully balanced the pole for a limited period. However, this control system frequently failed due to the cart running out of track. In order to solve this problem, a hybrid control system (see figure 5.3) was then applied to the physical system, the backpropagation system being overridden in extreme cases by a small rule based supervisory system that periodically corrected extreme angles of the pole that caused the cart to decentralise on the track.

It can be seen from figure 5.5 that the feedforward neural network (backpropagation algorithm) used to control the system has 2 hidden layers. Each hidden layer has 8 processing elements (neurons). The input layer has 4 processing elements and the output layer has 2 processing elements. The actual value of the weights connecting each processing element is shown in table 5.1. Here, the leftmost value corresponds to the number of the layer and the number of lines having the same leftmost value corresponds to the number of processing elements for that layer. For example, the first four lines have a leftmost value of 1. The number one corresponds to the first layer, and the four lines correspond to the four processing elements of this layer. The next 8 lines have a leftmost value of 2 indicating the second layer, and the last 8 lines have a leftmost value of 3 for the third layer.

The values next to the leftmost number on each line correspond to the weights that connect a processing element of that layer to all of the processing elements of the next layer. For example, each processing element of the first layer is connected to 8 processing elements of the second layer, hence, there are 8 values next to the leftmost number 1. The

same is true for the second layer, each processing element is connected to 8 processing elements of the third layer, hence, there are 8 values after the leftmost number 2. Finally, the third layer has only two values after number 3 because each processing element is connected only to the two processing elements of the output layer. It is worth noting that the arrangement of the weight values are sequential, meaning, a processing element of a layer is connected to the first processing element, second processing element, third processing element, and so on, of the next layer. The first line of the same leftmost value corresponds to the first processing element of this layer, the second line corresponds to the second processing element of this layer, etc.

Table 5.2 describes the rule base of the evaluator used to correct extreme behaviours of the neural network.

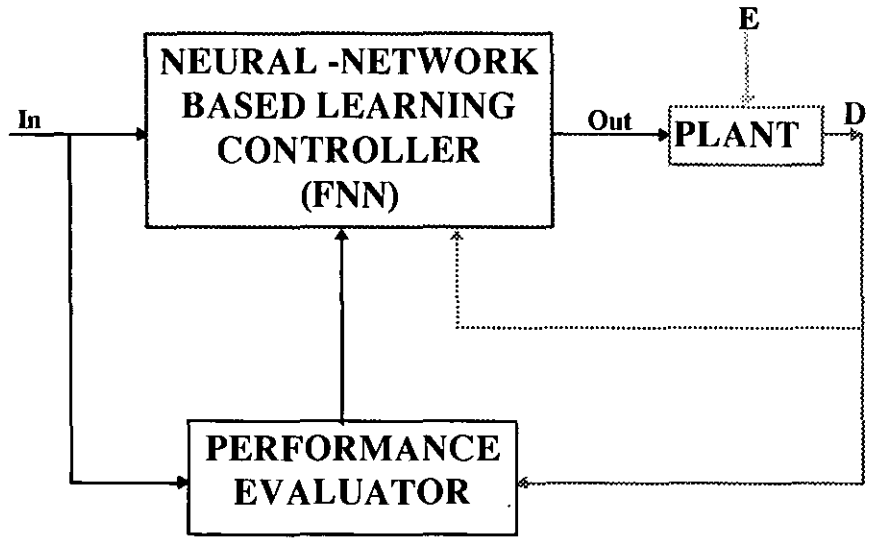


Figure 5.3
Hybrid controller block diagram for the flexible pole-cart balancing problem (on line)

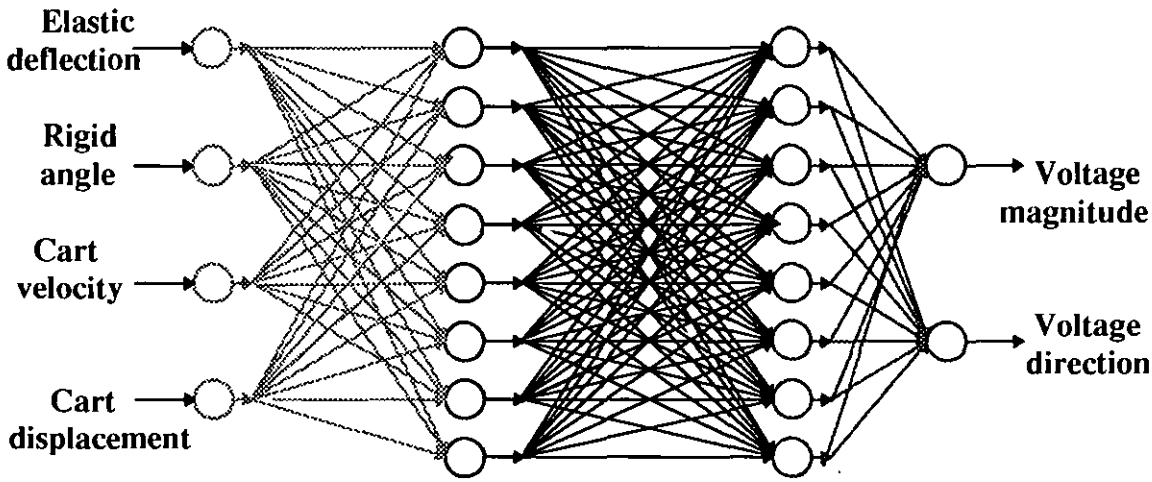


Figure 5.4
Backpropagation neural network model for the flexible pole-cart balancing problem (on line)

Table 5.1

The values of the weights connecting each processing element of the on line feedforward neural network controller

1 -0.188662 -14.036287 -3.352839 -2.473303 1.985948 19.490234 11.239498 0.481953

1 4.956312 -2.398801 -4.827594 -7.715945 0.969971 0.219576 0.509925 -12.291900

1 1.993420 -2.410884 -16.832338 9.828149 0.335660 1.427019 -0.122610 -14.757426

1 7.953069 -4.432342 7.229471 0.493617 1.000431 -0.147754 5.767792 -7.396082

2 -2.260085 2.721589 -0.617301 0.991330 -7.796785 1.129860 5.016824 -3.787388

2 -0.694484 -13.195033 -4.532439 4.279198 1.135099 3.270727 -7.059445 -2.270613

2 -7.269268 -14.795992 -4.052489 5.050180 1.482505 1.696126 -2.924294 2.483483

2 5.767020 1.752683 -0.856498 1.665344 -7.655233 -3.964407 -0.737276 -3.415481

2 0.008674 0.932936 1.678631 0.618250 -3.230417 0.894155 -1.421489 -1.976010

2 0.980527 13.268960 5.889343 -1.131561 0.221078 -7.700460 0.936903 3.213932

2 0.087941 10.657167 4.582888 -0.344960 -2.330751 1.065558 -0.837586 -1.921857

2 2.037073 -14.794135 -5.376902 3.622092 7.990932 1.187436 1.665400 3.076303

3 -0.107427 -8.428615

3 1.923418 26.984537

3 -7.034989 -0.344512

3 0.184818 -5.594577

3 3.655073 10.139201

3 -1.026488 -8.348601

3 8.367657 1.654985

3 -0.067262 6.702623

Table 5.2
The rule based evaluator

Rule 1	*If (pole angle > 2.31 degrees) then applied voltage = 5.0.*
Rule 2	*If (pole angle < -2.31 degrees) then applied voltage = -5.0.*
Rule 3	*If (pole angle > 1.01 degrees) and (pole angular velocity > 0.01 deg/s) then applied voltage = 3.0.*
Rule 4	*If (pole angle < -1.01 degrees) and (pole angular velocity < -0.01 deg/s) then applied voltage = -3.0.*
Rule 5	*If (displacement of the cart > 14 cm.) and (velocity of the cart > 0.01 cm/sec)and (pole angle > 1.01 degrees) then applied voltage = 2.0.*
Rule 6	*If (displacement of the cart > 14 cm.) and (velocity of the cart < -0.01 cm/sec) and (pole angle > 1.01 degrees) then applied voltage = 1.1.*
Rule 7	*If (displacement of the cart < -14 cm.) and (velocity of the cart < -0.01 cm/sec) and (pole angle < -1.01 degrees) then applied voltage = -0.5.*
Rule 8	*If (displacement of the cart < -14 cm.) and (velocity of the cart > 0.01 cm/sec)and (pole angle < -1.01 degrees) then applied voltage = -0.2.*

It can be seen from these rules that rules 1 to 4 take care of balancing the pole under extreme conditions, rules 5 to 8 bring the cart to the centre of the track. Rule 1 is the condition when the pole angle inclines more to the right, while rule 2 inclines more to the left. Rules 3 and 4 is the condition when the pole move fast towards the inclination. Rule 5 is the condition when the cart stays to the right, pole angle inclines to the right, and the cart moves to the right. Rule 6 is the same as rule 5 but the cart moves to the left. Rule 7 is the condition when the cart stays to the left, pole angle inclines to the left, and the cart moves to the left. Rule 8 is the same as rule 7 but the cart moves to the right.

5.4. Results of the Physical Experiments

In this work the author conducted several experiments on the real physical system. The graphs of figures 5.5(i) to 5.15(i) show the actual behaviour of the flexible pole-cart balancing system under different conditions. Each of these graphs depicts the motion and position of the system at any time, the X-coordinate. The Y-coordinate corresponds to the measurements of the angle of the pole in degrees, the deflection of the pole in centimetres, and the location of the cart on the track in centimetres.

The motion of the cart can be analysed by reviewing the graph of the cart displacement. The graph of the pole angle and the pole deflection shows the motion and position of the flexible pole on top of the cart. For example, figure 5.5(i) shows the behaviour of the system when it was initialised at -19.8 degrees. Here, in order to balance the flexible pole, the cart moves quickly to the left direction and after 0.4 seconds the pole angle reached 4 degrees. To bring back the pole angle to the centre the cart then moved back to the right. The system then stabilised after 0.7 seconds and the controller tried to bring the cart to the centre of the track. Also, it can be seen from this figure that the faster the motion of the cart, the larger is the deflection of the pole.

Figures 5.7(i) and 5.8(i) show the behaviour of the system when it was initialised nearly at the end of the track. The controller effectively balances the flexible pole and gradually brings the cart to centre of the track. In figures 5.14(i) and 5.15(i) the controller still balances the system even when the system is initialised at the extreme end of the track with the pole inclined over the end of the track. The controller developed was tested to establish how it react to external disturbances applied to the flexible pole. Figure 5.9(i) shows the behaviour of the system when an external force is applied to the pole. Here, at 2.75 seconds the pole was pushed to the left. Immediately the controller reaction was to move the cart quickly to the left. The controller easily stabilises the system and brings back the cart to the centre of the track.

External disturbances to the track were also applied to the system. Figures 5.10(i) and 5.11(i) show the graphical results of the behaviour of the system when the right and left ends of the track were elevated. Here, the controller balances the pole easily and the cart oscillates around the centre of the track. Figure 5.12(i) shows the graphical result of the behaviour of the system when the track was shaken randomly laterally and figure 5.13(i) shows the behaviour of the system under normal operation. It should be emphasised that for all of the test cases presented the controller developed was able to control the system for infinite time. This represents an improvement on the Quanzer controller, particularly the results shown in figures 5.14(i) and 5.15(i).

Figure 5.5(i)

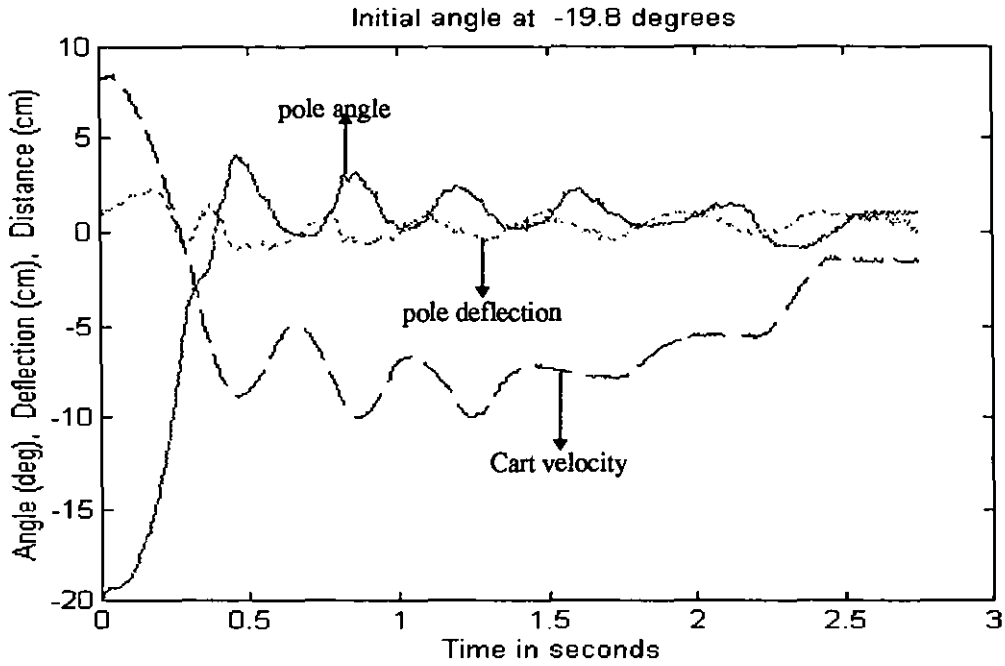


Figure 5.5(ii)

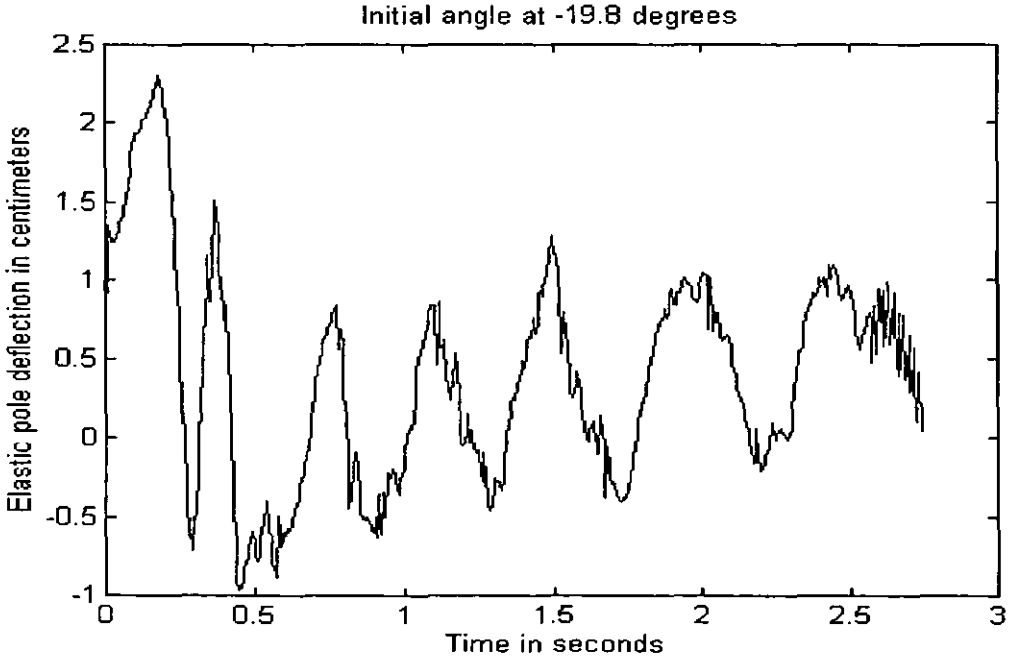


Figure 5.6(i)

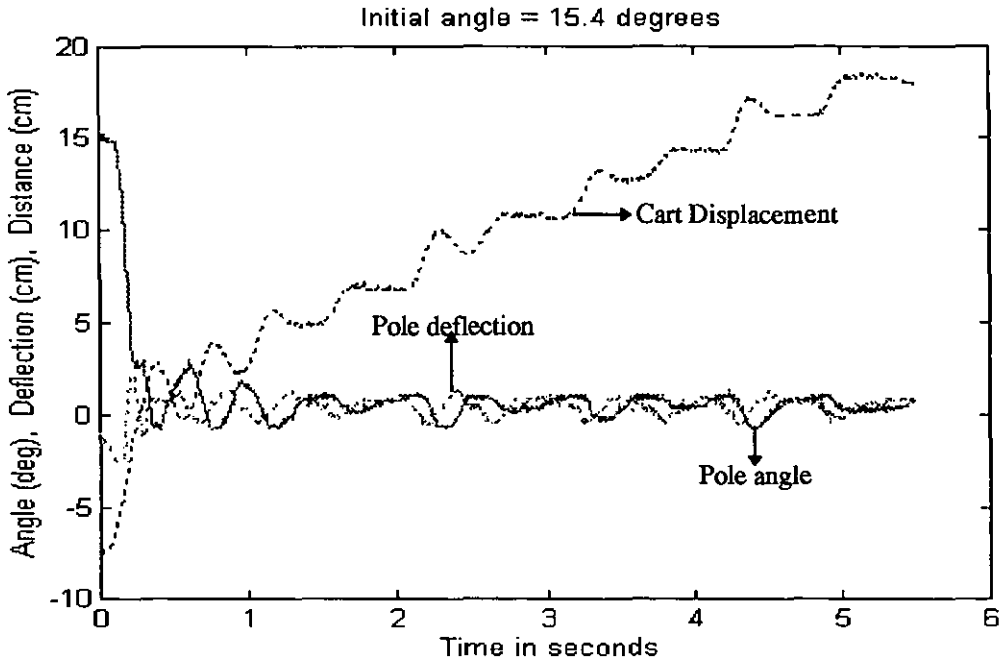


Figure 5.6(ii)

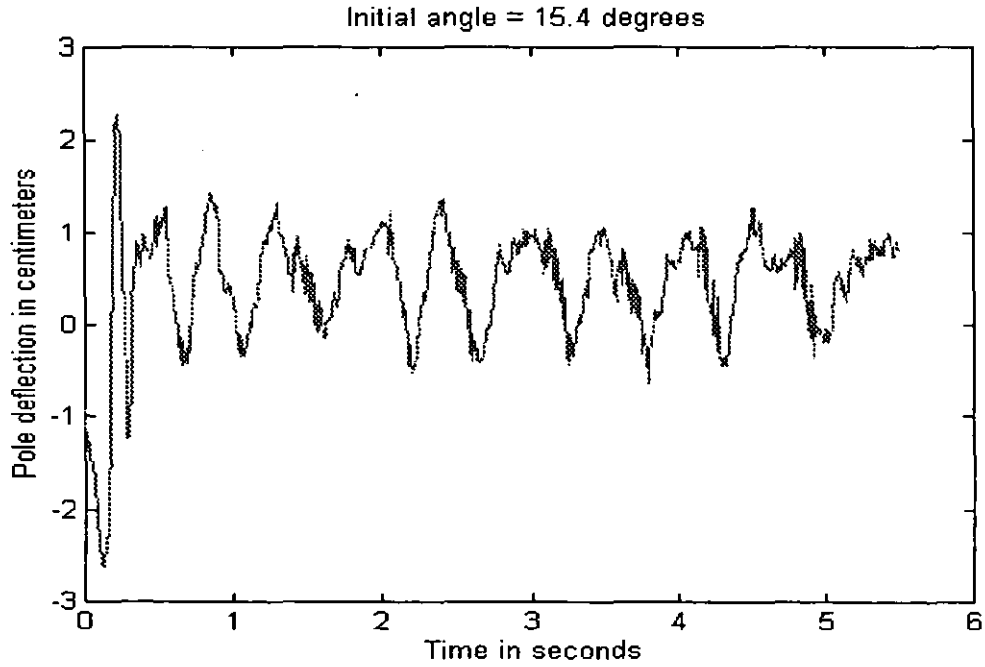


Figure 5.7(i)

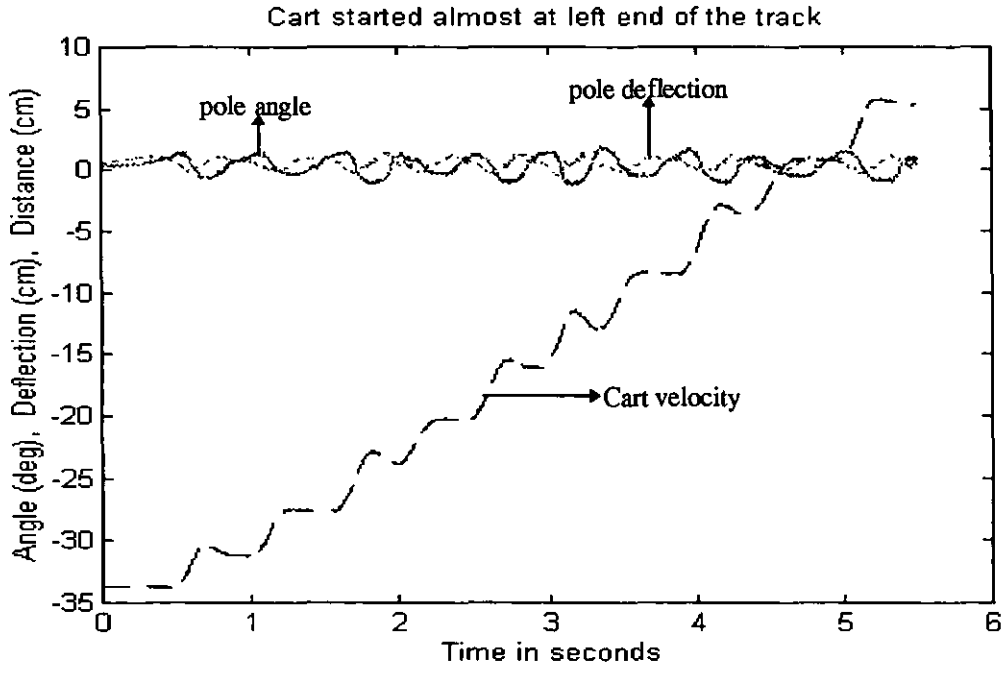


Figure 5.7(ii)

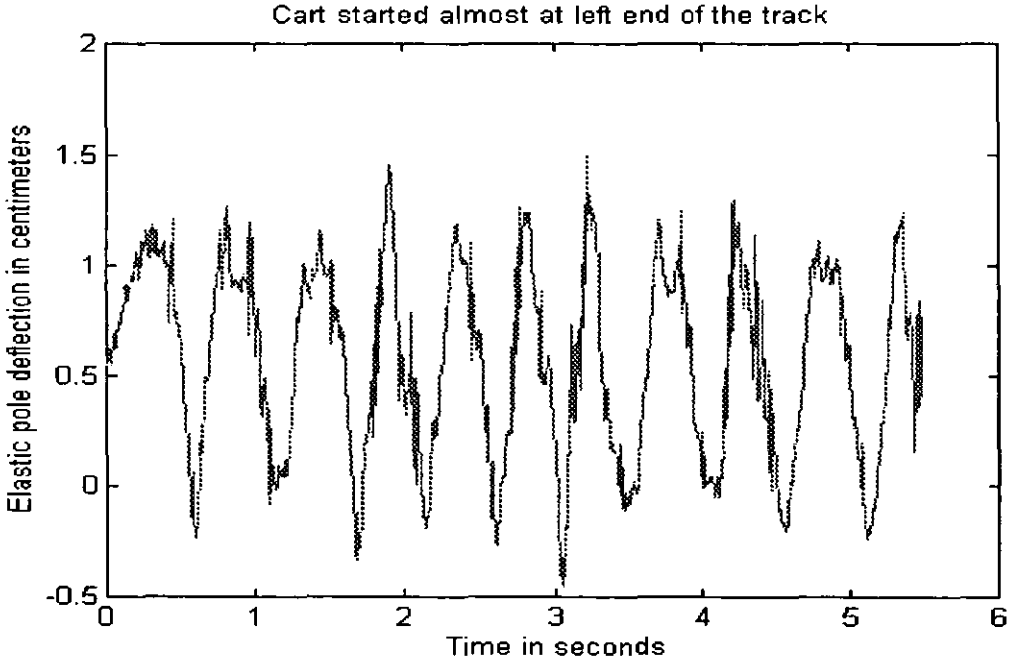


Figure 5.7(iii)

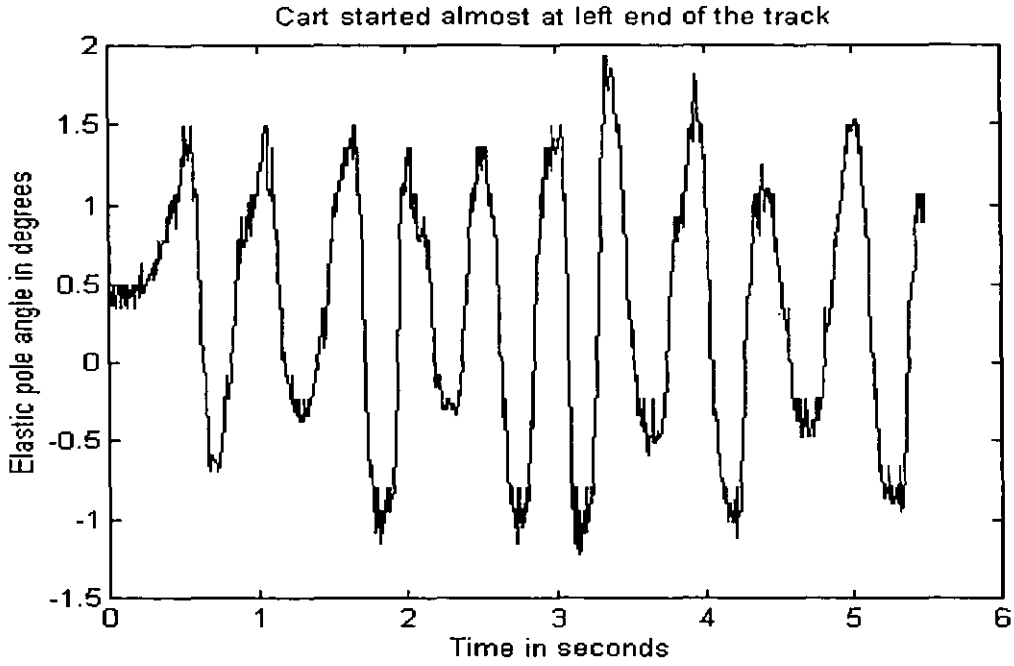


Figure 5.8(i)

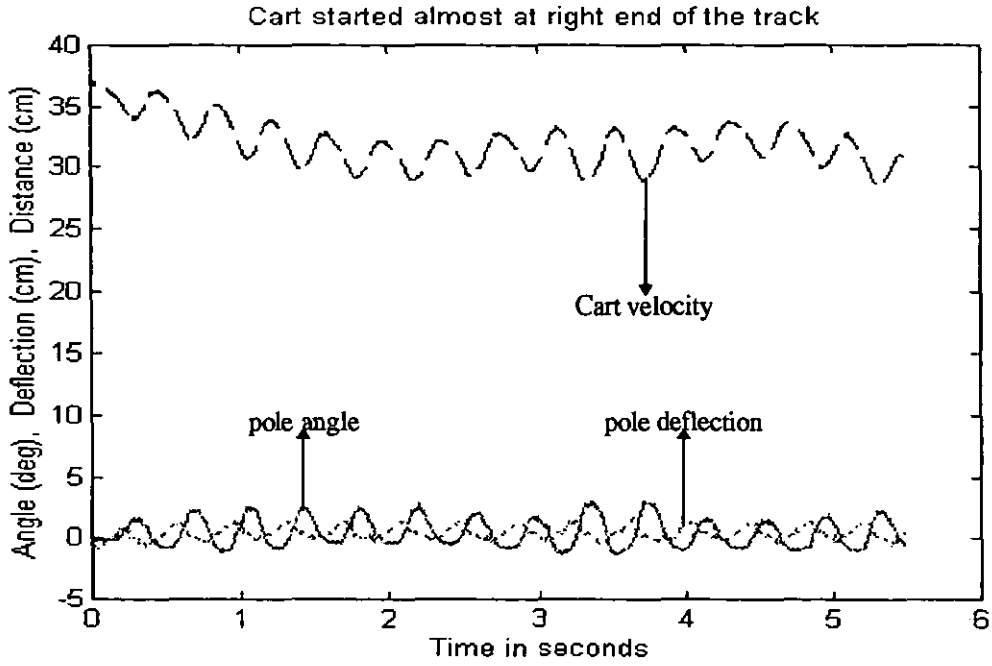


Figure 5.8(ii)

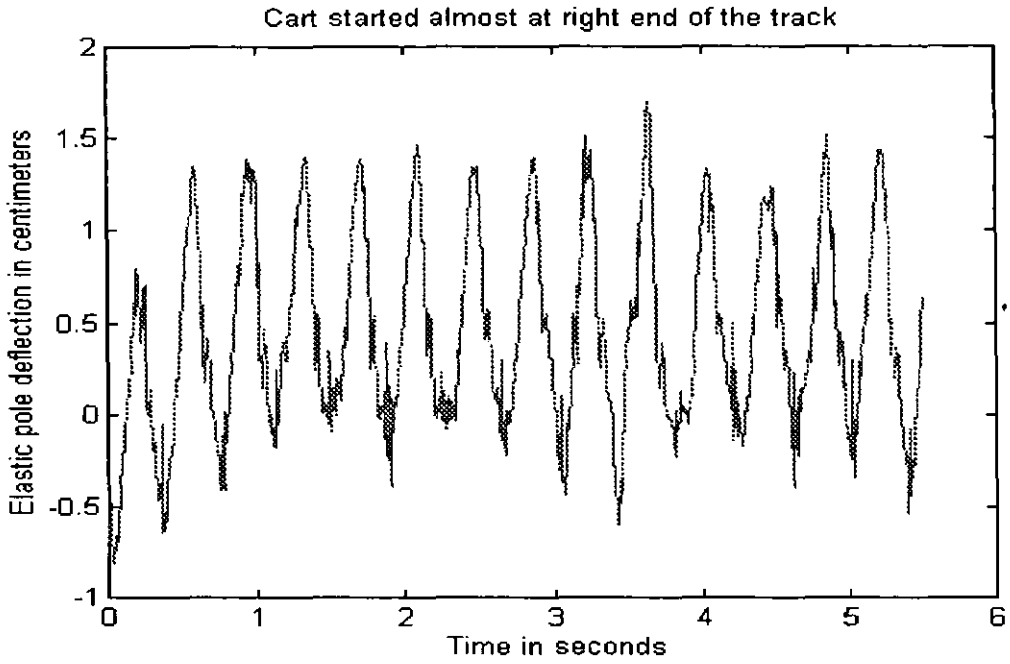


Figure 5.8(iii)

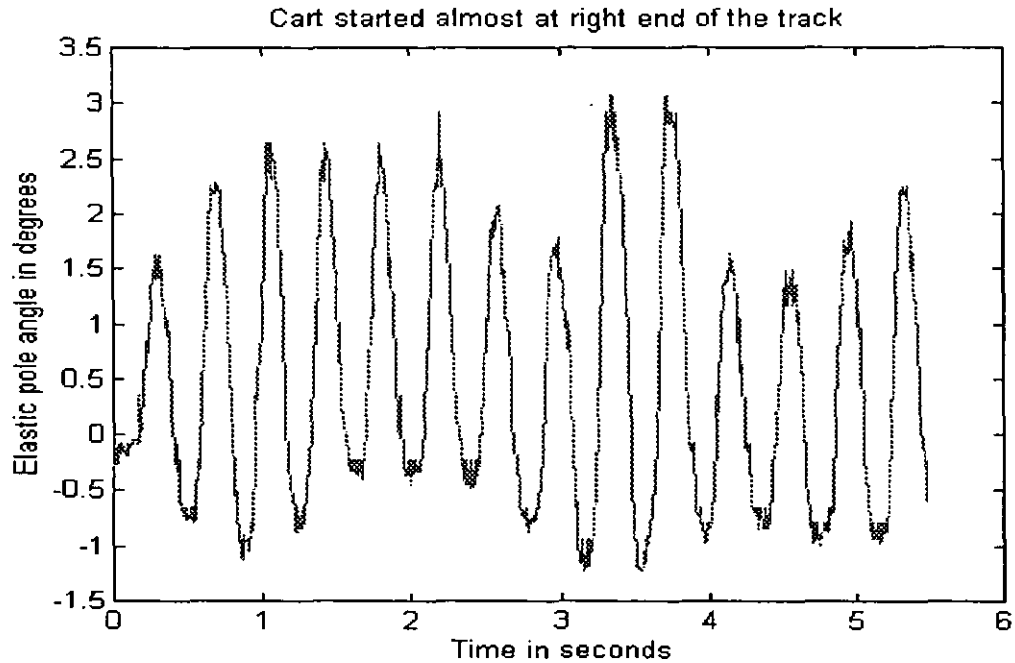


Figure 5.9(i)

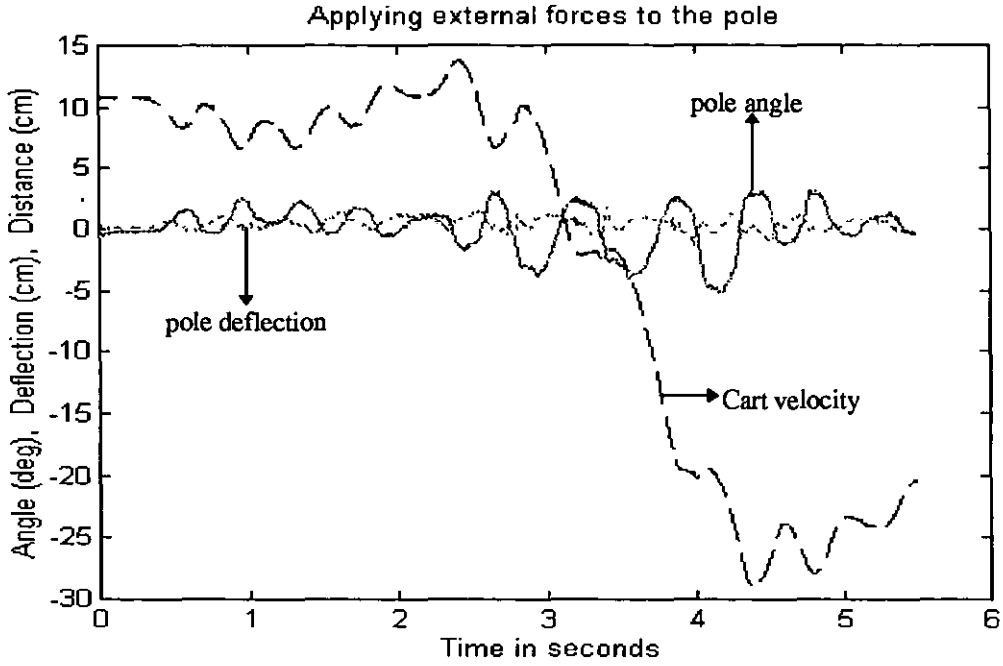


Figure 5.9(ii)

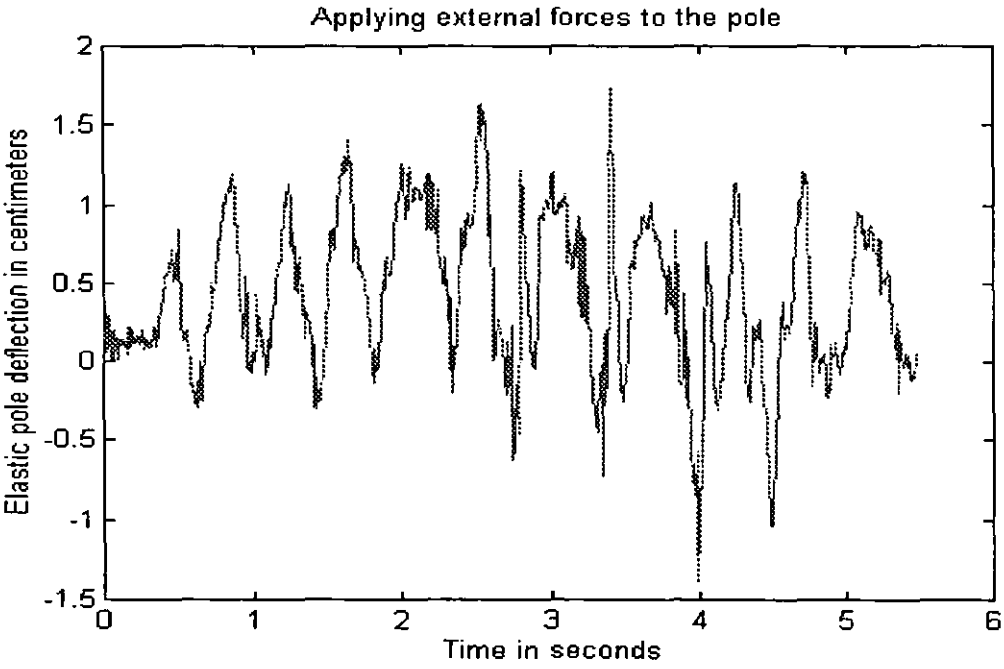


Figure 5.9(iii)

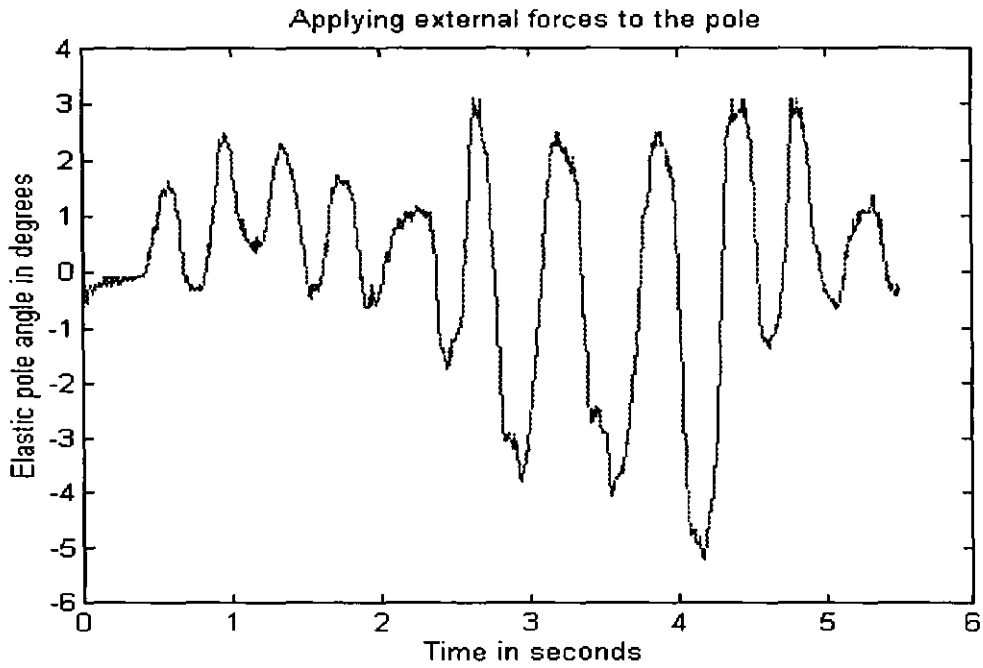


Figure 5.10(i)

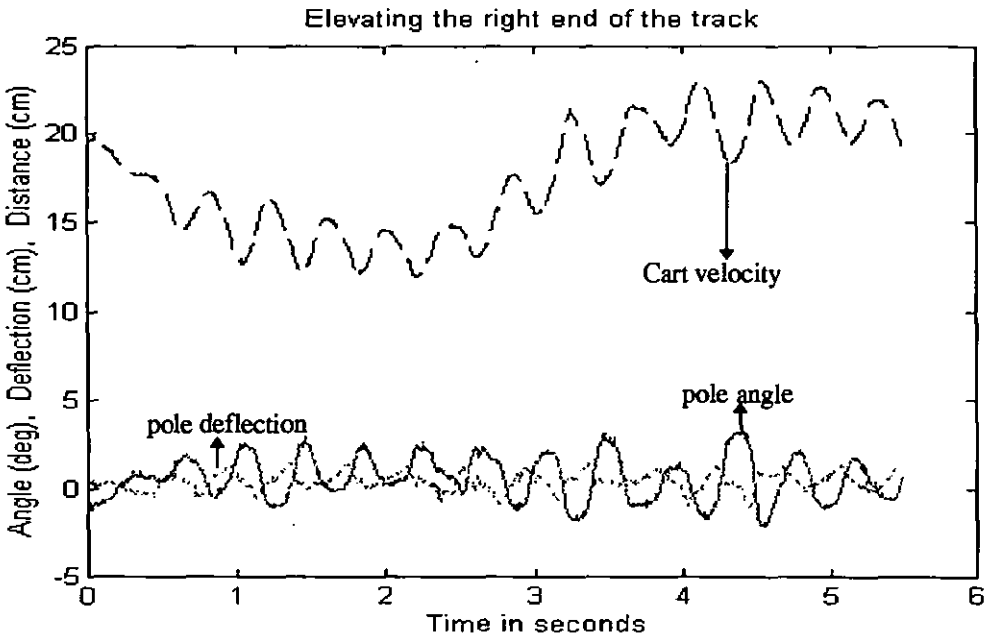


Figure 5.10(ii)

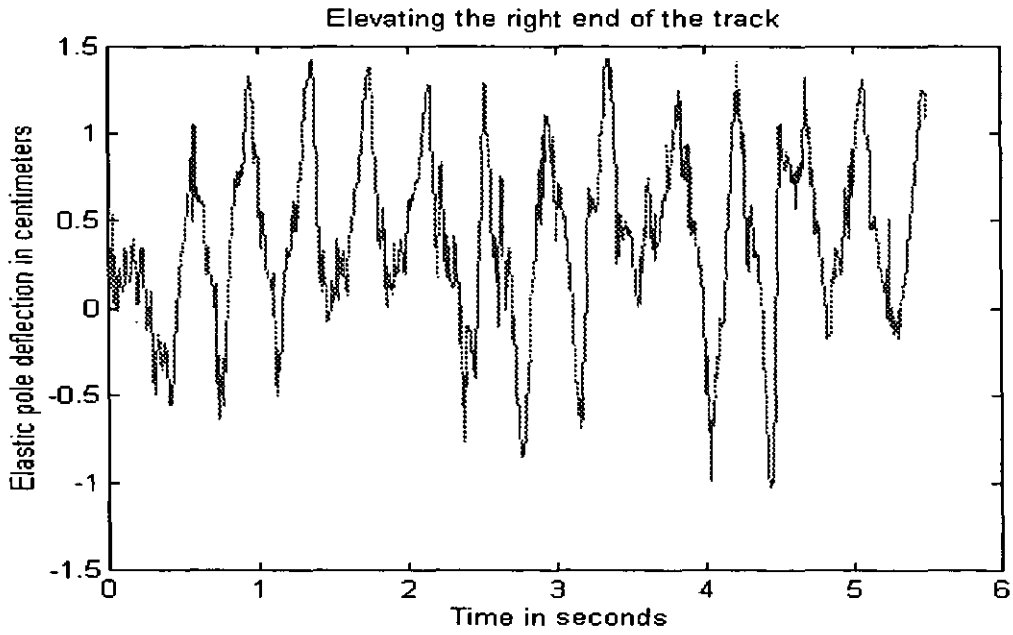


Figure 5.10(iii)

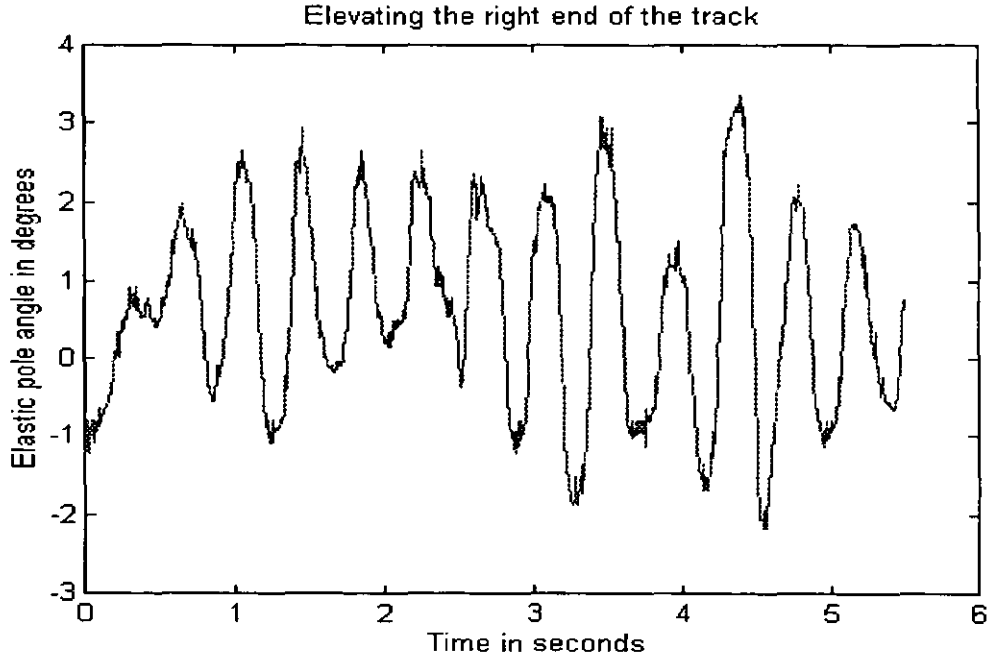


Figure 5.11(i)

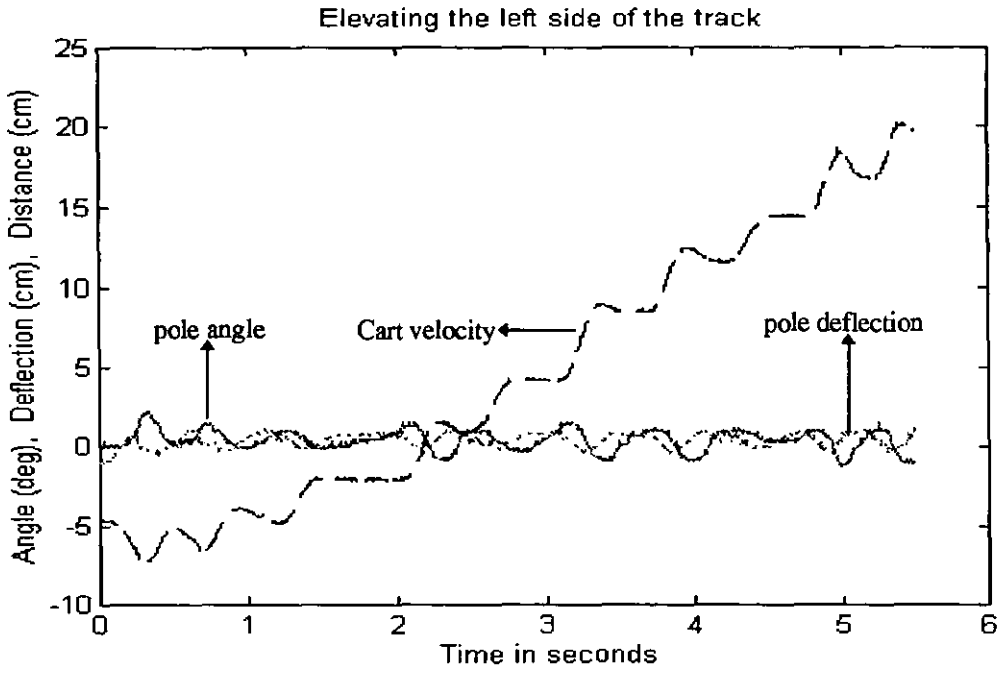


Figure 5.11(ii)

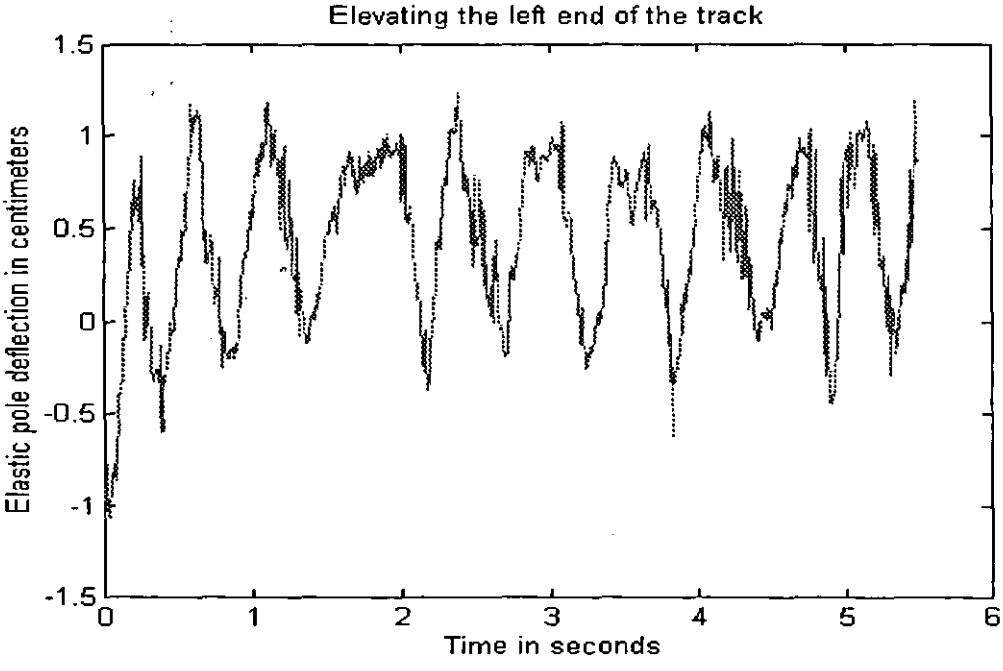


Figure 5.11(iii)

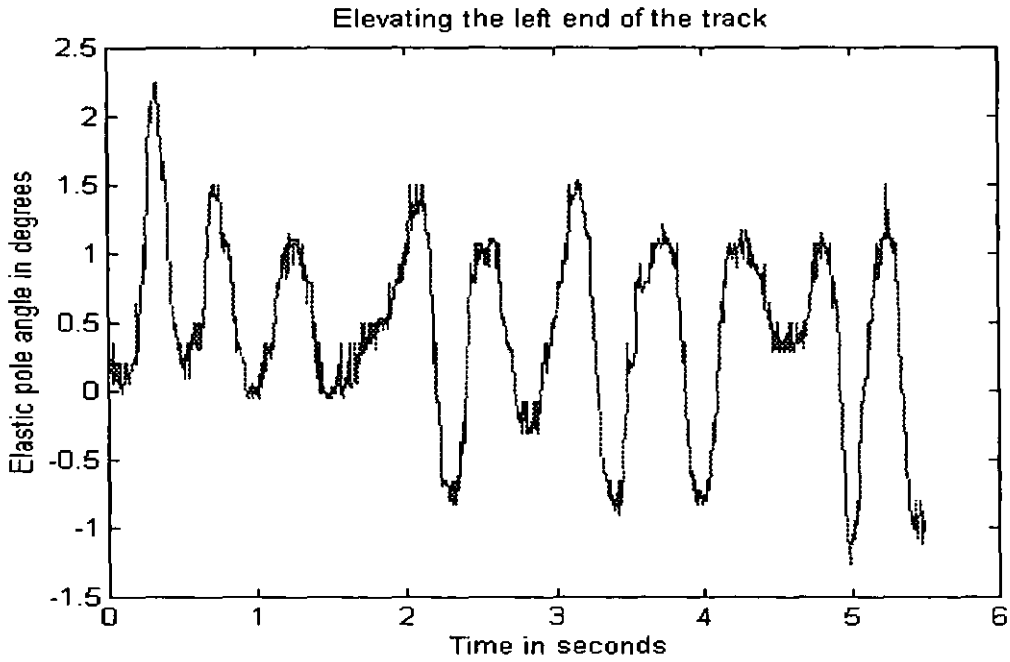


Figure 5.12(i)

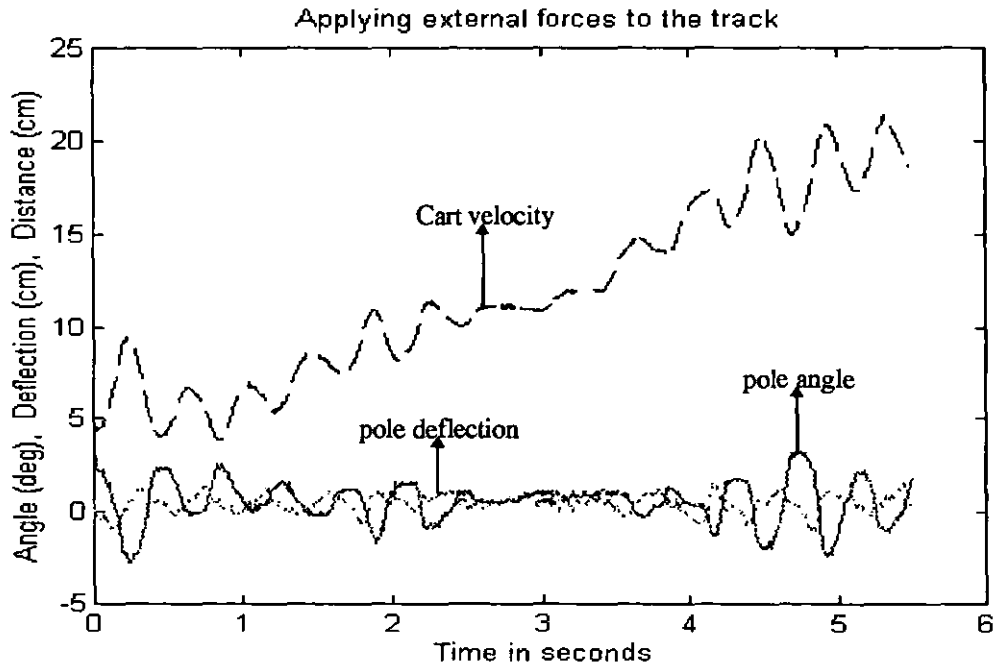


Figure 5.12(ii)

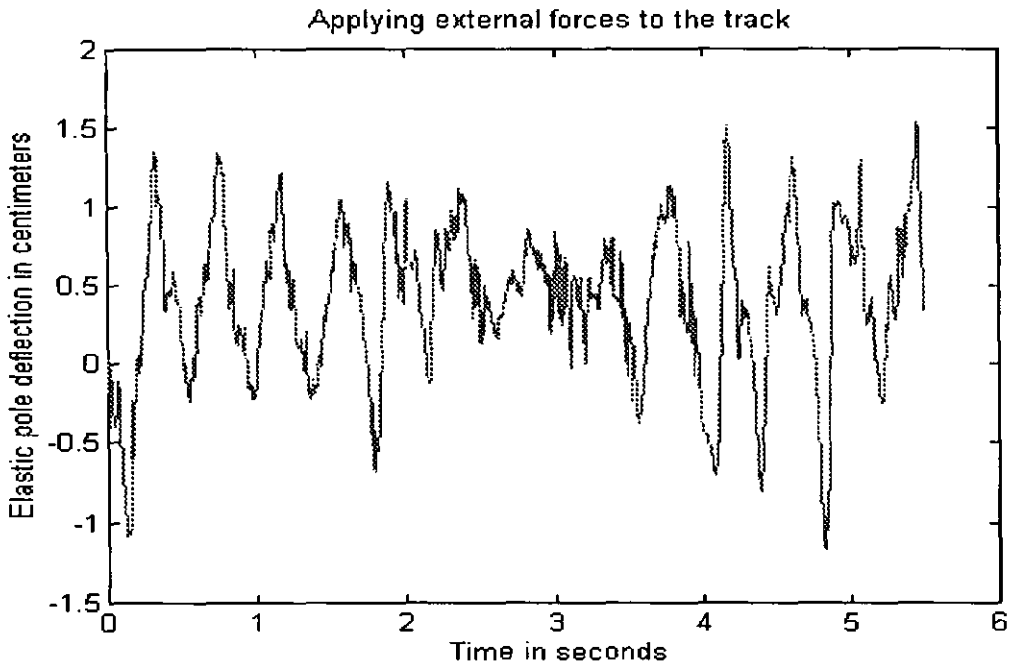


Figure 5.12(iii)

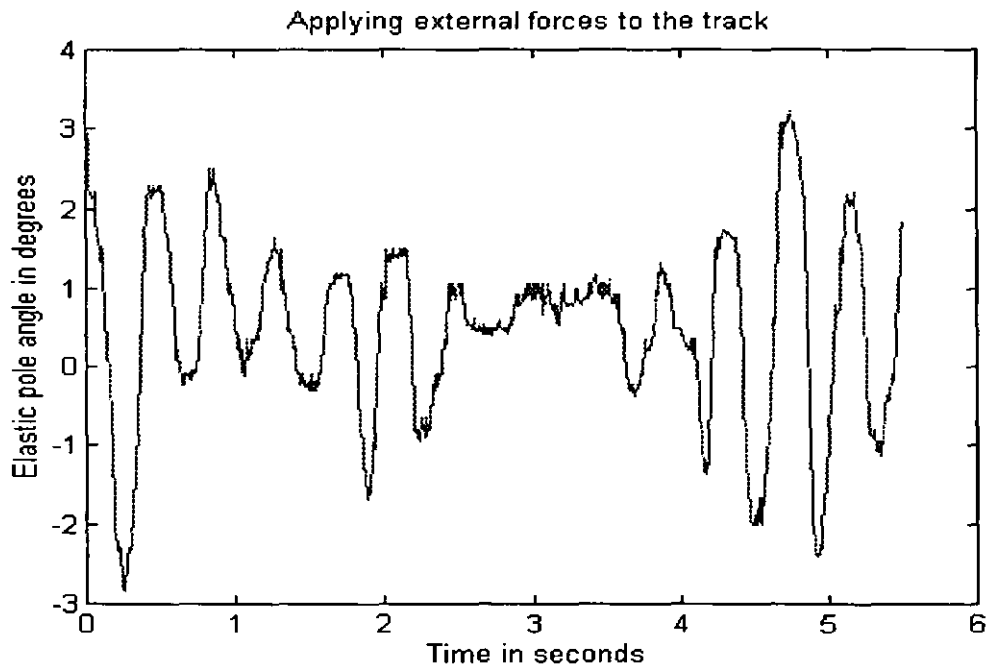


Figure 5.13(i)

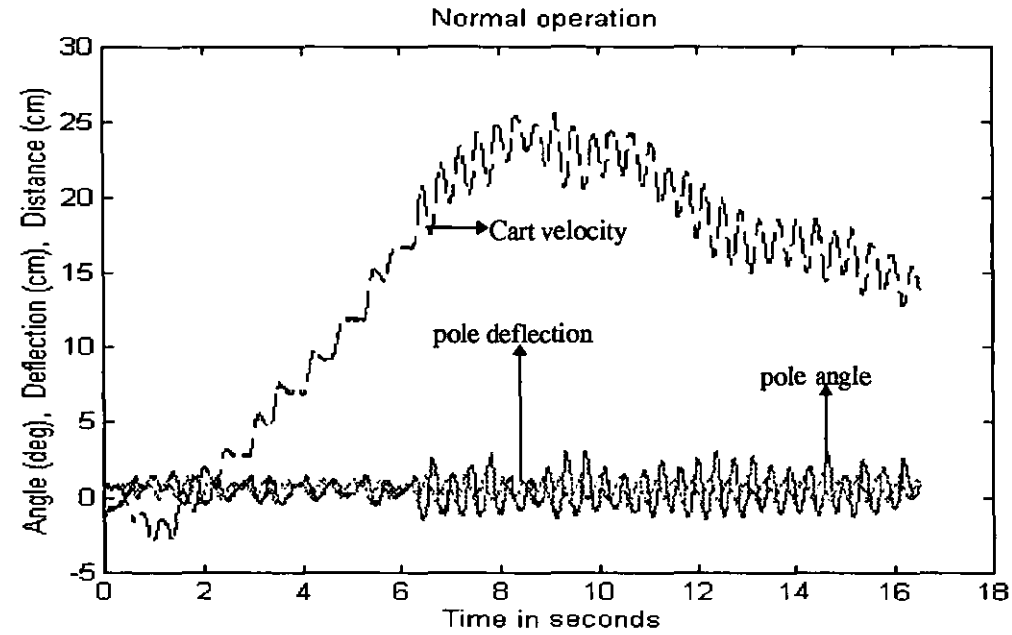


Figure 5.13(ii)

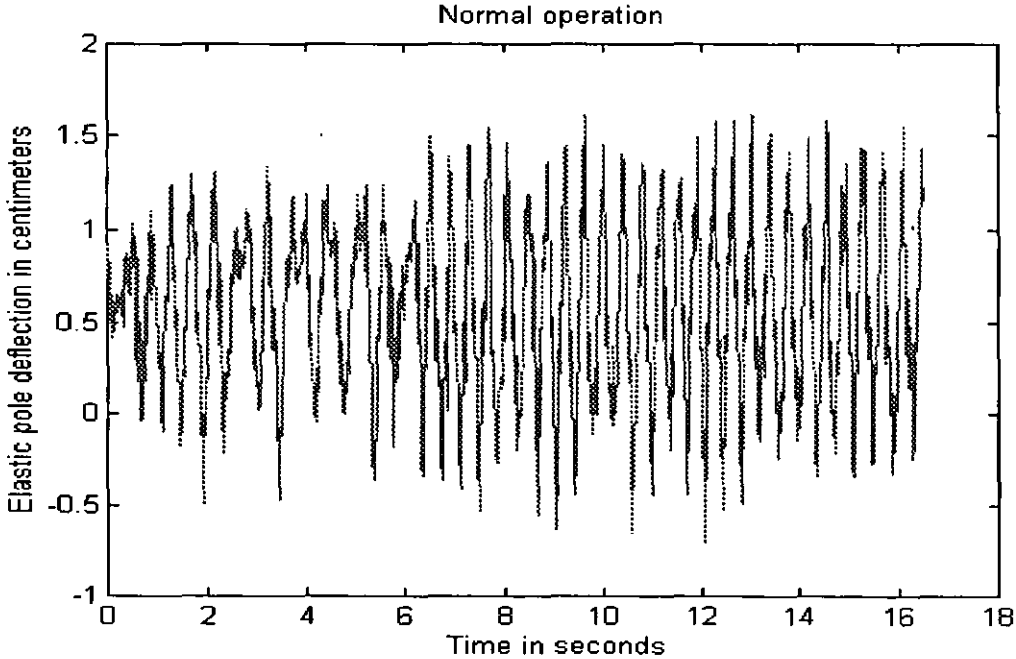


Figure 5.13(iii)

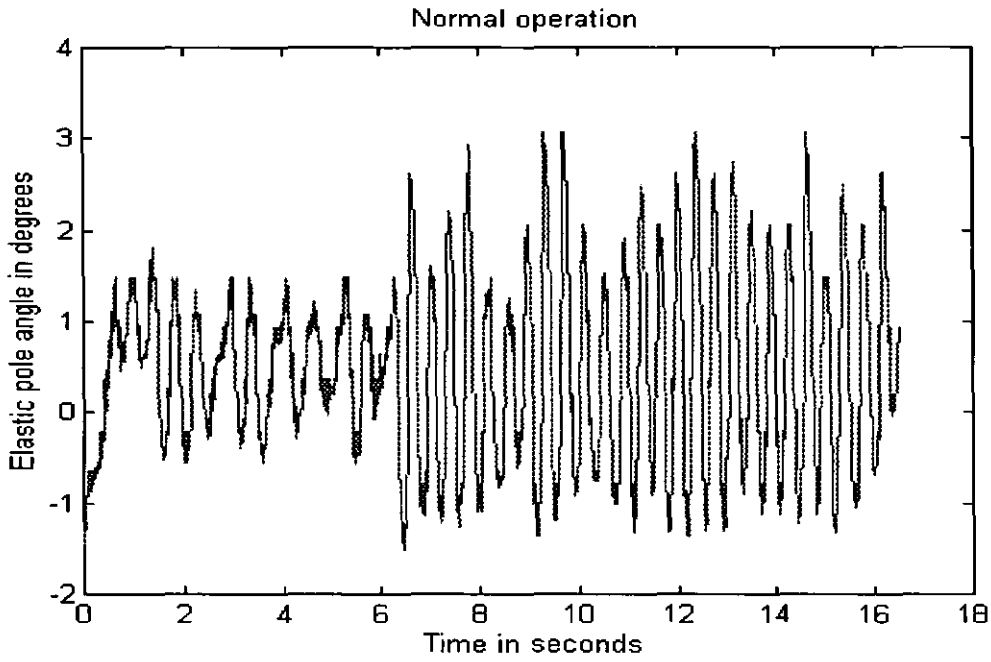


Figure 5.14(i)

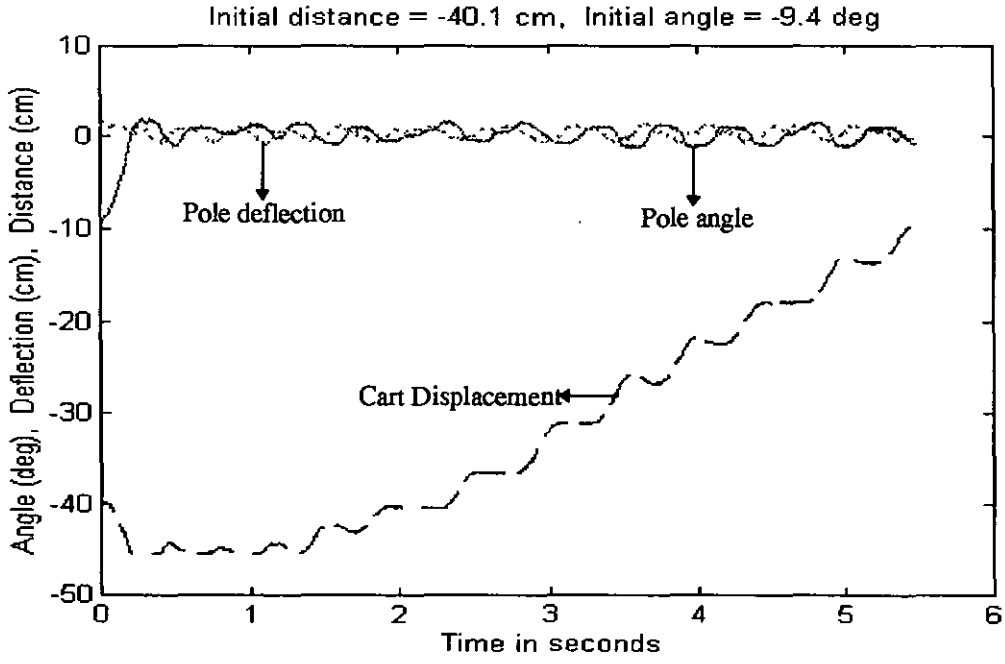


Figure 5.14(ii)

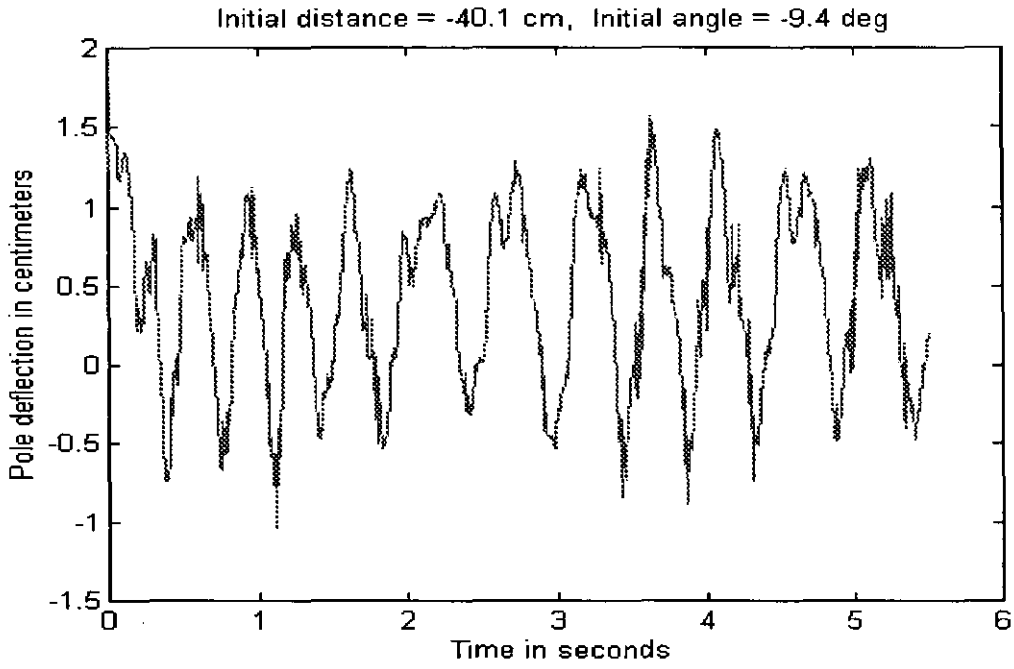


Figure 5.14(iii)

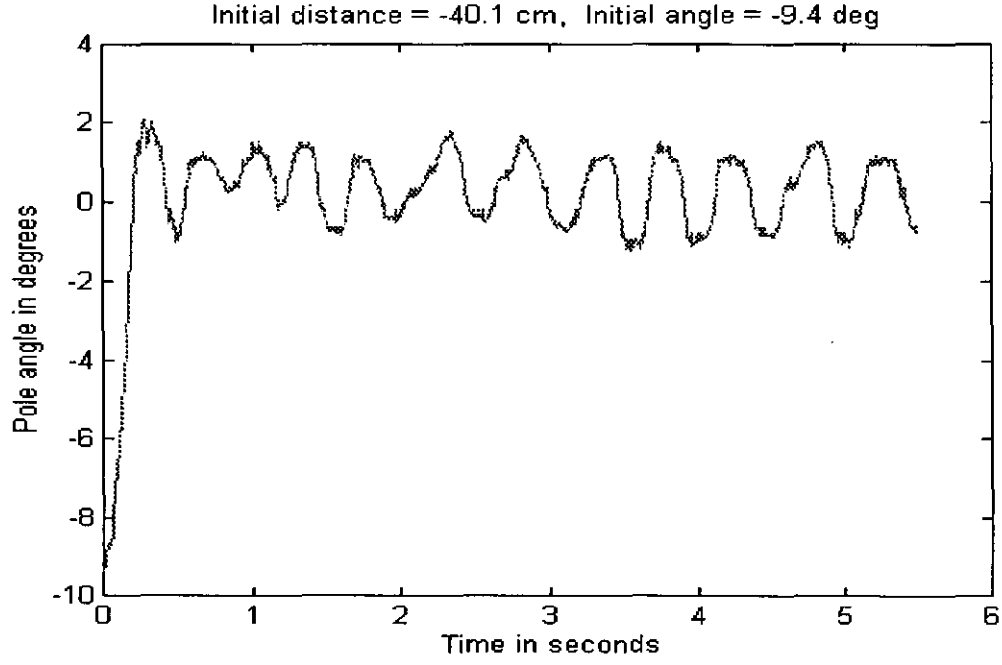


Figure 5.15(i)

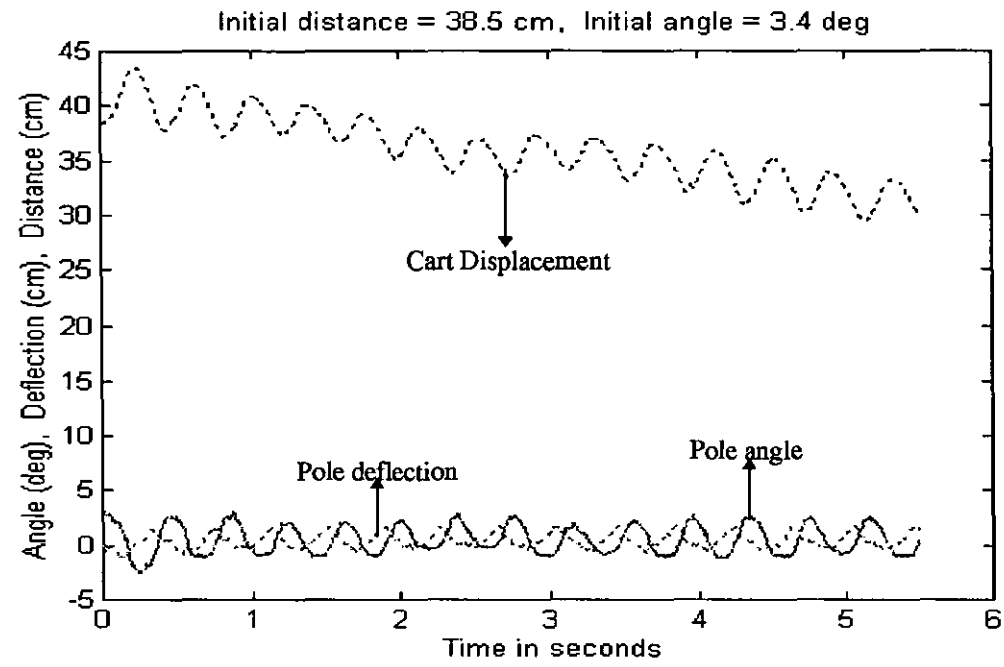
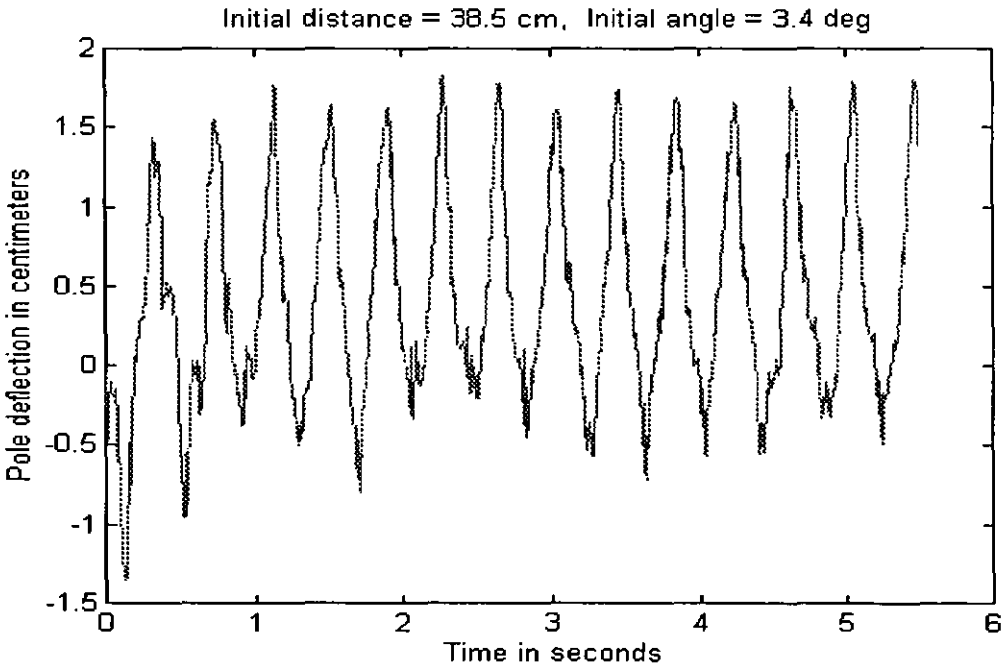


Figure 5.15(ii)



5.5. Summary

An on line reinforcement learning hybrid neural network controller was developed to balance a flexible pole hinged root on top of the cart moving along a limited track. The physical experiments show that the controller not only balances the flexible pole but also brings the cart to the centre of the track for infinite time. The learning controller developed is sufficiently robust to control the system at different initial pole angles and different initial cart positions on the track. The stability, flexibility, and adaptability of this learning controller was tested by applying external disturbances to the plant.

The next chapter of this thesis discusses the development and test of an on line intelligent controller that controls the flexible pole-cart balancing system without knowing the mathematical descriptions of the dynamics of the system, using a fuzzy logic control system.

CHAPTER 6

Multiple Fuzzy Logic Systems: An on line controller for the Flexible Pole-Cart Balancing Problem

6.1. Introduction

Classical controllers are designed on the basis of mathematical descriptions such as differential equations or transfer functions, while modern controllers use first order vector matrix differential equations based on the state space method [54]. In these techniques, a controller designer has to possess extensive knowledge of both mathematics and the system under control. However, an experienced person can skillfully control vehicles, machines and manufacturing plants even though the systems under control are very complex and nonlinear. These experts mostly use know-how which has been gathered from experience. This suggests that there is another technique which can facilitate the control of a complicated system without knowledge of its mathematical description. This technique is popularly known as fuzzy logic control - the use of fuzzy inference to control a system.

The concept of fuzzy logic was introduced by Zadeh in 1965 [55]. This system is unique in that it is able to simultaneously handle numerical data and linguistic knowledge. It is a nonlinear mapping of an input data vector into a scalar output, i.e. it maps numbers into numbers but fuzzy set theory and fuzzy logic establish the specifics of the nonlinear mapping [56]. The applications of this technique are multi-disciplinary in nature. These include, for example, automatic control, consumer electronics, signal processing, information retrieval, time series prediction, database management, computer vision, data classification and decision making [57]. The application of fuzzy logic to control problems was introduced by Mamdani in 1975 [59, 59].

This work presents the flexible pole-cart balancing problem as a testbed for fuzzy logic applications. Here, the objective is to develop and test an on line fuzzy logic

controller that predicts the value of the force applied to the cart at any given time in order to balance the flexible pole hinged at its root on top of the cart. In this work multiple fuzzy logic systems have been used to fuzzify the input data from the environment. There are six input data to the system (the elastic pole deflection, deflection velocity, angular position, angular velocity, cart displacement, and cart velocity). Results of the physical experiments are shown graphically in section 6.9.2.

This chapter begins by the discussion of the concepts and architecture of a fuzzy logic systems and it continues by the development of a fuzzy logic controller for the flexible pole-cart balancing problem. The results of on line experiments conducted on this controller are presented.

6.2. Fuzzy Logic System

A fuzzy logic system is a system design that is based on how the human brain thinks. It arose from the desire to describe complex systems with linguistic description [54]. Fuzzy logic looks at the world in imprecise terms in much the same way that our own brain takes in information. The information is described in terms of fuzzy linguistic terms. These fuzzy linguistic terms are called fuzzy sets and can be regarded as sets of singletons, the grades of which are not only 1 but also ranging from 0 to 1. Each singleton is an element of fuzzy sets.

The concept of fuzzy sets is made precise through the definition of an associated membership function. This membership function indicates a grade of membership of each element (physical value) in a fuzzy linguistic term of interest. Fuzzy membership functions are the mechanism through which the fuzzy system interfaces with the outside world [60]. The domain of the membership function is the set of possible values for a given variable. The possible output values of the membership function is the set of all real numbers from 0 to 1. A typical choice of the shape of the fuzzy membership function is a triangular, trapezoidal, or a gaussian function.

Fuzzy sets can be combined through fuzzy rules to define specific actions. The fuzzy system can provide insight into their own operation because the fuzzy rules provide a commonsense description of the system own action. The technique used to store and represent fuzzy rules is the fuzzy associative memory matrix (see section 6.6). This matrix may have dimensions higher than two. Usually the number of inputs, or antecedents, to the fuzzy rules determines the dimension of the matrix.

Figure 6.1 depicts a fuzzy logic system that is widely used in fuzzy logic controllers and signal processing applications [56]. It contains four components: fuzzifier, rules, inference engine, and defuzzifier. Once the rules have been established, a fuzzy logic system can be viewed as a mapping of inputs to outputs.

Rules are expressed as a collection of IF-THEN statements, e.g. IF the *cart position is far left* and *the pole angle inclined more to the right* THEN *apply a force to the right*. This rule reveals that it is necessary to understand linguistic variables versus numerical values of a variable (e.g., *angle inclined more to the right* versus *20 degrees)*. It is also necessary to quantify linguistic variables (e.g., *how much force will be applied to the right*). This can easily be done using fuzzy membership functions (see section 6.7).

The fuzzifier maps crisp input numbers into fuzzy sets. It is needed in order to activate rules which are in the terms of linguistic variables, which have fuzzy sets associated with them. The inference engine maps fuzzy sets into fuzzy sets. It handles the way in which rules are combined. The defuzzifier maps output set into crisp numbers (e.g., in control application, such a number corresponds to the control action).

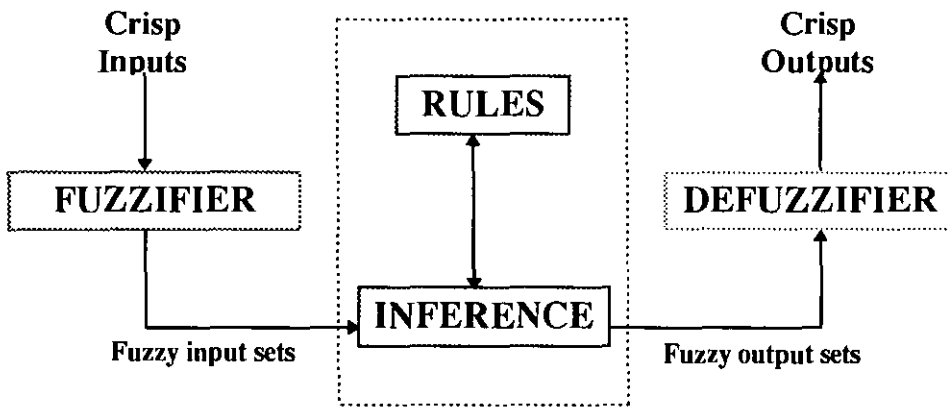


Figure 6.1
The fuzzy logic system (FLS)

6.3. The Flexible Pole-Cart Balancing System

The task of the flexible pole-cart balancing system is to balance an elastic pole that is hinged on a movable cart [72, 73, 74, 75]. It is assumed that the hinge is frictionless. The cart is allowed to move along a track with limited length and that has friction. Forces of different magnitude are applied to the cart in either a left or right direction to balance the pole. Figure 3.1 shows the diagram of the dynamics of the system. As has been described earlier, in the real physical system (see section 5.2), the length of the track that the cart can travel is 91.4 cm . The length of the pole is 41.0 cm. The total mass of the cart and the camera sensor is 0.755 kg. An additional load of 0.35 kg is attached on the tip of the pole to increase its elastic deflection to make the control problem more testing.

6.4. Processes Involved in the Formulation of the Flexible Pole-Cart Balancing Control: Fuzzy Logic Perspective

This section describes the various processes involved in formulating the problem from a fuzzy logic perspective and provides a specification of the application of a fuzzy logic controller to the flexible pole-cart balancing system. These processes are briefly discussed below.

It is important to know all the variables to be used in the controller including the input data, the variables within the fuzzy rules (e.g. antecedents and consequents) and their maximum and minimum values. In this controller, there are 6 input data (the cart displacement, cart velocity, pole deflection, pole deflection velocity, pole angle, and pole angular velocity). The rules have 2 variables for the antecedents and 7 variables for the consequents. It has been found helpful sub-divide the complex problem into its elements. Therefore cart displacement, pole angle, and pole deflection are handled separately, to make the controller design simple.

It is necessary to establish the shape of membership function suitable for describing each problem element (triangular, trapezoidal, gaussian, etc.) and number of regions and their range for each membership function. The shape is not necessarily the same for each problem element. The number of membership functions corresponds to the number of the input regions of the fuzzy rules. In this controller, there are 5 regions created (see section 6.7) to limit the usage of computer memory which increases with the number of fuzzy rules and regions.

It is also important to know the strategy to be used in selecting useful sets of fuzzy rules and the conjunctions within the rules. Knowledge of how and when to combine more than one rule is helpful. In this controller each fuzzy logic system has only 13 rules (see section 6.6). Note that, in this application, we are particularly concerned with cart position if the cart is in the negative, left, part of the track and heading further left, or in the positive, right, track and heading further right. Similarly, if the pole angle and deflection is too large or changing too fast, this should be corrected regardless of the location of the cart on the track. In this controller we have used 6 variables and 5 input regions. If all of these variables were used in the antecedents and 5 regions are adopted for each variable, 5^6= 15625 rules must be examined. It is impossible for the designer to generate this large rule set. In order to cope with this problem, the number of variables must be reduced. In order to achieve this the author has implemented multiple fuzzy logic systems (see section 6.5).

Any controller must be tested until it works as effectively as possible. The effect of varying the fuzzy rules, the variables and the range of the values and shape of the membership functions was explored by experiment. It was also found necessary to check the values of the sensors (e.g. the input data).

6.5. Application of Fuzzy Logic Controller to the Real Physical Flexible Pole-Cart Balancing System

Figure 6.2 shows the fuzzy logic controller generated for the flexible pole-cart balancing problem. 5 fuzzy logic systems (FLS) and a rule based evaluator are used to control the flexible pole-cart balancing system. The architecture of the fuzzy logic systems is shown and discussed in full in section 6.2. The importance of using multiple FLS is to minimize the memory consumption of the computer, and each FLS serves as a good filter to the noise on the input data. FLS1 is a fuzzy logic system that maps the cart displacement and cart velocity to the crisp output1. Crisp output1 corresponds to the crisp numerical value that will compensate for the effect of the movement of the cart on the overall system. FLS2 is a fuzzy logic system that maps the pole angle and angular velocity to crisp output2. Crisp output2 corresponds to the crisp numerical value that will compensate for the effect of the movement of the flexible pole on the overall system. FLS3 is a fuzzy logic system that maps the pole's deflection and deflection velocity to crisp output3. Crisp output3 is the crisp numerical value that will compensate for the effect of the movement of the flexible pole, due to its deflection and deflection velocity, on the overall system.

Since the contribution of the effects of crisp output2 and crisp output3 to the plant are similar, the two of them can be fuzzified further using FLS4. This maps crisp output2 and crisp output3 to crisp output4. Crisp output4 is the crisp numerical value that will compensate the fuzzified effect of the movement of the pole due to its angular position, angular velocity, deflection, and deflection velocity, on the overall system. In order to obtain the overall crisp value that will compensate the effect of the total movement of the system, crisp output1 and crisp output4 map to crisp output5 through FLS5 which will fuzzify further the fuzzified effect in FLS1 and FLS4. The FLS5 process filters the noise on the final data required to control the system. Finally, to ensure that the cart stays at the center of the track a rule based evaluator (see figure 6.2) is used to evaluate the condition of the plant. The evaluator adds additional constant forces to those

supplied by the fuzzy system when the cart exceeds particular displacements. The output of the rule based evaluator is then fed to the plant for appropriate action. External disturbances can be applied to the plant at any time without affecting the performance of the controller.

6.6. Fuzzy Associative Memory (FAM) Matrix

The FAM matrix is a method of storing and representing fuzzy rules. In this controller, each fuzzy logic system (FLS) has two inputs. Each input variable has 5 fuzzy sets associated with it, which are labeled NL (negatively large), NS (negatively small), ZE (zero), PS (positively small), and PL (positively large). Note that here ZE is a fuzzy set that would typically represent a range of values near 0, not just a single numerical value 0. The output variable has 7 fuzzy sets associated with it: *NL (negatively large), NM (negatively medium), NS (negatively small), ZE (zero), PS (positively small), PM (positively medium), and PL (positively large).* The number of inputs, or antecedents, to the fuzzy rules determines the dimension of FAM matrix. Thus, in this controller we are using a 2 dimensional FAM matrix.

Note that, we are particularly concerned with the pole position if the pole angle is too large and increasing. The same is true for its deflection. These need to be corrected regardless of the location of the cart in the track by applying maximum force to the cart with the same direction as the inclination of the pole. Similarly, if the cart is too near the end of the track, this should be corrected regardless of the state of the flexible pole (angle and deflection) by applying maximum force to the cart towards the end of the track, making the flexible pole incline more in the other direction, thus in turn allowing the controller action to balance the flexible pole by applying more force to the opposite side of the cart and at the same time bringing it to the center of the track.

The FAM matrix for FLS1 is shown in table 6.1. This is used to fuzzify the cart's displacement and velocity to obtain a crisp result (output) that can compensate the effect

of the dynamic movement of the cart (due to its displacement and velocity) on the system. These rules can be interpreted as :

Rule1 : IF X is NL and $\dot{X}$ is NL THEN the output result is ***NL***.

Rule2 : IF X is NL and $\dot{X}$ is ZE THEN the output result is ***NM***.

Rule3 : IF X is NL and $\dot{X}$ is PL THEN the output result is ***PS***.

and so on . . .

Note that :

X = the displacement of the cart.

$\dot{X}$ = the velocity of the cart.

The FAM of the other FLS's are shown in tables 6.2 to 6.5. The operations of these are similar to table 6.1.

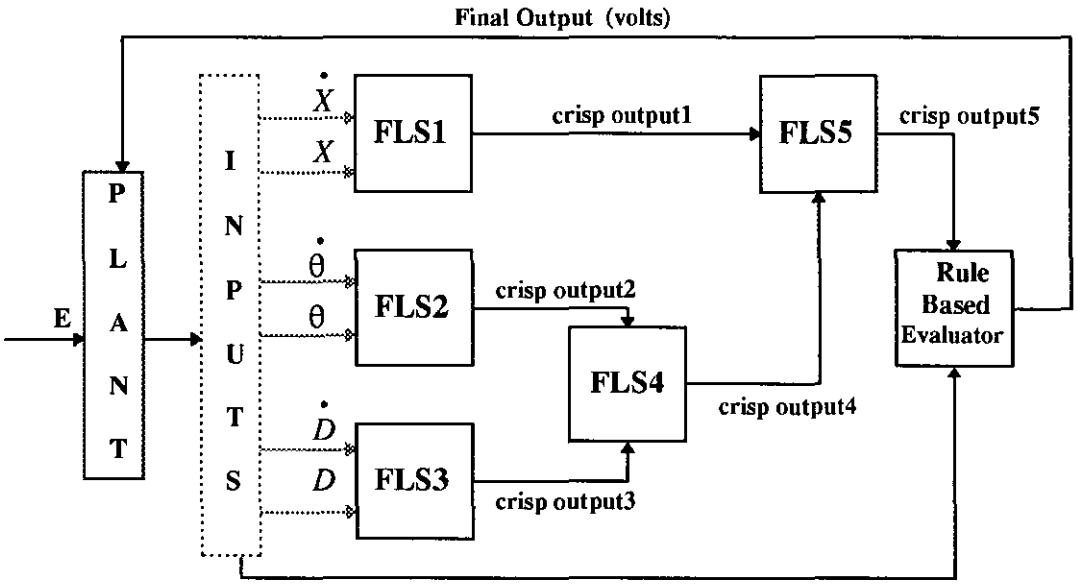


Figure 6.2
Multiple fuzzy logic controller block diagram

X \ $\dot{X}$	NL	NS	ZE	PS	PL
NL	*NL*		*NM*		*PS*
NS		*NM*		*NS*	
ZE	*NS*		*ZE*		*PS*
PS		*PS*		*PM*	
PL	NS		*PM*		*PL*

Table 6.1
(Fuzzy associative memory matrix for FLS1)

θ \ $\dot{\theta}$	NL	NS	ZE	PS	PL
NL	*NL*		*ZE*		*ZE*
NS		*NM*		*NS*	
ZE	NS		*ZE*		*PS*
PS		*PS*		*PM*	
PL	ZE		*ZE*		*PL*

Table 6.2
(Fuzzy associative memory matrix for FLS2)

This is used to fuzzify the pole's angle and angular velocity to obtain a crisp result (output) that will compensate the effect of the movement of the pole (due to its angular position and angular velocity) to the entire system.

D \ $\dot{D}$	NL	NS	ZE	PS	PL
NL	*NL*		*ZE*		*ZE*
NS		*NM*		*NS*	
ZE	*NS*		*ZE*		*PS*
PS		*PS*		*PM*	
PL	*ZE*		*ZE*		*PL*

Table 6.3

(Fuzzy associative memory matrix for FLS3)

This is used to fuzzify the pole's deflection and deflection velocity to obtain a crisp result (output) that will compensate the effect of the movement of the pole (due to its deflection and deflection velocity) to the entire system.

D fs $\dot{D}$ \ θ fs $\dot{\theta}$	NL	NS	ZE	PS	PL
NL	*NL*		*NS*		*PS*
NS		*NM*		*PS*	
ZE	*ZE*		*ZE*		*ZE*
PS		*NS*		*PM*	
PL	*NS*		*PS*		*PL*

Table 6.4

(Fuzzy associative memory matrix for FLS4)

This is used to fuzzify further the fuzzified effect of the movement of the pole's angle and angular velocity with the fuzzified effect of the movement of the pole's deflection and deflection velocity to obtain a crisp result (output) that will compensate the effect of the movement of the pole (due to its angular position, angular velocity, deflection, and deflection velocity) to the entire system.

$(\theta \text{ fs } \dot{\theta}) \text{ fs } (D \text{ fs } \dot{D})$

$X \text{ fs } \dot{X}$

	NL	NS	ZE	PS	PL
NL	*NL*		*NM*		*NS*
NS		*NM*		*NS*	
ZE	*NS*		*ZE*		*PS*
PS		*PS*		*PM*	
PL	*PS*		*PM*		*PL*

Table 6.5

(Fuzzy associative memory matrix for FLS5)

This is used to fuzzify further the fuzzified effect of the movement of the pole's angle, angular velocity, deflection, and deflection velocity with the fuzzified effect of the movement of the cart's displacement and velocity to obtain a crisp result (output) that will compensate the effect of the total movement of the entire system.

6.7. Membership Functions (MF's)

Membership functions map each element of "universe of discourse" to a continuous membership value (or membership grade) between 0 and 1. The "universe of discourse" may contain either discrete objects or continuous values. In this controller each membership function is sampled to discrete grades, whose representation depends on the type of input variables (e.g., -5.0 to 5.0 centimeters for the cart displacement, -5.0 to 5.0 degrees for the pole angle, etc.). The shape and the regions of the membership function can be changed by reassigning its grade distribution as shown in figures 6.3 to 6.5. Determination of the shapes of each membership function usually requires some trial and error [60]. The exact shape of the functions, as well as where they intersect the horizontal axis and how much overlap exists between adjacent functions, is open to experimentation.

There are two shapes of membership functions used in this controller (see section 6.9.1).

1. *Trapezoidal MF's* specified by four parameters *{a,b,c,d} which determine the* x *coordinate* as follows: *trapezoid(x; a,b,c,d)* $= \max(\min(\frac{x-a}{b-a},1,\frac{d-x}{d-c}),0)$ **(6.1)**
2. *Triangular MF's* specified by three parameters *{a,b,c}* which determine the x coordinate as follows: *triangle(x; a,b,c)* $= \max(\min(\frac{x-a}{b-a},\frac{c-x}{c-b}),0)$ **(6.2)**

The leftmost and rightmost regions of the MF of figures 6.3 to 6.5 are an open trapezoid whose values for d and c are equal to 0. Other shapes of MF are triangles. Obviously a triangular function is a special case of a trapezoidal function.

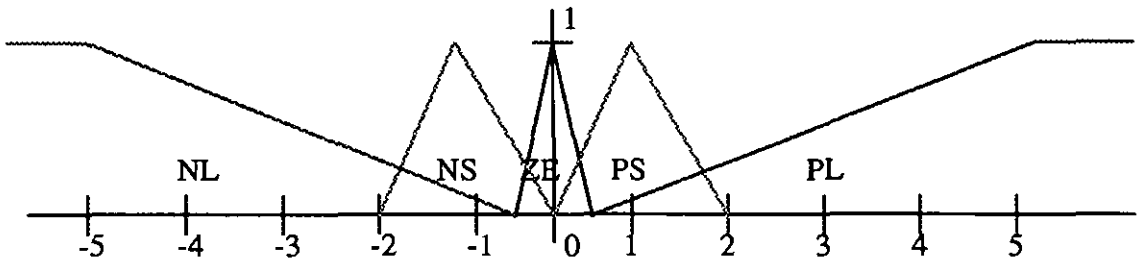


Figure 6.3
Membership functions for the cart's displacement

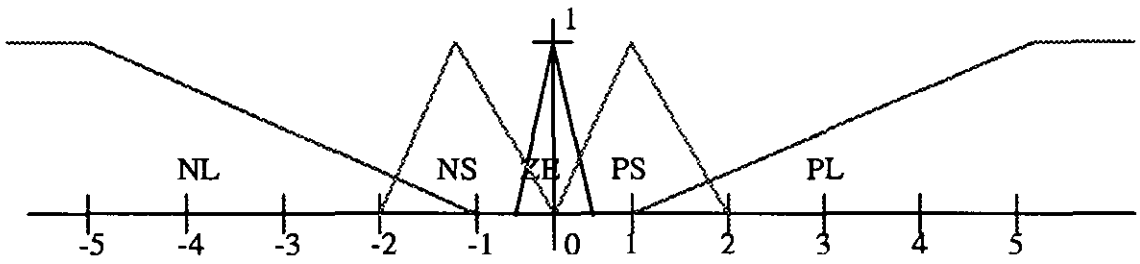


Figure 6.4
Membership functions for the cart's velocity, pole's angular position, and pole's angular velocity

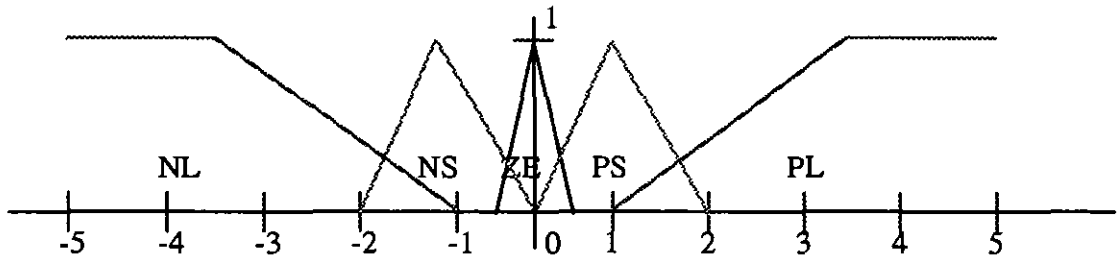


Figure 6.5
Membership functions for pole's deflection, and pole's deflection velocity

6.8. Defuzzifier

A defuzzifier is a way of obtaining a deterministic value, in the universe of discourse, from a fuzzy value (membership function). The most popular method of defuzzification is a center-of-gravity method [61, 62]. In this research our "universe of discourse" contains discrete objects, thus our membership function is represented by a sampled data (a set of elements). The center-of-gravity (C.G.) for discrete membership functions can be calculated using equation 6.3.

$$\text{Crisp output} = \text{C.G.} = \frac{\sum_{i=1}^{n} o_i \cdot \mu_i}{\sum_{i=1}^{n} \mu_i} \qquad (6.3)$$

where n represents the number of elements of the sampled membership function, μ_i the grade of *ith* element, and o_i the output variable of the *ith* fuzzy set. The maximum value of n is equal to the total number of fuzzy rules in the FLS. The value of μ_i can be calculated using equation 6.1 or 6.2. For this particular system the output variable o_i has 7 fuzzy sets associated with it. (e.g. NL,NM,NS,ZE,PS,PM,PL). The value of the output variable is based on the voltage capacity of the actuator, hence has specific values in volts:

NL = -4.75, NM = -2.65, NS = -1.35, ZE = 0.0, S = 1.35, PM = 2.65, PL = 4.75.

These values were derived through experimental observation of the flexible pole-cart balancing process.

6.9. Results of the Physical Experiments

6.9.1. Discussion and analysis

The author conducted a number of different sets of experiments in this work. The first experiment was to attempt to develop a single fuzzy logic system (FLS) to control the plant (the flexible pole-cart balancing system). The application of this controller was not encouraging. The controller was very sensitive to noise from the pole deflection as the values of the deflection of the pole and the deflection velocity can vary abruptly. In order to further eliminate this noise a controller was built with multiple FLS. This technique is effective because each FLS acts as a noise filter.

In this work, different total numbers of fuzzy rules were also applied. Attempts were made using 27, 75, 135 rules, etc. Unfortunately the results of the application of these controllers are not appropriate because increasing the number of rules increased the memory consumption of the computer program. Attempts to change the number of input regions (number of membership functions) to 3 regions did not give encouraging results. Best results were obtained using 5 regions (see figures 6.3 to 6.5). The size of these regions plays an important role. The more regions overlap each other, the better is the result, because of the design aim to use minimum number of rules. Choosing the exact position where regions overlap is critical and depends on knowledge of the physical structure of the plant and the capability of the sensors (e.g. knowing the exact size of the track, the minimum and maximum deflection of the pole as well as its angle for the system to operate, etc.). The shape of the regions is also important. It can be seen in figures 6.3 to 6.5 that open trapezoids were used in the leftmost and rightmost regions. Whenever the plant reaches these positions (the beginning of the horizontal line and beyond), the controller gives a maximum output value to the system.

Since the output of the FLS is based on generalized results, the pairing of input variables (antecedents of the fuzzy rules) for fuzzification is particularly important. Good

results occur when input data that have similar characteristics (e.g., $[X, \dot{X}]$, $[\theta, \dot{\theta}]$, $[D, \dot{D}]$, etc.) are combined. This technique is effective in building multiple FLS. As discussed earlier this eliminates the excessive noise on the flexible pole's deflection sensor. After fuzzifying $[\theta, \dot{\theta}]$ and $[D, \dot{D}]$, we further fuzzify the two results together.

Selecting the fuzzy associative memory (FAM) matrix plays a vital role in the process. FAM matrices of smaller dimensions are easier to deal with. Although we have 6 inputs from our plant, we have been able to reduce the size of our FAM matrix by considering two inputs at a time. It can be seen from section 6.6 that the input of each FAM matrix has 5 fuzzy sets. This means that we have 5x5 = 25 possible fuzzy rules generated. However, there is no need to assign all of these rules because membership functions (see figures 6.3, 6.4, 6.5) are assigned in such a way that neighboring membership functions penetrate each other. This means that a defect in one rule can be compensated (interpolated) by the surrounding four rules [1]. Thus this technique enables us to minimize the total number of fuzzy rules in our FLS.

In this work, the accuracy of the sensor initial values (offsets) are important. The fuzzy controller design assumes that there are zero sensor values when the system is balanced. Unfortunately, in the real physical system it is extremely difficult to achieve this (i.e., all the values of the sensors = 0 when the pole is perfectly balanced on the center of the track). This initialization difficulty causes a slight offset in the data to the controller that leads to the cart traversing off the track. This problem was resolved using by the rule based evaluator (see figure 6.2) thus correcting for the initial transducer offset errors.

6.9.2. Graphical results

The graphs of figures 6.6(i) to 6.9(i) present the results of on line application of the fuzzy logic controller. These figures show the complete status of the system with respect to time (i.e. the pole angle, pole deflection, and the cart displacement). The

movement of the cart is shown by the graph of the cart displacement, and the movement of the pole by the graphs of pole deflection and angle.

Figure 6.6(i) shows the result of operating the system at an initial condition of: pole angle = -20.5 degrees, pole deflection = 1 cm, and cart displacement = 5 cm. Here, the cart initially moved quickly to the left in order to balance the pole. After 0.5 seconds the pole position changed to 10 degrees causing the cart to move back to the right. Because of this movement, the pole moved back towards the left, even though it reaches 18 degrees at 0.8 second. The movement of the pole going left is best seen on the graph of the pole deflection. It can be seen that at 0.75 second the pole deflection is -3.0 cm. This means that the pole moved quickly towards the left. Finally, at 1.0 second the system stabilized.

Figure 6.7(i) shows the result of operating the system initially on the left end of the track. It can be seen that the controller brings the cart to the center of the track after 4.7 seconds without any difficulties in balancing the pole. Figure 6.8(i) shows the result of applying external forces to the pole. Here, at 1.3 seconds the pole was pushed towards the left and stabilized at 2.3 seconds. At 3.1 seconds the pole was again pushed towards the right direction and stabilized at 4.1 seconds. Figure 6.9(i) shows the result of elevating the right end of the track. Here, the graph shows that the cart moved towards the center of the track, keeping the pole balanced. It can be seen from the graphs that the deflection of the pole stays below 1.0 cm as soon as the system stabilizes. The figures also show a superposed vibration at the natural frequency of the pole/mass system.

Other results of operating the system at different conditions are shown in figure 6.10(i) to 6.16(i). This represents an improvement on the Quanzer controller. It should be emphasized that for all test cases presented the controller develoveped was able to control the system for infinite time.

Figure 6.6(i)

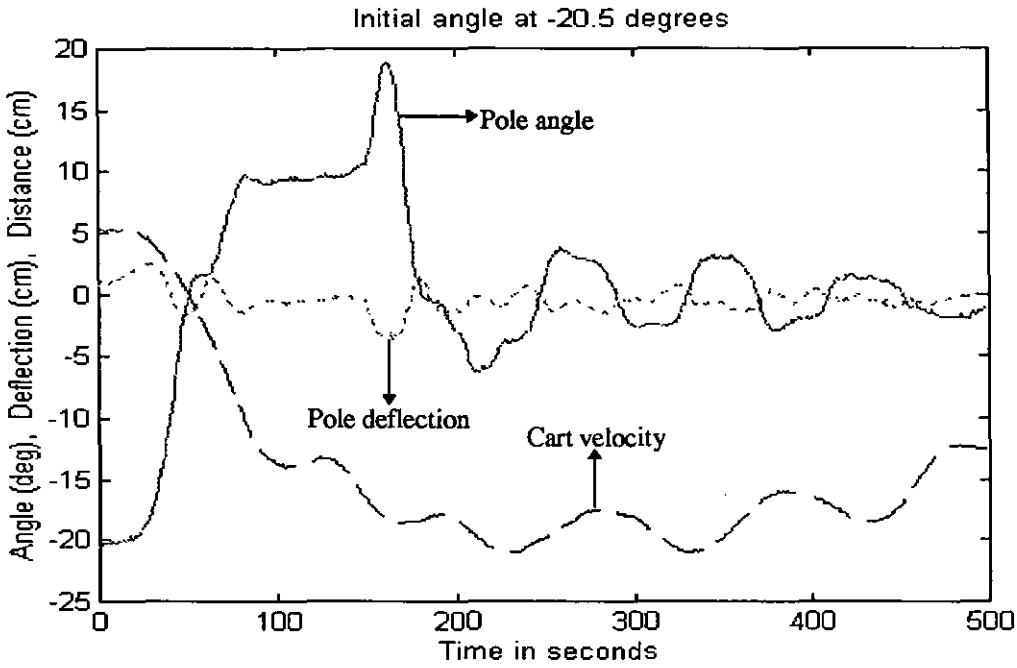


Figure 6.6(ii)

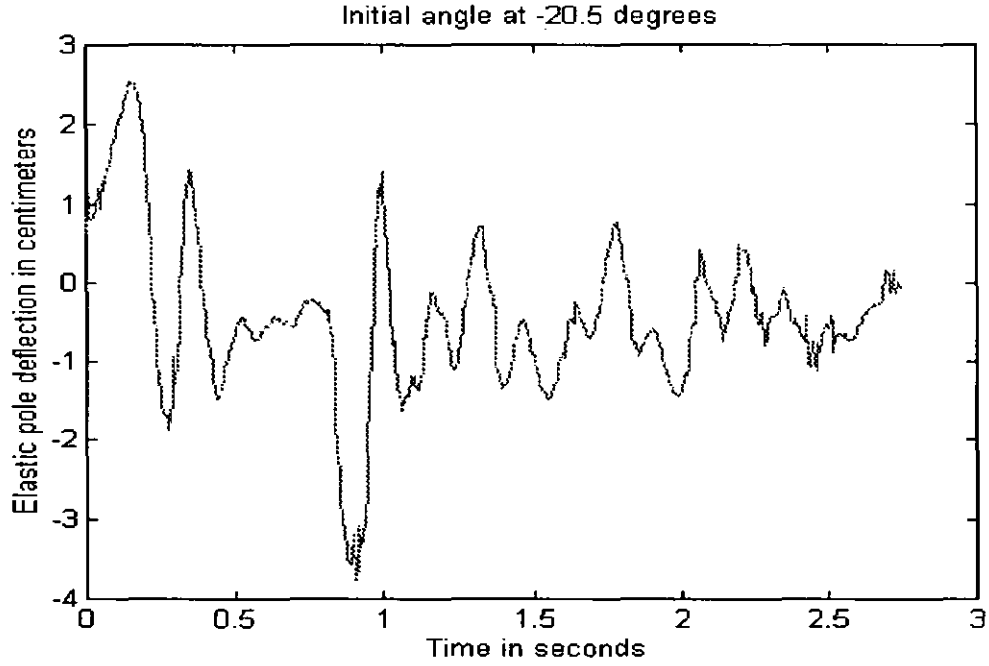


Figure 6.7(i)

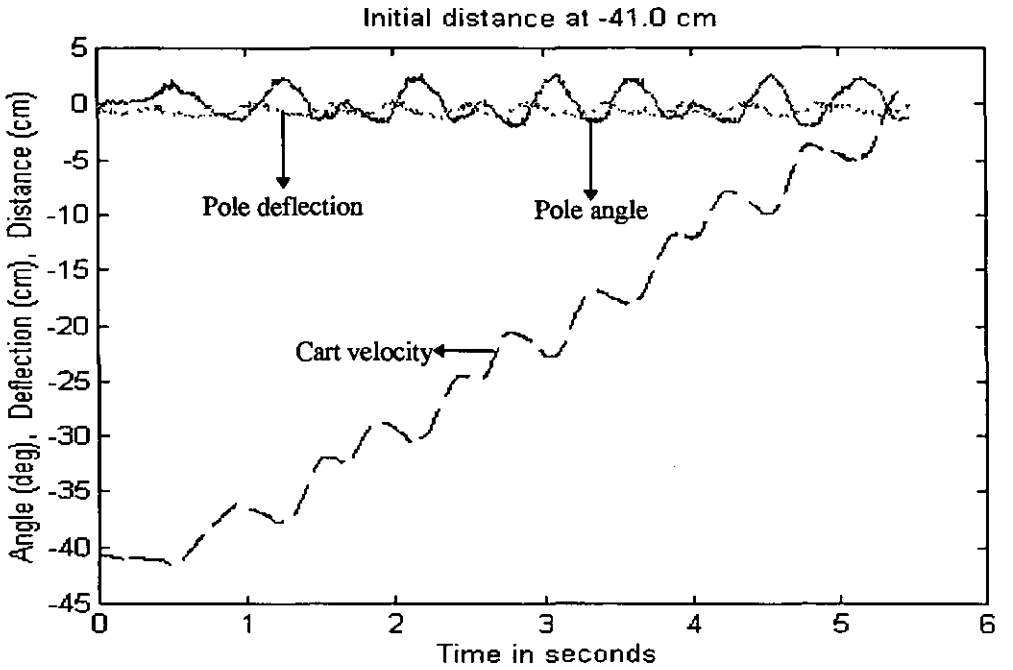


Figure 6.7(ii)

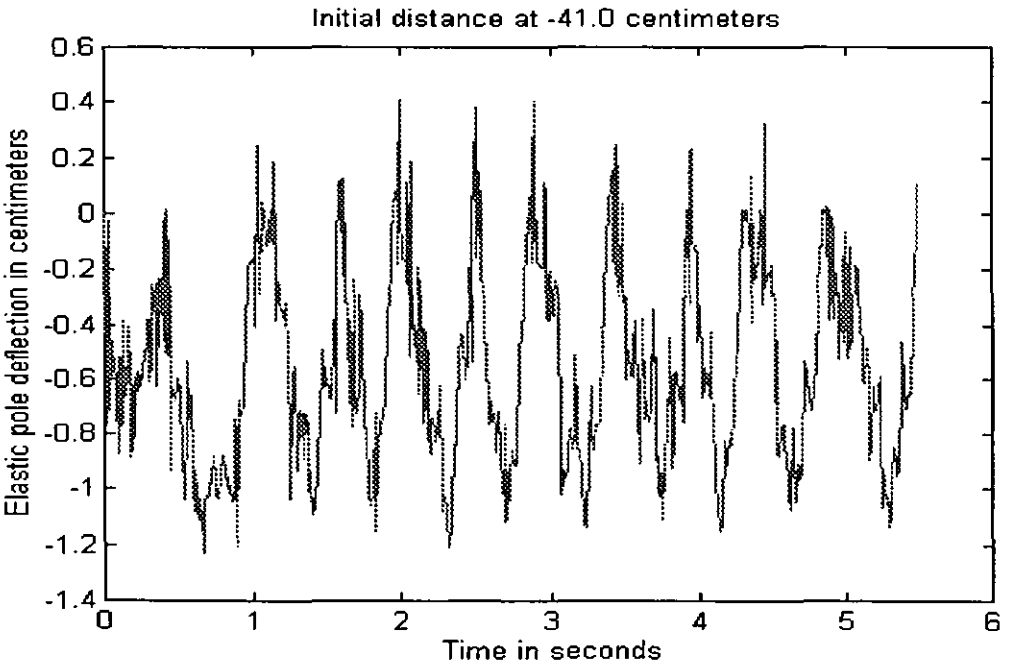


Figure 6.7(iii)

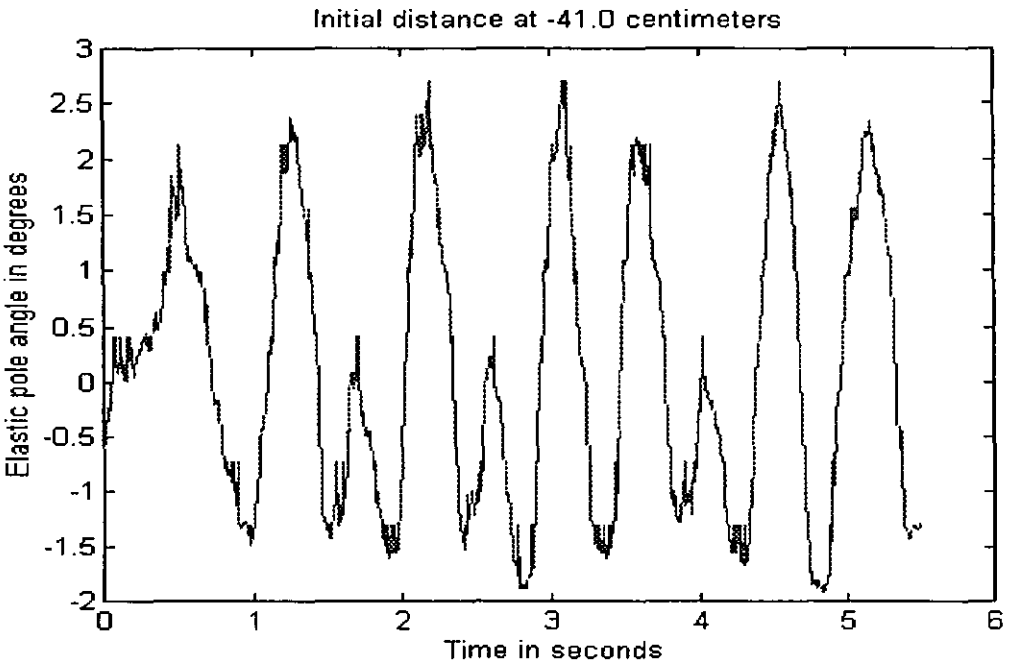


Figure 6.8(i)

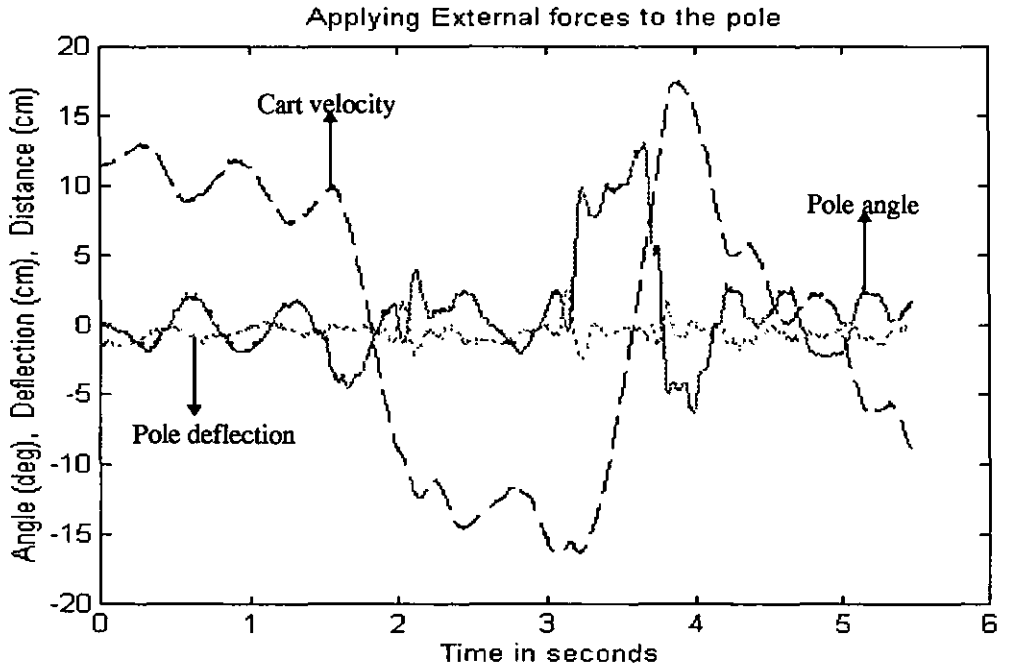


Figure 6.8(ii)

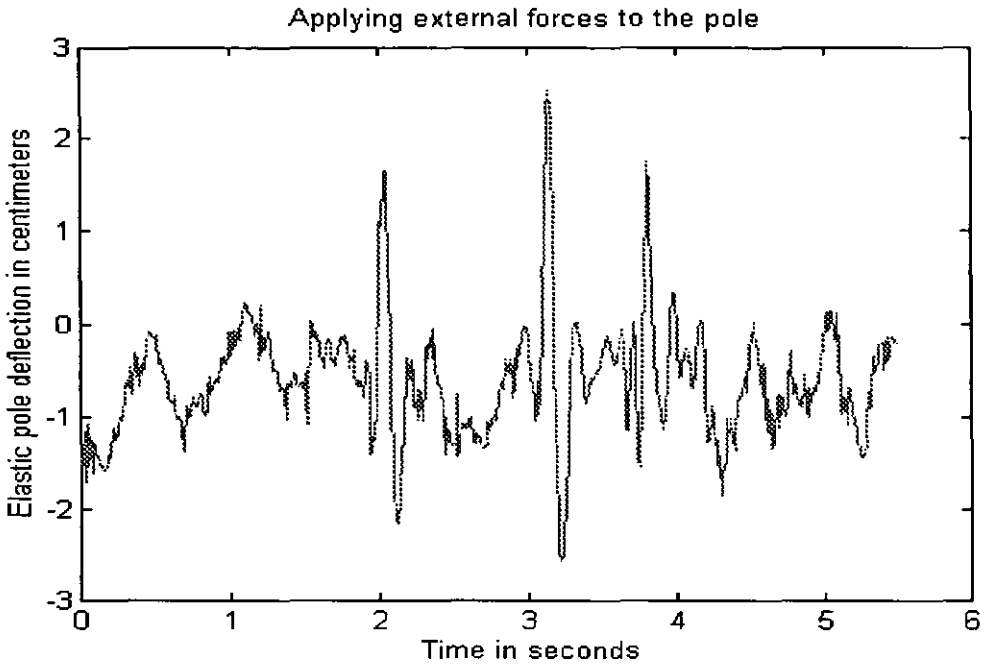


Figure 6.9(i)

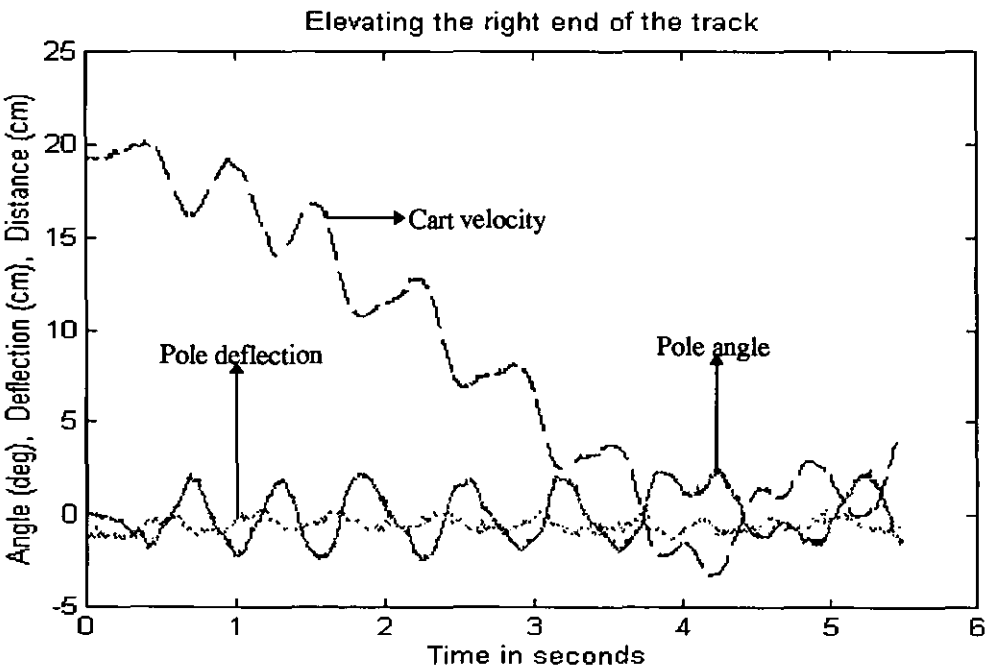


Figure 6.9(ii)

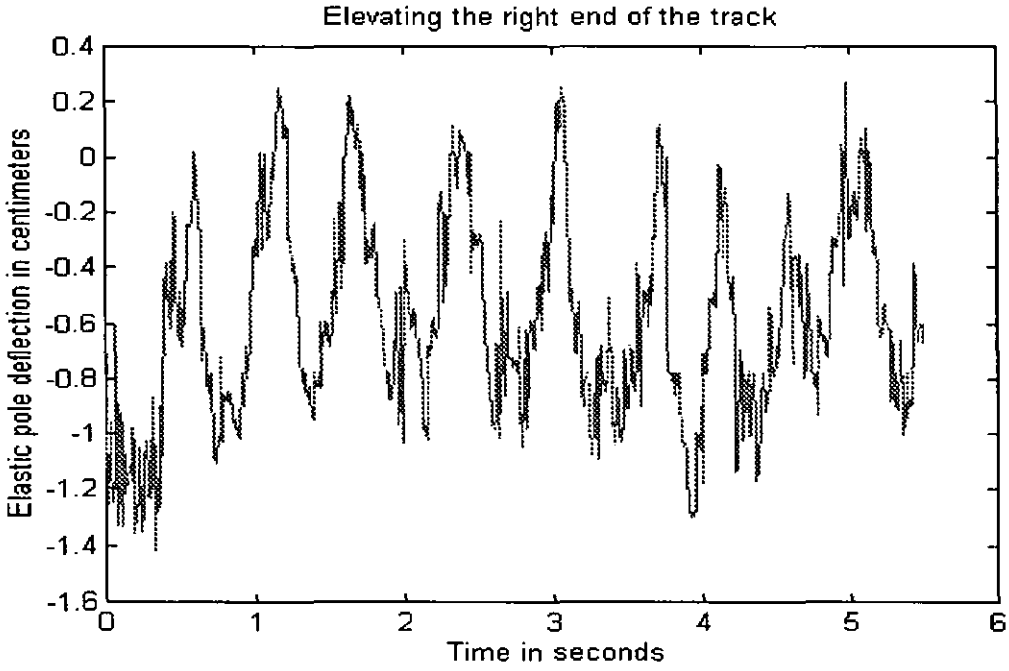


Figure 6.9(iii)

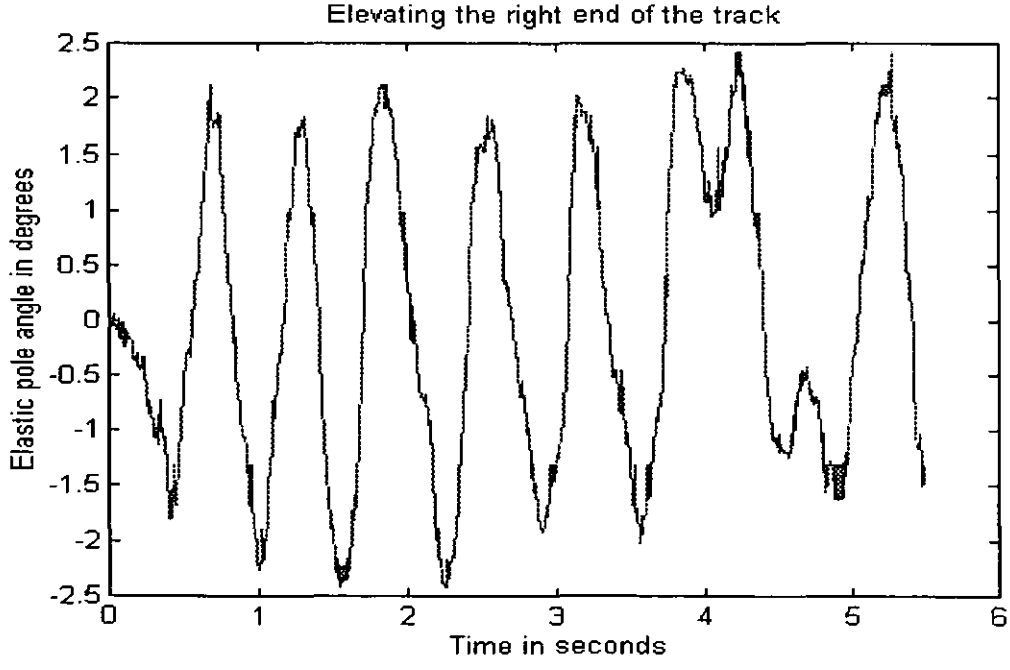


Figure 6.10(i)

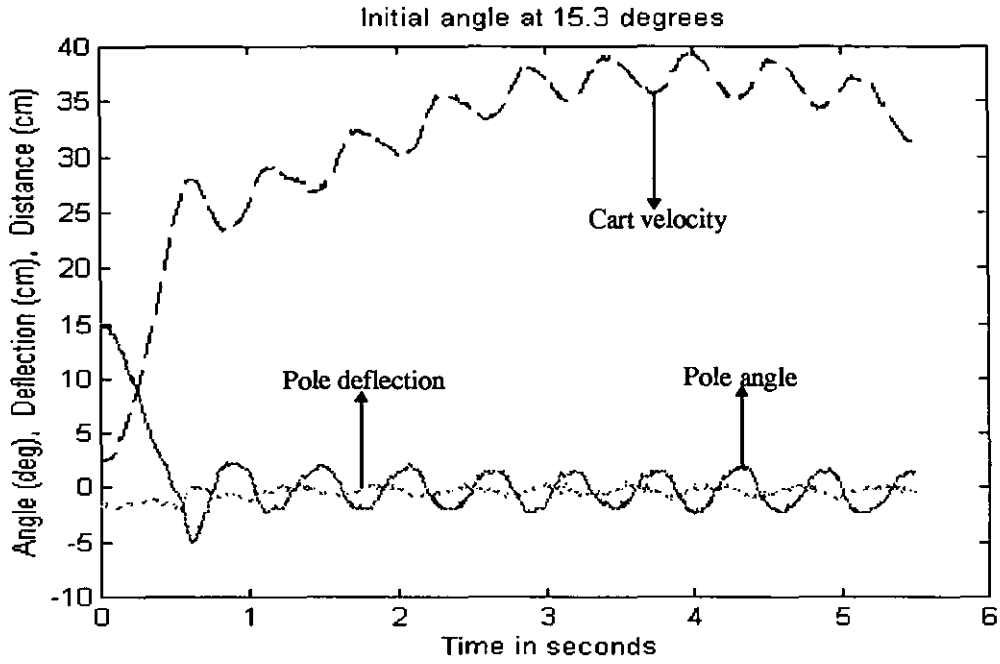


Figure 6.10(ii)

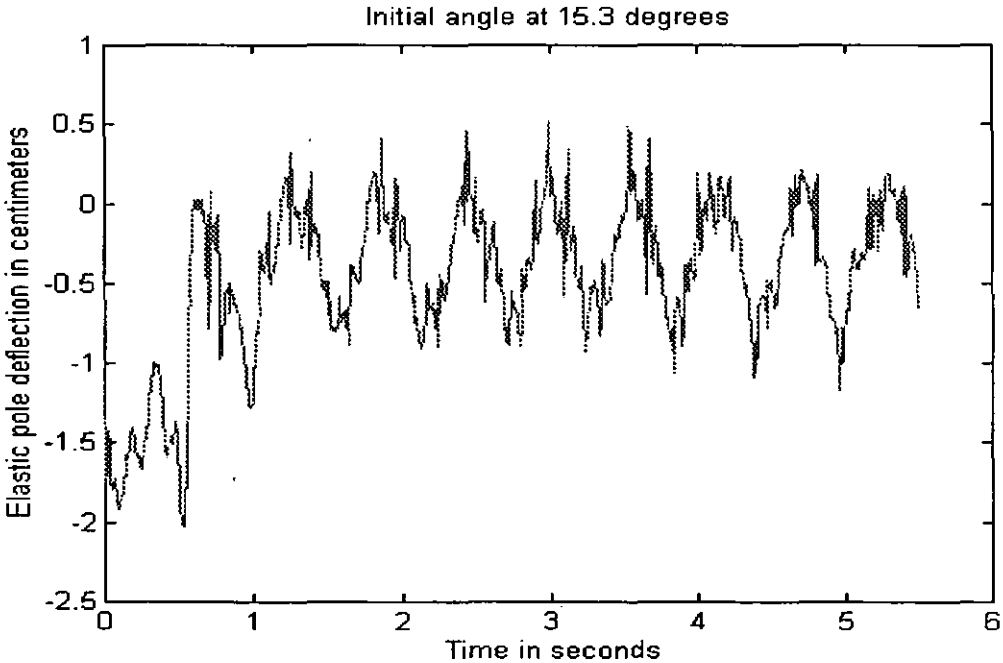


Figure 6.11(i)

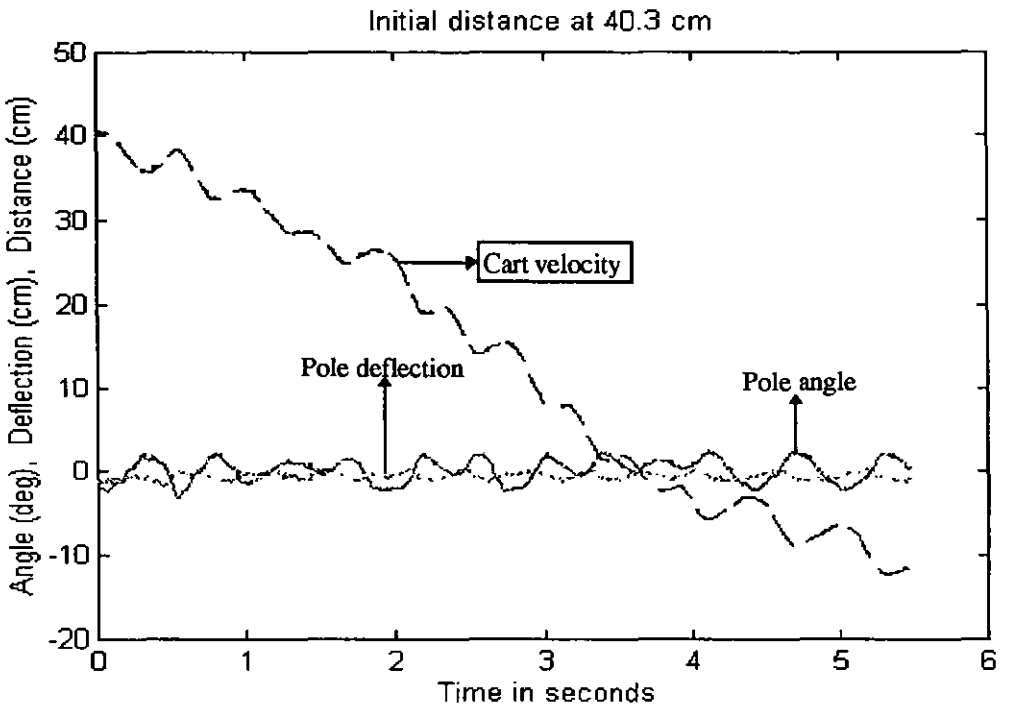


Figure 6.11(ii)

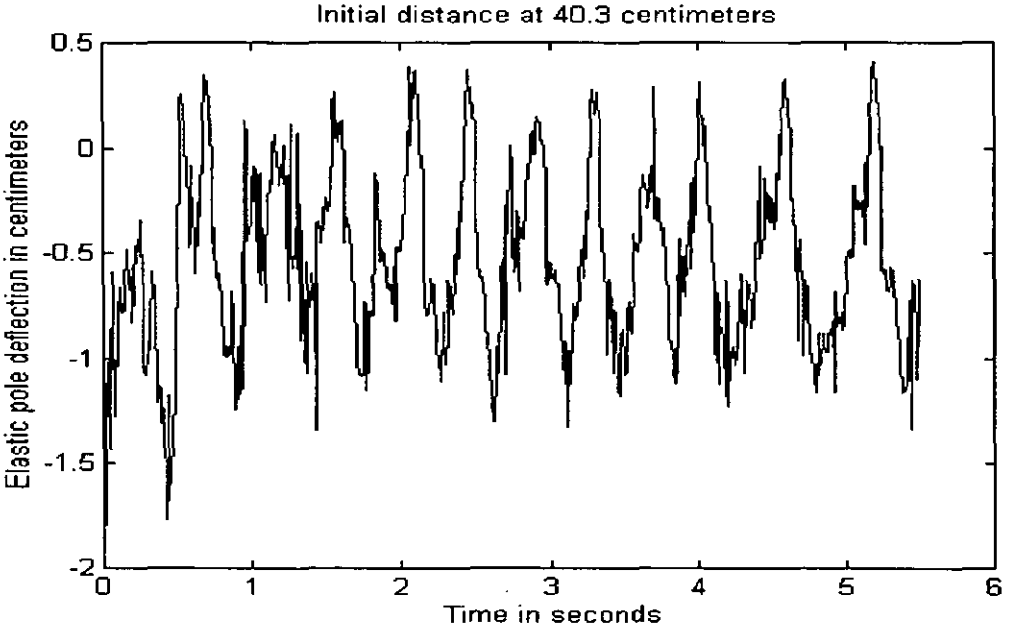


Figure 6.11(iii)

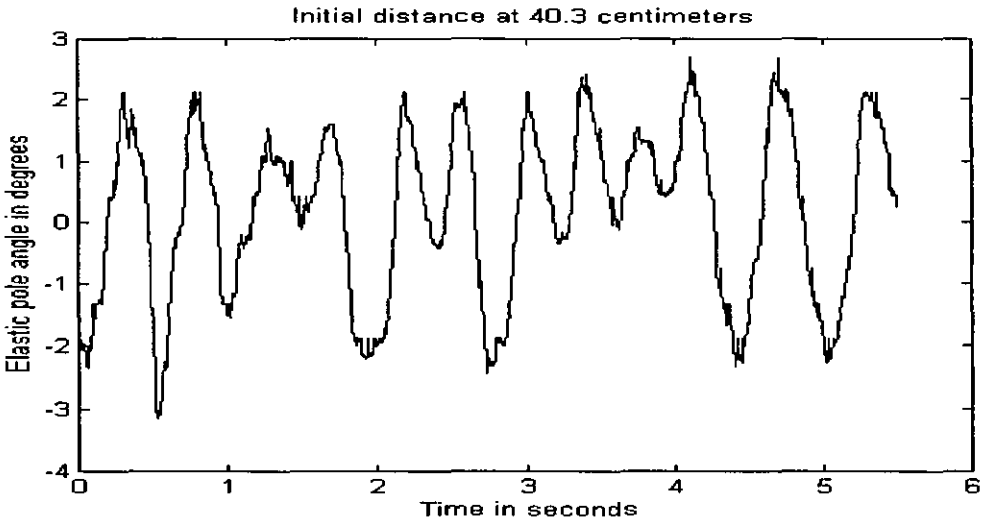


Figure 6.12(i)

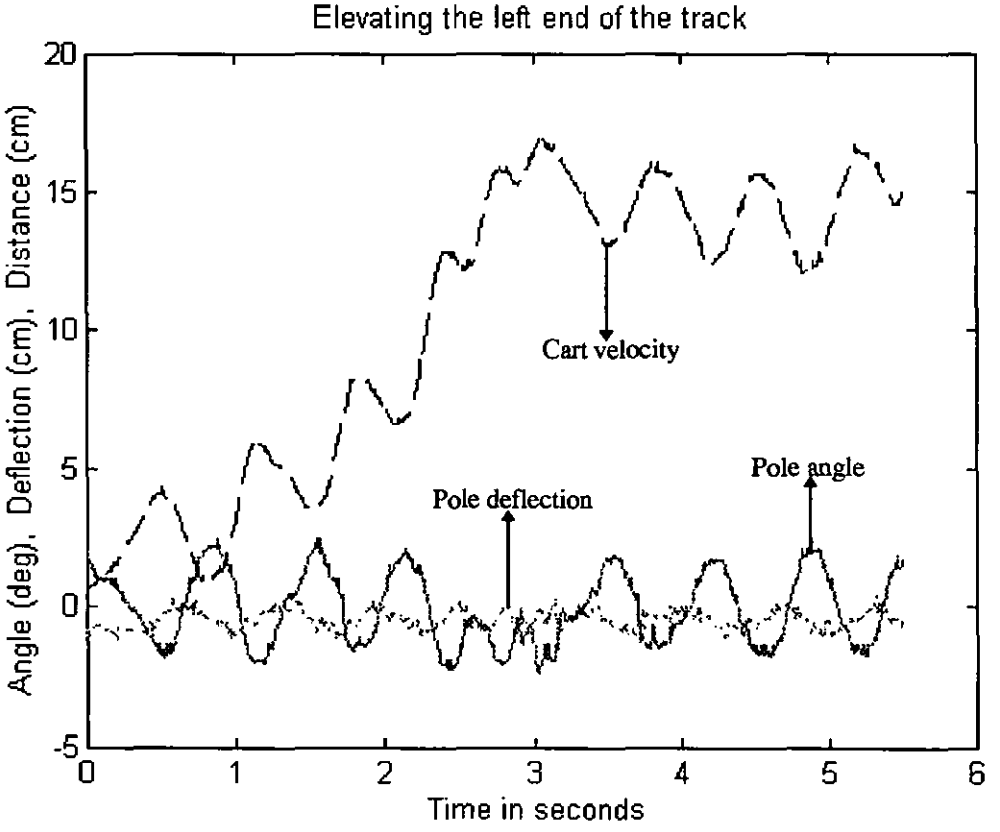


Figure 6.12(ii)

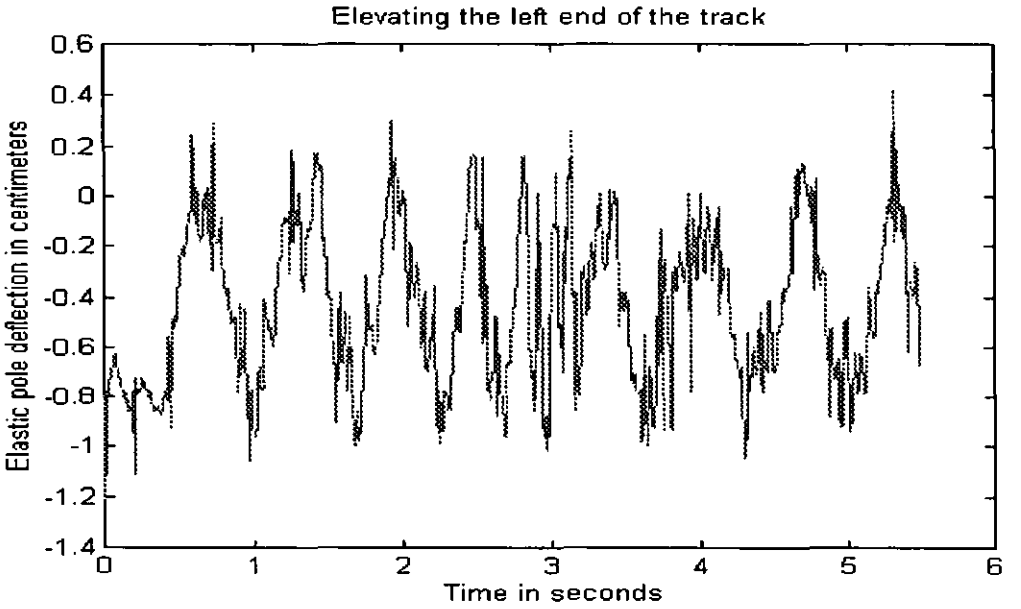


Figure 6.12(iii)

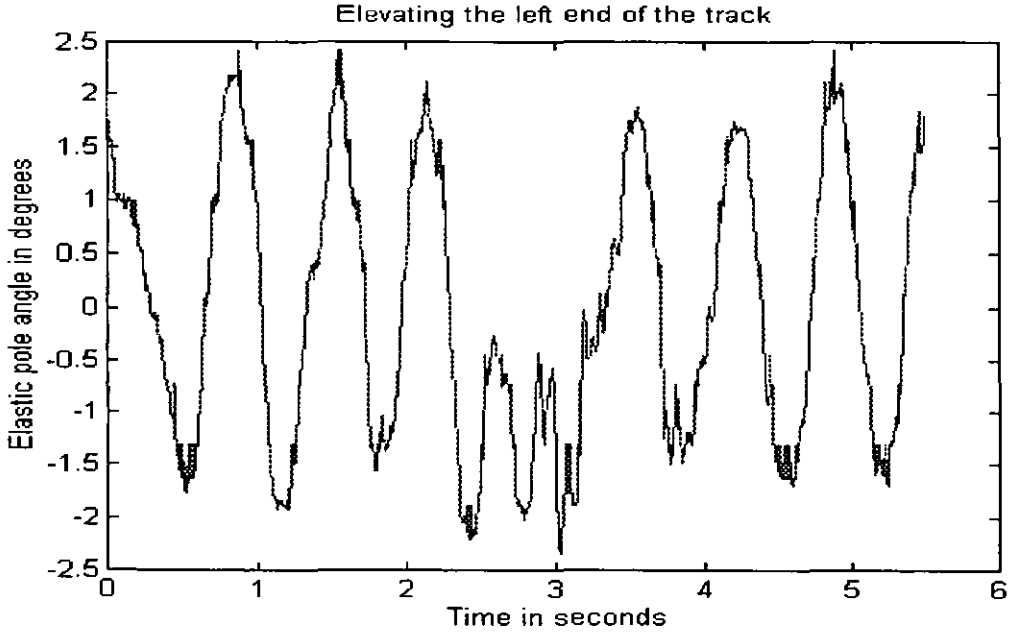


Figure 6.13(i)

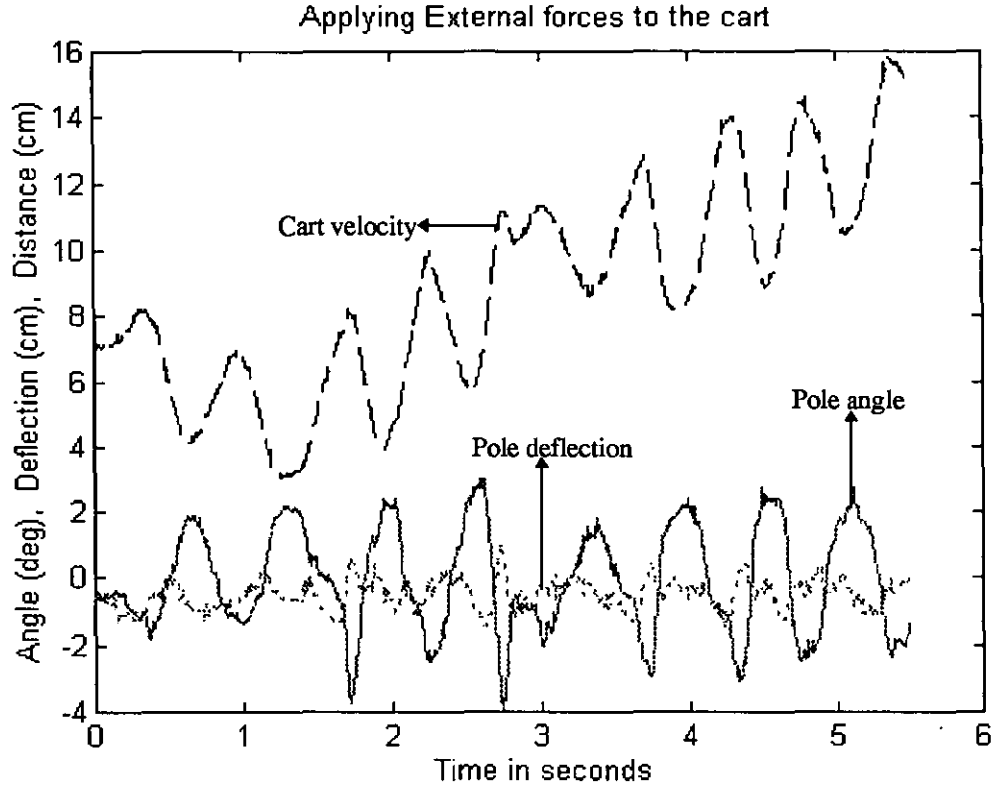


Figure 6.13(ii)

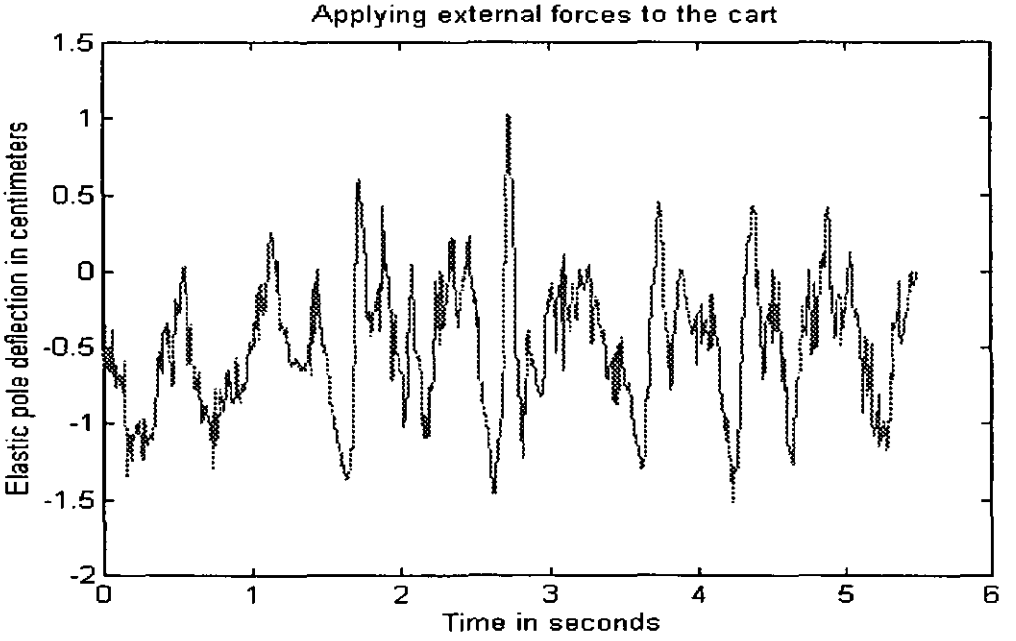


Figure 6.14(i)

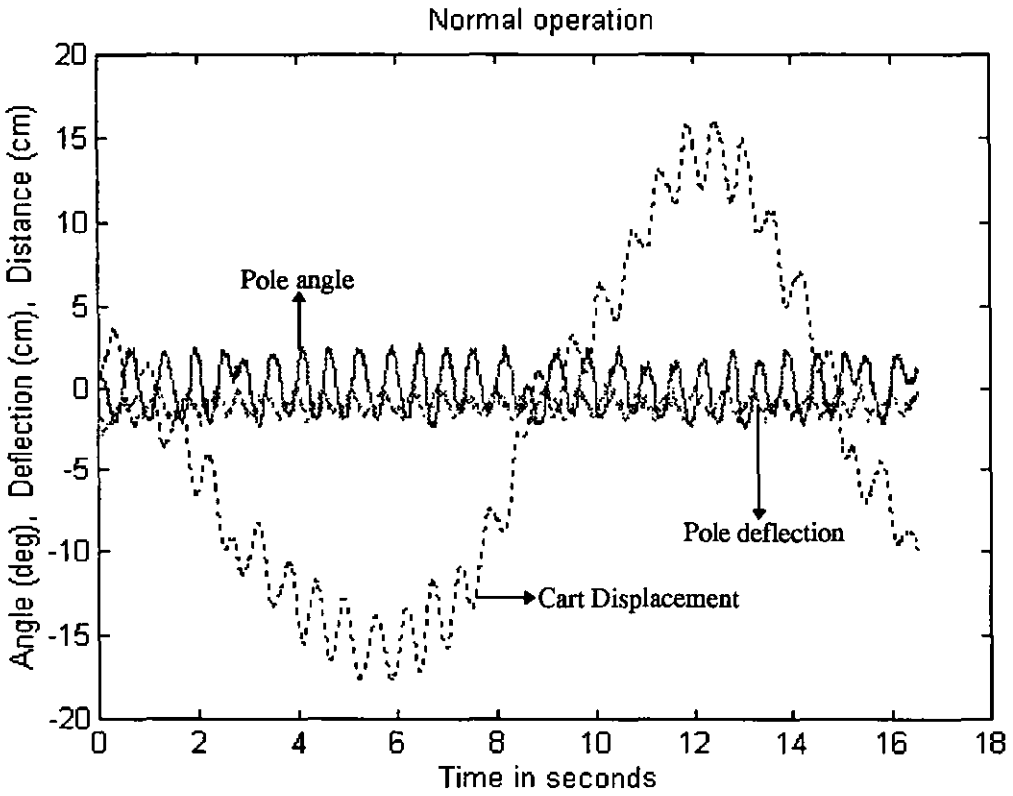


Figure 6.14(ii)

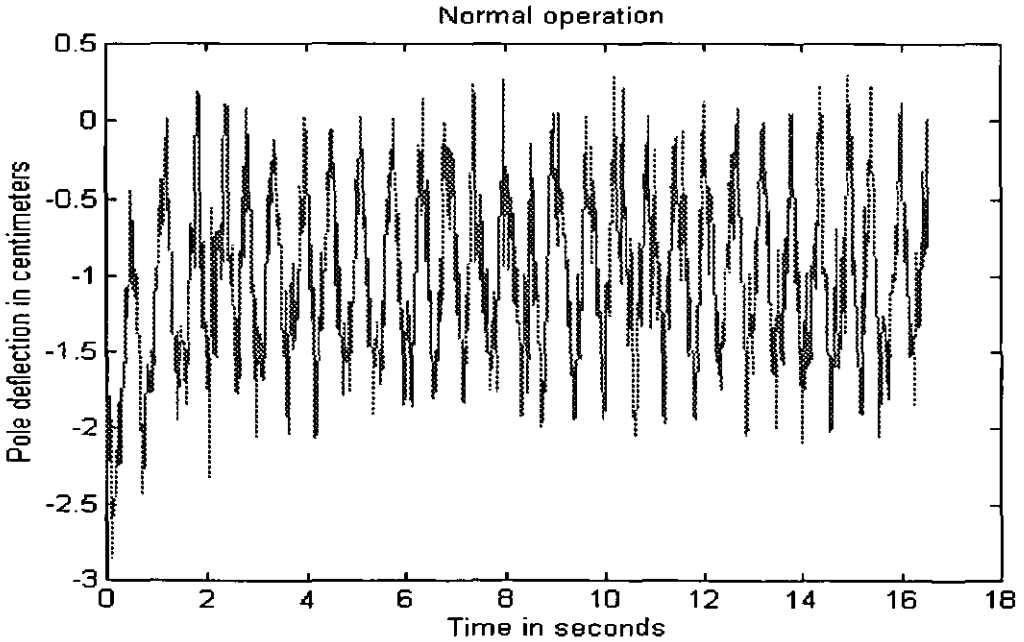


Figure 6.14(iii)

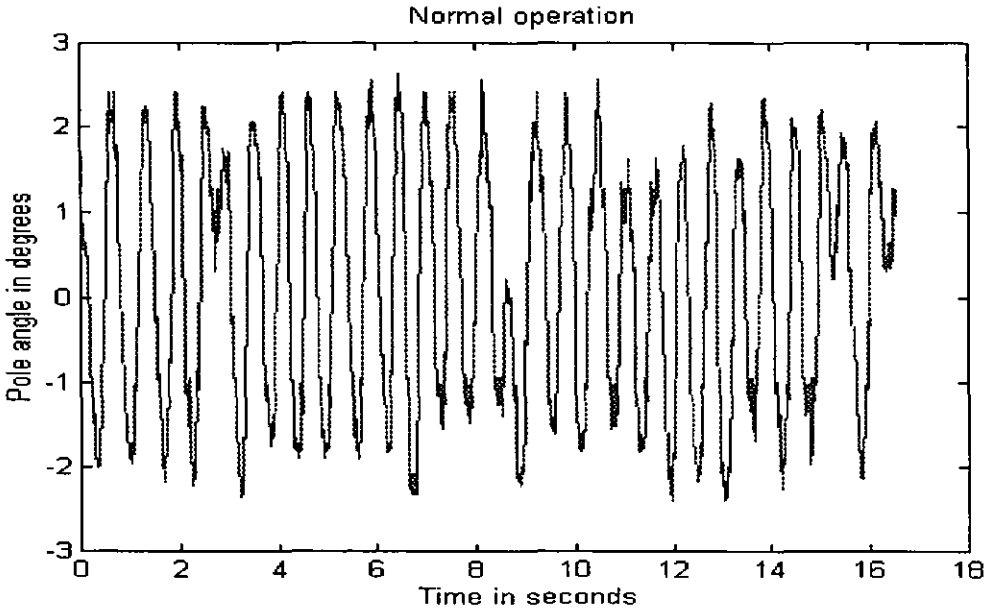


Figure 6.15(i)

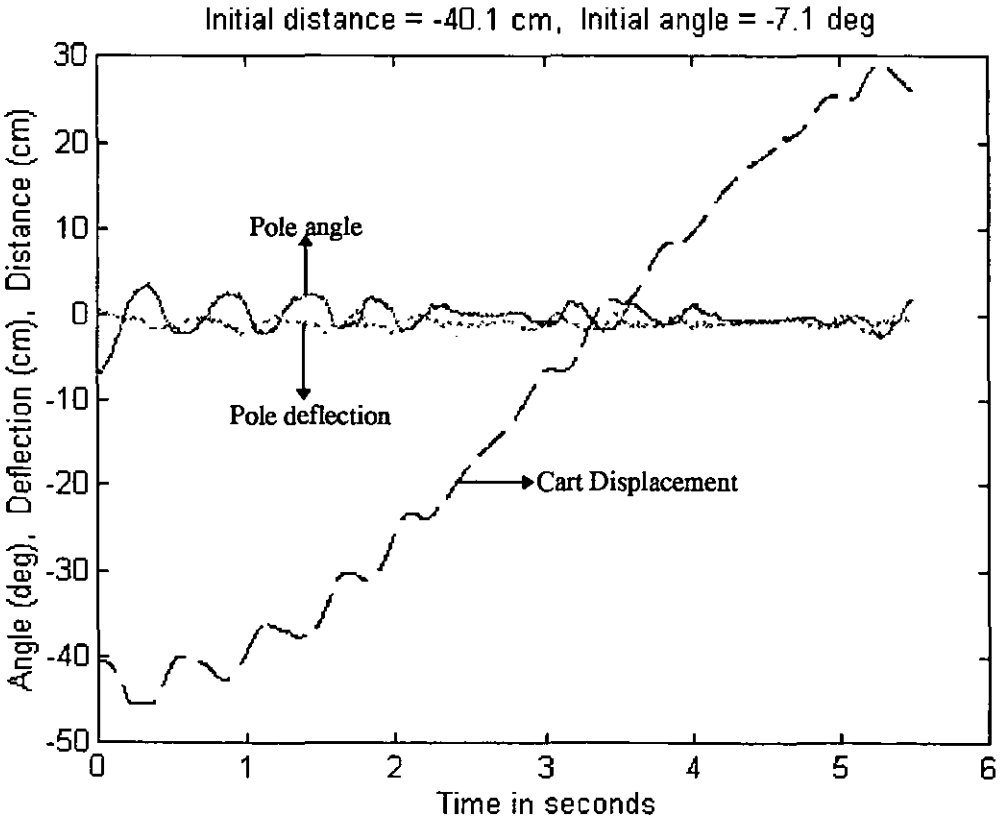


Figure 6.15(ii)

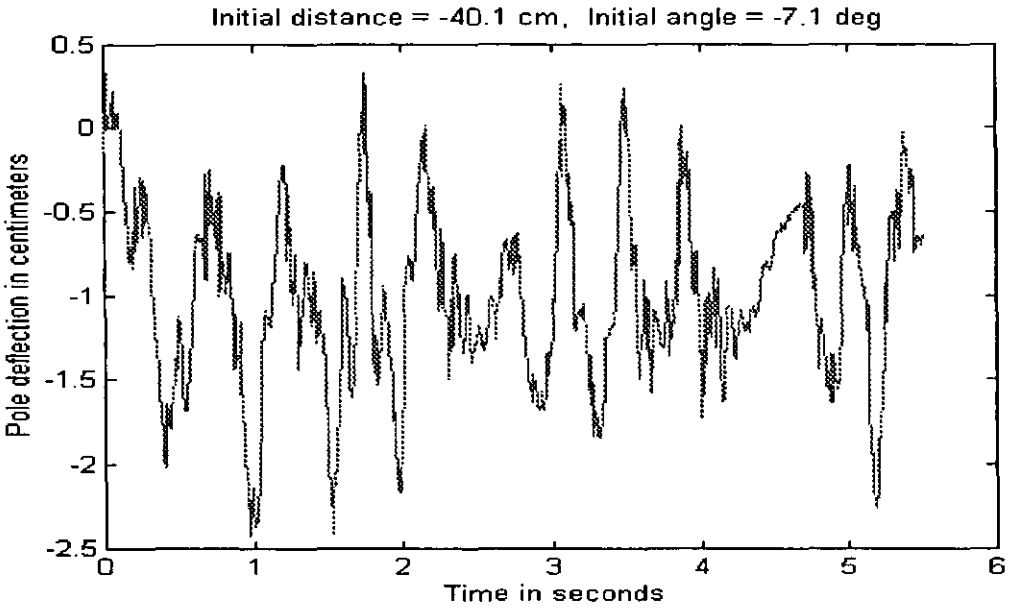


Figure 6.15(iii)

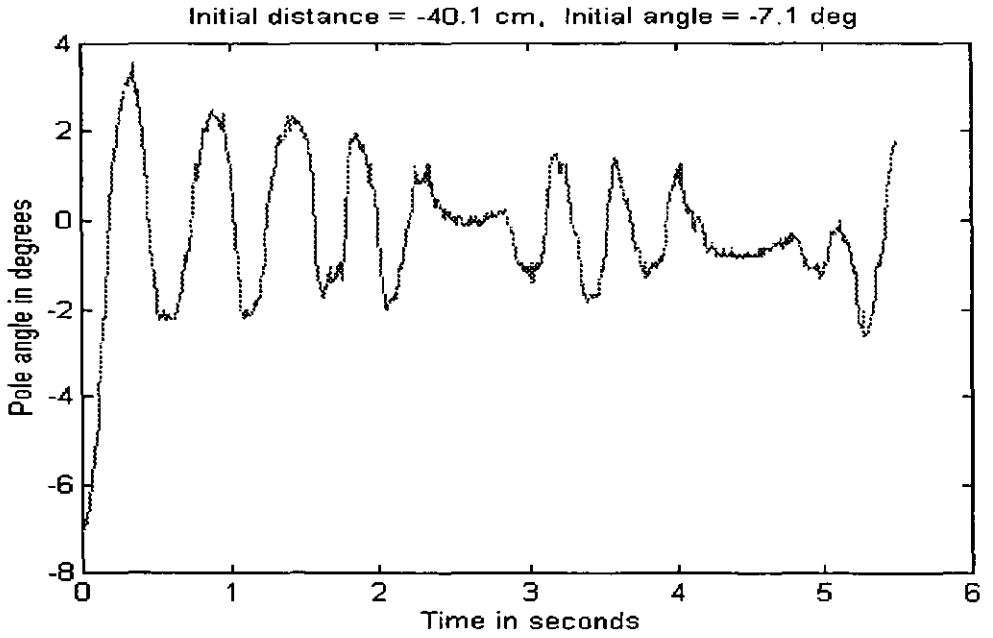


Figure 6.16(i)

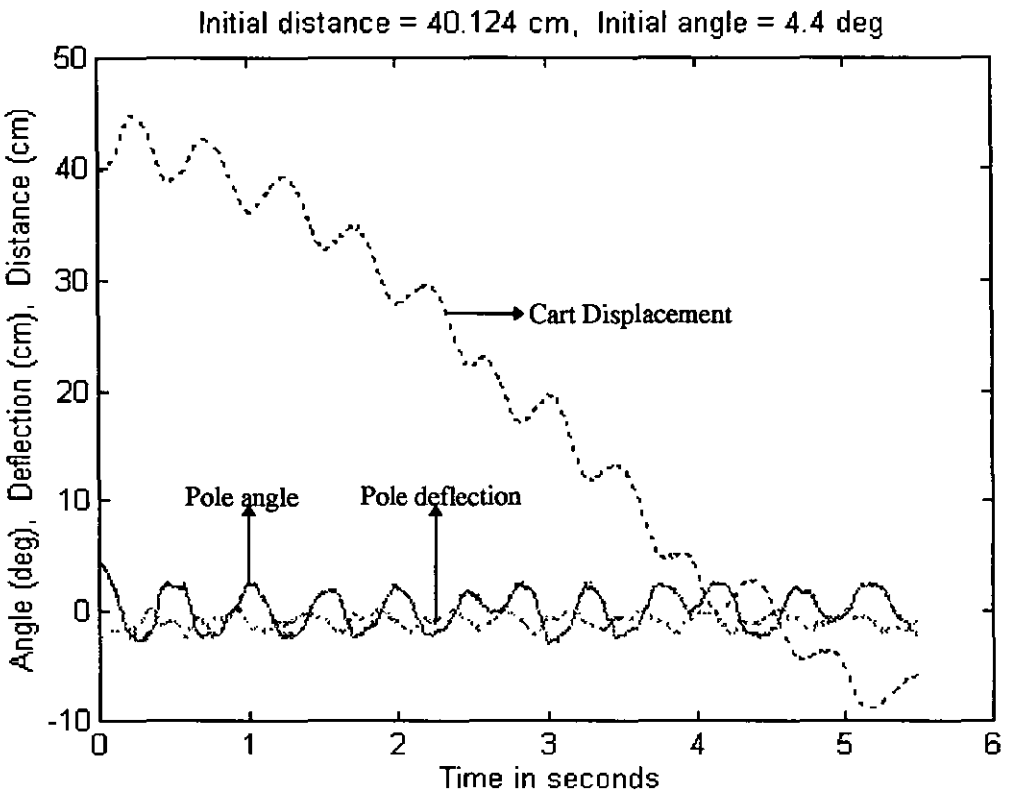
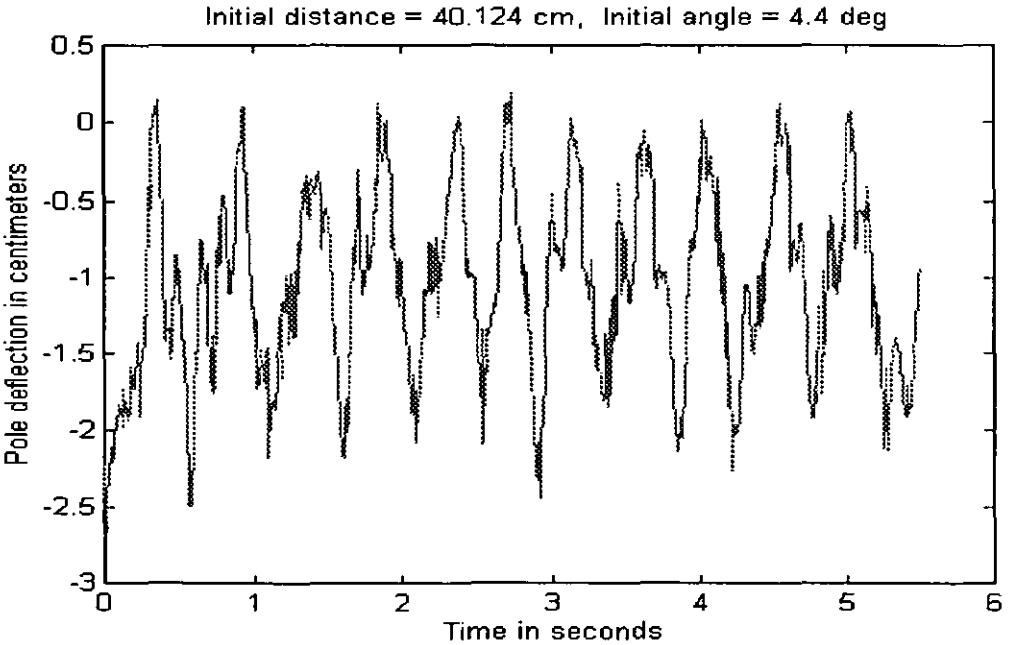


Figure 6.16(ii)



6.10. Summary

This chapter presented the applicability of fuzzy logic based algorithms to control a cart balancing a flexible pole. The controller design includes 5 fuzzy logic systems (FLS) and a single rule based evaluator that centres the pole on the track. Results of physical experiments show that the controller not only balances the flexible pole indefinitely but also brings the cart to the centre of the track. The controller can also easily adapt to disturbances from the external environment (e.g. moving or shaking the track randomly, elevating the height of the track on either side, pushing the pole in any direction, preventing the pole from moving further by putting an obstacle in its path). The operation of the system can also be initialised anywhere in the track. The controller is sufficiently fast to balance the system from an initial angle of 20 degrees.

The next chapter presents the application of a combination of genetic algorithm and fuzzy logic system techniques in the control of the flexible pole-cart balancing problem.

CHAPTER 7

A Fuzzy-Genetic Controller for the Flexible Pole-Cart Balancing Problem

7.1. Introduction

A fuzzy logic system is a nonlinear mapping of an input data vector into a scalar output, i.e. it maps numbers into numbers [56]. Such systems were introduced by Zadeh in 1965 [55]. Section 6.2 of this thesis explained the concepts and architectures of the fuzzy logic systems. The application of fuzzy logic to control problems was introduced by Mamdani in 1975 [58, 59]. Mamdani uses fuzzy logic to control the plant using fuzzy inference with rules preconstructed by an expert. Here, the most important task is to formulate the rule base which represents the experience and intuition of human experts. However, when a rule base from a human expert is not available, efficient control may not be possible [65]. Also, tuning the fuzzy logic membership functions requires the adjustment of many parameters simultaneously and is difficult to do manually. A probabilistic optimisation method utilising evolution strategies such as genetic algorithms can be employed to solve this problem.

Genetic algorithms are algorithms for optimisation and learning based on the mechanism of genetic evolution [63]. They were proposed by John Holland in 1975 and are a search procedure based on the mechanics of natural selection. A probabilistic component provides a means to search poorly understood, irregular spaces. This makes it likely that the system converges towards the global solution because it simultaneously evaluates many points in the parameter space. Genetic algorithms have been successfully applied to a variety of function optimisations, self-adaptive control systems, and learning systems [64].

The objective of this research is to test the capability of a genetic algorithm approach to formulate and optimise the parameters (i.e. fuzzy rules, membership

functions, and fuzzy associative memory matrix) necessary to implement a fuzzy logic controller to control the flexible pole-cart balancing problem (see section 3.2 for the dynamics of this problem). This minimises the role of a human expert in the design of a fuzzy logic controller.

In this chapter, the objective function (a function on which an optimisation algorithm operates seeking its maximum or minimum point) of the genetic algorithm used to obtain the fitness of each chromosome is calculated from the evaluation function of the fuzzy logic system. The data used to train the genetic emulation is taken from the results of a rule based controller acting, in simulation, on the derived dynamics of the flexible pole-cart balancing problem [72, 74] as described in chapter 3.

This chapter begins by the discussion of the concepts and architecture of a Genetic Algorithms and continues by the development of an off-line fuzzy-genetic controller as an application to the flexible pole-cart balancing problem. The results of the experiments conducted on this controller are presented.

7.2. Genetic Algorithms

Genetic Algorithms (GA's) are a biologically inspired class of algorithms that can be applied to, among other things, the optimisation of nonlinear multimodal (many local maxima and minima) functions [60]. They solve problems in the same way that nature solves the problem of adapting living organisms to the harsh realities of life in a hostile world.

The major concepts of the genetic algorithm are that of chromosomes, and the operations of mutation and reproduction. The GA maintains a set of trial solutions called chromosomes and forces them to evolve towards an acceptable solution. The algorithm uses survival of the fittest as well as the knowledge of the previous gene pool to improve each generation's ability to solve the problem [67]. A chromosome is constructed by stringing binary representations of vector components end to end (see figure 7.1). This

represents an encoding of information upon which the algorithm operates. The length of a chromosome depends on the vector dimension and the desired accuracy.

In order to obtain acceptable problem solutions the operation of reproduction and mutation is applied. A reproduction (or crossover) is a form of mating which combines two chromosomes to produce two new chromosomes (see figure 7.2). It is in this process that the GA can exploit the knowledge of the gene pool by allowing good chromosomes to combine with chromosomes that are not as good. This is based on the assumption that each individual, no matter how good it is does not have the answer to the problem [67]. It randomly selects a site along the length of the chromosome, and then splits the two chromosomes into two pieces. The new chromosomes are then formed by matching the top piece of one chromosome with the bottom of the other. The process of mutation will randomly change the bit value of the chromosomes (see figure 7.3). This is based on the idea that while each generation is better than the previous, the individuals that provide no offspring might have some information that is essential to the solution. Also, this will reinject information to the population because it might be that the initial population did not get all the necessary information.

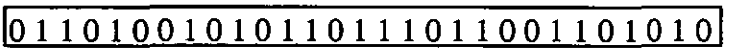


Figure 7.1
A chromosome - consists of a string of bits

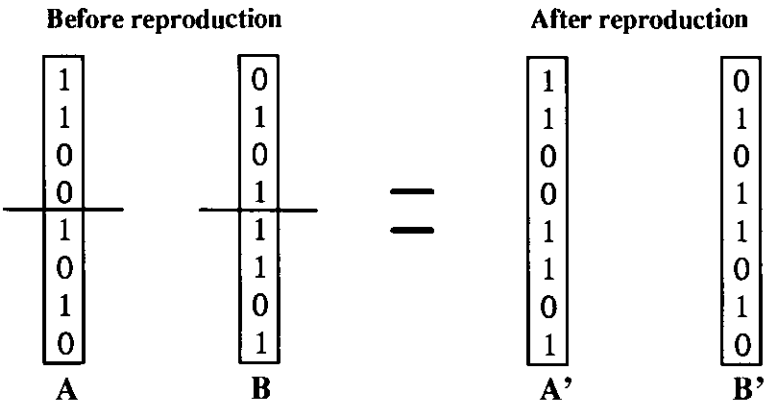


Figure 7.2
Reproduction or crossover operation of 2 chromosomes

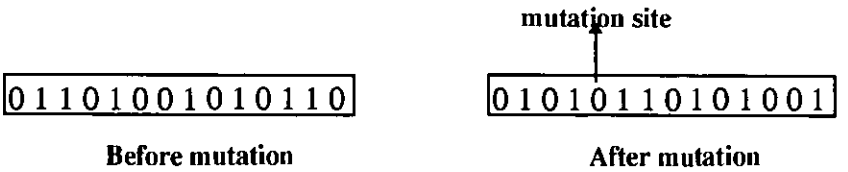


Figure 7.3
Mutation operation of a chromosome

7.3. Application of Fuzzy-Genetic Controller to the Flexible Pole-Cart Balancing System

The main objective of this research is to develop a hybrid genetic fuzzy logic controller that can predict the value of the force applied to the cart at any given time in order to balance a flexible pole hinged at its root on the top of the cart without the knowledge of a human expert. As mentioned earlier formulating the parameters (fuzzy rules and membership functions) for the fuzzy logic controller is extremely difficult without the knowledge of an expert. In this work a genetic algorithm is used to formulate these parameters by training the system using a training data set taken from the results of the rule base controller simulating the derived dynamics of the flexible pole cart balancing problem. This is equivalent to, for example, copying system data that is obtained from a system under human control. Figure 7.4 shows the block diagram of the entire process. The genetic algorithm determines its chromosomes fitness through the objective function calculated from the fuzzy logic system evaluation function. The trained information consists of the parameters needed by the fuzzy logic system (i.e., fuzzy rules, membership functions, input regions, etc.). A comparison of the result of the fuzzy-genetic controller and the rule based controller is shown in section 7.4. The Genetic Algorithm source code is shown in Appendix E.

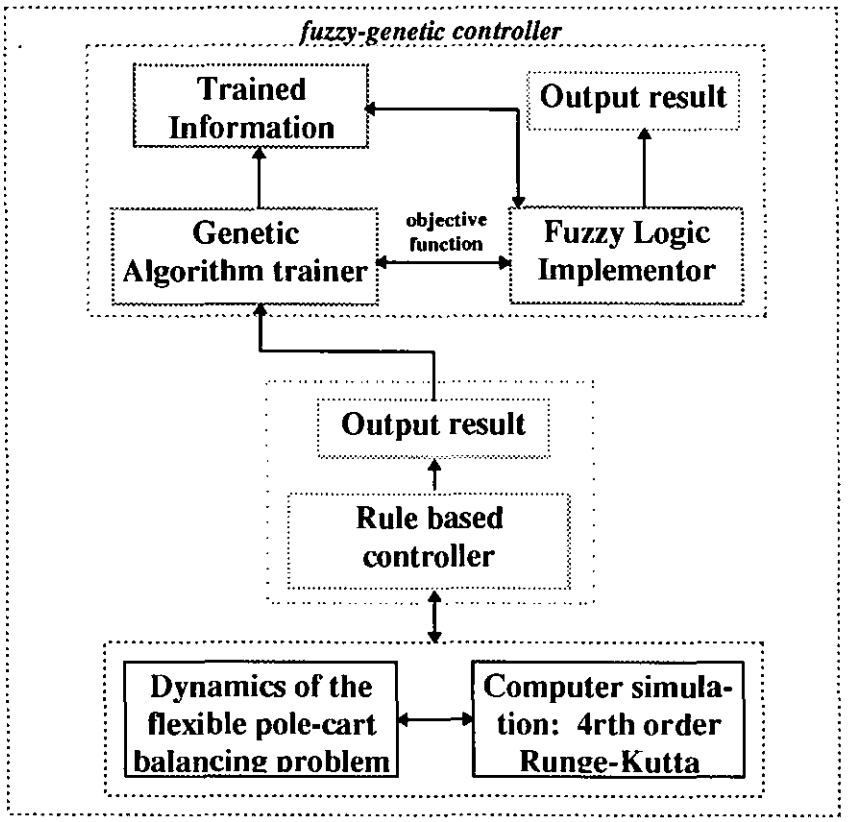


Figure 7.4
Hybrid controller block diagram for the flexible pole-cart balancing problem

7.3.1. Fuzzy-Genetic operation

In this controller there are two major processes involved: training and implementing the trained system. Training is accomplished by running the GA operating on the fuzzy system evaluation function. The function to be optimised by the GA is called the evaluation function. Here, the goal is to minimise the error function, that is, the accumulation over the training set of the absolute difference between the fuzzy system output and the desired output value. The GA seeks to maximise its evaluation function, it is therefore necessary to turn the error function upside down by subtracting it from a fixed constant that is larger than the largest error [60]. Here, the fixed constant value is equal to 2, and in order to insure that the maximum value of the accumulated error should not exceed 1, this is multiplied by an inverse of the maximum cumulative error. Thus, the value of the evaluation function is assigned to be between 1 and 2, with the 2 corresponding to minimum error. Every time an optimum value is attained, the fuzzy system parameters are saved to a file.

During the training process, GA's operate in cycles called generations, and a population of chromosomes is maintained. An individual chromosome is assigned a fitness value based on a problem-specific evaluation function [60]. Fitness values can be normalised, scaled, shared or left unchanged [68]. Maximum fitness is rewarded, the evaluation function must be chosen so that its maximum corresponds to the desired value of the function to be optimised. For every generation, pairs of individual chromosomes are chosen for the reproduction operation. The fitness of an individual determines the likelihood that it will be selected for this operation. During the reproduction operation mutation is randomly applied and a new population is determined. Over the course of a number of generations, the average fitness of the population increases, and the fittest individuals approach acceptable solutions to the application problem.

The implementation of the fuzzy logic system controller is straightforward, following the determination of the relevant parameters. These parameters are passed to the fuzzy system described in full in chapter 6 and in [74]. Thus, the parameters of the trained

fuzzy system are taken from a file, this is then interfaced with the training file and normalisation procedures. Since the system is now trained the GA is not needed further.

7.3.2. Adaptation of the Fuzzy Logic System Parameters Using Genetic Optimisation

The most important parameter in determining the fuzzy logic system output is representing and manipulating the fuzzy set of rules. The Fuzzy Associative Memory (FAM) matrix is responsible for doing this, hence, we will focus much attention on adapting the FAM. To apply genetic optimisation to FAM matrix adaptation, the matrix entries are strung together into a single long vector. Binary representation is used so that each matrix entry itself is a vector of 0's and 1's. This is the chromosome upon which the genetic algorithm operates.

In this system we use 4 input variables (the pole angle, the pole deflection, the cart displacement, and the cart velocity) and 1 output (the force applied to the cart) variable. We use a single FAM matrix that deals with all inputs simultaneously. Also, we use 3 input fuzzy sets for every input variable, hence the FAM matrix has $3^4 = 81$ entries. Take note that this number will grow very quickly if we increase either the number of input variables or the number of input fuzzy set for each variable. Since each FAM matrix entry is an output fuzzy set, we use 3-bit binary representation for this, giving 8 possible output fuzzy sets. As we mentioned earlier, these FAM matrix can be thought of as 0's and 1's, stringing these together end to end, we get a chromosome of length $3 \times 81 = 243$ for our genetic optimisation.

The number of fuzzy membership functions is equal to 3 since it is equivalent to the number of input fuzzy sets. The shape of the membership function is assumed to be trapezoidal, hence we have 4 points defining each membership function. However, assuming that each trapezium penetrates each other, then adjacent trapeziums share two of their defining points (subintervals), thus, each membership function, with the exception of the final right trapezoid, increases the total number of defining points by two (see figure

7.5). Hence, if N is the number of membership functions, then there are a total of 2(N-1) defining points [60].

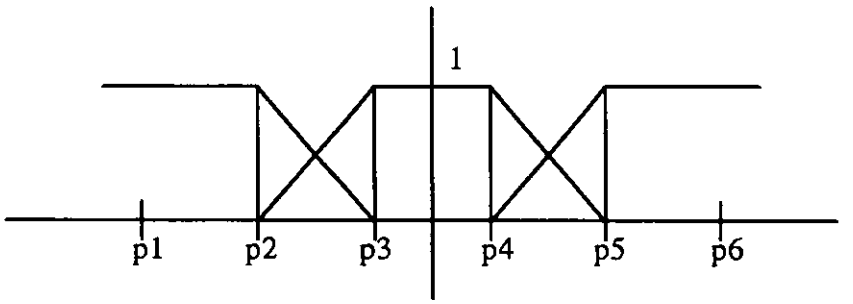


Figure 7.5
Defining points for 3 membership functions
(There are 5 equal length subintervals)

7.3.3. The GA Program

Figure 7.6 shows the step by step process of the genetic algorithm program for the flexible pole cart balancing problem. Refer to Appendix E for the source code. Below are the parameters used in this program

- Population size = 54

  Typical population sizes range from 40 to 100 individuals. Longer chromosomes may require larger population.

- Mutation probability = 1/population size = 0.0185

  This is the probability of mutation for a single bit during crossover operation.

- Reproduction/crossover probability = 0.6

  This is the probability to determine if reproduction operation is to be applied to selected pair. If reproduction is not applied, the pair is sent on to the next generation unchanged except for possible mutation.

- Total number of generations = 150

  A generation is a step or a cycle of GA's operation. Pairs of chromosomes are chosen for the reproduction operation.

- Bit length = 3

  This determines numeric accuracy or, alternately, dynamic range when the problem involves optimisation over numeric vectors.

- Vector length = 81

  The number of components in each vector when optimisation is over a set of vectors.

- Chromosome length = (bit length)x(vector length) = 243

  Length of each chromosome.

- Maximum tolerance = 0.01

  An error tolerance use to terminate the generation. If an individual is found with a fitness within tolerance of maximum fitness, the algorithm stops.

- Maximum fitness = 2.0

The maximum fitness for each chromosome.

The program shown in figure 7.6 starts with the initialisation of variables needed in fuzzy logic system and genetic algorithm (see pages 301, 302, 303 and 310 appendix E). The training data is taken from the external file fl_ga.trn and normalised for GA's operation (see page 317 appendix E). The process then continue by starting the loop for the optimisation operation. The loop will terminate only when the prescribed number of generations or the error tolerance is attained. The optimisation process involves the following operations:

- Selection of pairs of chromosomes based on its fitness (see page 304 appendix E).
- Reproduction and mutation operation (see page 305 appendix E).
- Calculation of the chromosomes fitness using fuzzy logic evaluation function (these are the chromosomes for the new population; see page 323 appendix E).
- Obtained X fittest chromosomes from the pool of X new and X old population (these are the X chromosomes for the new population; see page 306 appendix E).
- Re-scale the fitness of the new X population (see page 305 appendix E).
- Finally, if new optimum value found, then store information to external file fl_ga.fzs. These are the parameters used for fuzzy logic controller.

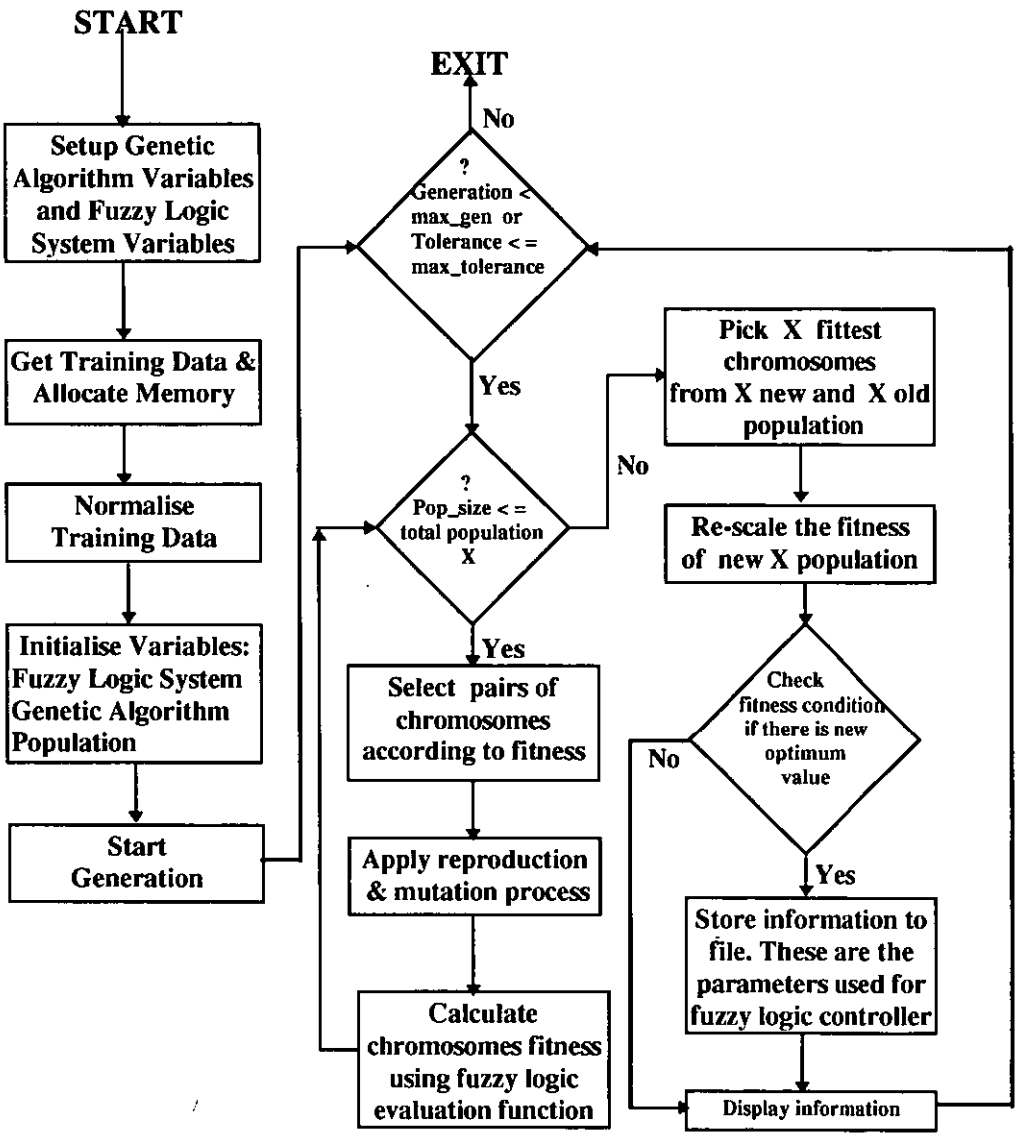


Figure 7.6
The genetic algorithm computer program

7.4. Discussion of Results

A comparison of the result of application of the fuzzy-genetic controller to the desired output for the flexible pole-cart balancing problem is shown in figure 7.7. The graph shows that the controller is able to predict the desired behaviour with a high accuracy. Slight errors appear at the maximum values of the output because this is the time when the cart abruptly changes its direction. The predicted error of the fuzzy-genetic model is shown in figure 7.8. The error is measured from the absolute difference of the actual output of the fuzzy logic implementor and the desired output behaviour. The convergence speed of the GA is shown in figure 7.9. It can be seen from this graph that it took only 200 seconds to get the optimum fitness of 1.9693. Figure 7.10 shows the graph of the chromosomes fitness versus the number of generations. Here, the maximum fitness had been attained in less than 50 generations.

As the GA progresses, the average fitness of the population increases. Since there is an upper bound of fitness, this has the effect of compressing the range of fitness values for the population (e.g. the difference between the largest and smallest fitness values shrinks). In order to solve this problem the concept of scaling was introduced. The scaling simply magnifies the range of fitness values in a linear way [60]. For example, suppose the algorithm reaches a point where all of the fitness values are between 1.935 and 1.945 (with an optimal maximum value of 2.0). For purposes of reproduction selection, scaling would change the upper value 1.945 to 1.0, and the lower value 1.935 to 0.05. The effect of this is that the individual chromosome with a fitness value 1.0 has a better chance of selection for the next reproduction than the individual whose fitness value is nearly equal to 0.0.

In order to improve the convergence speed, the GA uses a survival of the fittest approach by picking the fittest individual from the old and new population to participate in the next generation. For example, if the standard size of the population is P, mutation and reproduction operations will produce a new population of size P, then the old and the new population can be combined together, producing a superpopulation of size 2P. The GA

then picked the P fittest individuals from this superpopulation for the next generation's population. This technique proved to be very effective on the training process.

Figure 7.7

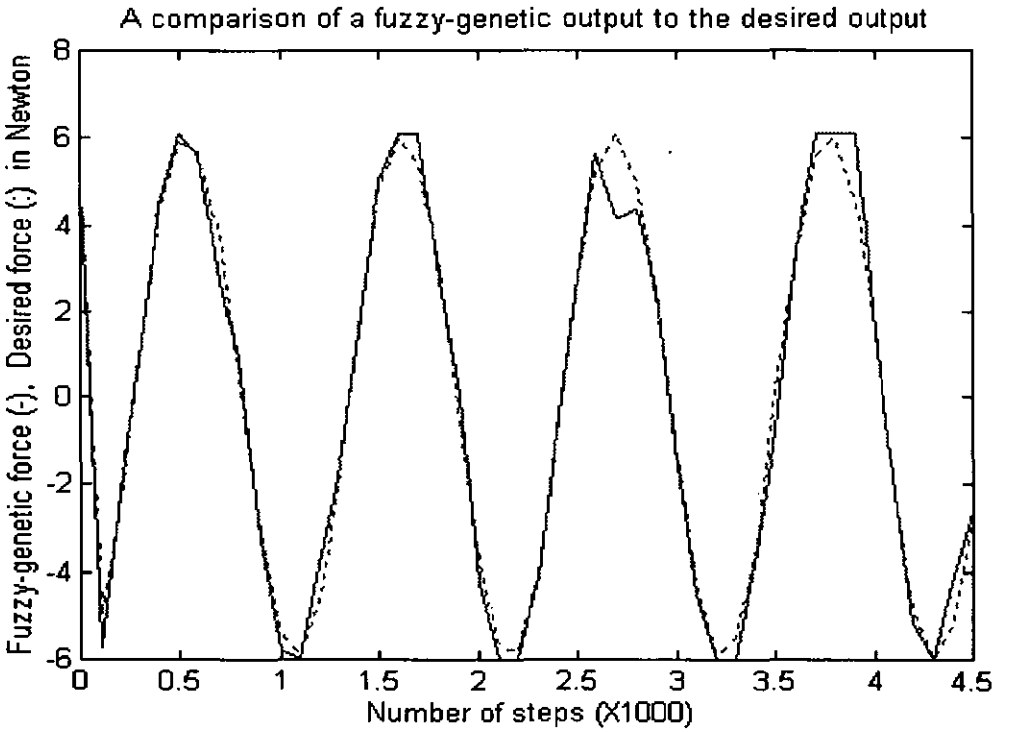


Figure 7.8

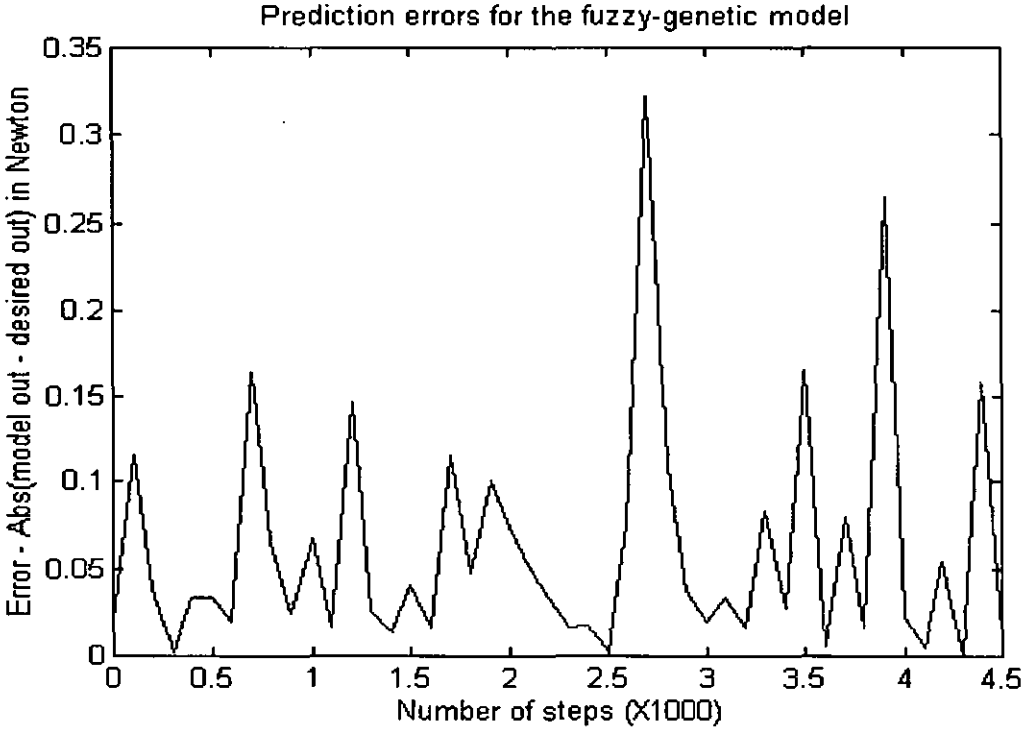


Figure 7.9

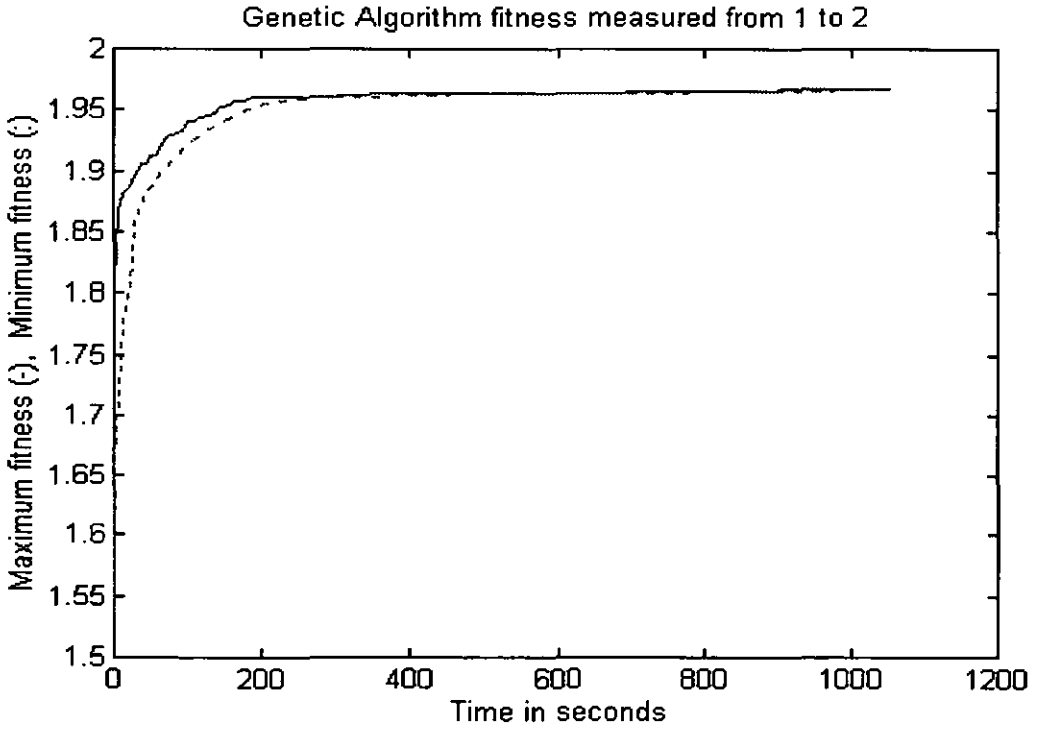
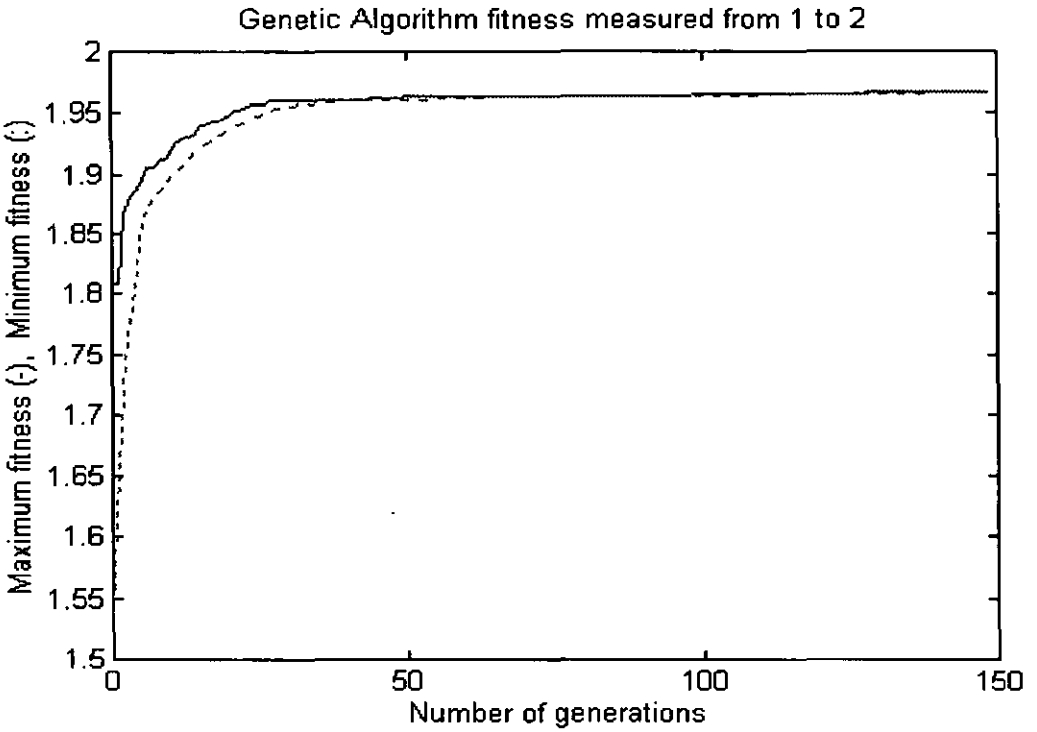


Figure 7.10



7.5. Summary

This chapter investigates the applicability of developing a controller based on genetic algorithms combined with fuzzy logic to control the flexible pole-cart balancing problem. The genetic algorithm is used to obtain the values of the variables required by the fuzzy logic controller, removing the need for expert knowledge. The system employs genetic search to extract the fuzzy rules and membership functions using an objective function calculated from the fuzzy logic system evaluation function. The extracted rules are used in the fuzzy associative memory matrix entries so that the fuzzy logic system performance fits the desired behaviour. Results show that the controller developed is able to predict the desired output for the flexible pole-cart balancing problem with high accuracy.

CHAPTER 8

Conclusions

8.1. The Inverted Pendulum Problem

For the past decade many researchers have used the inverted pendulum problem as a model for the development of learning control systems [3,4,5,6,9,10,11]. Many authors have successfully used neural network techniques to construct these control systems. Although it is true that the dynamics of the inverted pendulum is nonlinear, Geva and Sitte[10] discovered that it is easy to find, by simple random search in weight space, linear controllers that not only balance the pole but also centre the cart. With this discovery Geva and Sitte concluded that controlling an inverted pendulum is not as difficult a nonlinear problem as has been assumed by many authors.

Further the inverted pendulum system has only two degrees of freedom and therefore any learning controller demonstrated on this application is likely to have limited application in more demanding applications such as those encountered in manufacturing industries. Such a controller would be hard to use in a robot with flexible joints for example. Flexible robot manipulator research is active [29,30,31,12] because when compared with the traditional robot manipulators constructed by rigid links, the flexible robot manipulators not only are able to move larger payloads without increasing the mass of the linkages, but also have many other advantages: They require less material and smaller actuators, have less link weight, consume less power, and are more manoeuvrable and transportable [29]. They have not been widely used in production industries due in part to the fact that manipulators are required to have a reasonable accuracy in the response of the manipulator end-effector to the input commands from their control systems. Hence, if the advantages associated with lightweight are to be sacrificed, advanced control systems for flexible robot manipulators have to be developed [29]. The

flexible pole-cart problem described in this thesis both extends the complexity of the pole balancing test case and includes features of the flexible robot arm problem and therefore allows many relevant issues to be explored.

8.2. The Mathematical Model of the Flexible Pole-Cart Balancing Problem

To design advanced controllers for a cart balancing an elastic pole, it is important to derive a set of closed and explicit dynamic equations of motion describing the behaviour of the system. Section 3.2.2 of this thesis presented the mathematical derivations of equations approximating the behaviour of the system. It was found that the system equations are highly nonlinear. The correctness of these equations was verified qualitatively by reviewing the graphical output of the computer simulation using the program MAYMAY. This program simulates a controller balancing an elastic pole on top of a cart moving on a limited track in real time. The design of the algorithm of this program is shown in section 3.3. Numerical integration using fourth order Runge-Kutta was implemented. The results of this program are shown in section 3.3.4. The efficiency of the program is tested by running it under various initial conditions. In this simulation the size of the cart, the track, and the pole may change. It can be seen from the results that the system still balances the pole and brings the cart to the centre of the track even though the magnitude of the force applied to the cart and the initial angle of the pole are changed.

Based on the results of the computer simulation, the author established that theoretically, it is likely to be possible to balance an elastic pole in its first mode of vibration on top of a cart moving along a limited track. It is therefore of considerable interest to implement this in the real world. This had not been demonstrated at this stage of the work. A flexible pole-cart was commissioned and supplied together with a PD based controller of limited capability. It was therefore a logical step to attempt to develop and benchmark novel non-conventional controllers for this problem.

8.3. Off-Line Neural Network Controller for the Flexible Pole-Cart Balancing Problem

As a first step neural networks based upon feedforward backpropagation and Kohonen networks were used to learn the forces required to balance a flexible pole on a cart by generating a learning controller in simulation. The information fed to the neural network was taken from the results of the computer simulation of the derived dynamics of the flexible pole-cart balancing problem.

The method of normalising input patterns prior to their application to the neural network is important. For the backpropagation network, difficulty was encountered in determining the direction of the required force. The neural network experienced local minima and did not converge for a bipolar value of force from -1.0 to + 1.0 (see table 4.2). This problem was resolved by adding an additional vector to the output. For the leftwards force the additional output vector must have a value of 0.0, otherwise 1.0. The application of momentum and noise terms helped also in avoiding local minima.

8.4. On-Line Neural Network Controller for the Flexible Pole-Cart Balancing Problem

Following the off-line implementation an on line hybrid controller using feedforward neural network (FNN) and a rule based evaluator was developed and tested to control the flexible pole-cart balancing problem on a testbed. The feedforward neural network learned from a set of training data derived from a real system and was initially tested against the computer simulation of the derived dynamics of the flexible pole-cart balancing system. The inputs to the neural network were the elastic pole deflection, the

elastic pole angle, the displacement of the cart, and the velocity of the cart. An electric motor has been used to control the motion of the cart, hence, the neural network output is a voltage needed for the electric motor to operate (-5 to +5 volts) rather than the fore used in the earlier simulation..

The FNN based controller developed then successfully balanced the real pole for a limited period. The control system frequently failed due to the cart running out of track. The inaccuracy of the initial value of all sensors was the main contributor to this failure. This problem was resolved using the performance evaluator (see figure 5.4). The FNN system was overridden by a small rule based supervisory system that periodically corrected extreme angles of the pole or caused the cart to decentralise on the track.

The results of the physical experiments on this controller were shown and discussed in section 5.4. Here, the robustness of the controller developed was verified and tested. This results also show the stability, flexibility and adaptability of the controller.

8.5. Fuzzy Logic System Controller for the Flexible Pole-Cart Balancing Problem

An on line fuzzy logic controller was then developed to balance the flexible pole hinged at its root on top of a cart moving along a limited track. The controller design uses 5 fuzzy logic systems and a rule based evaluator. A reduction of the number of fuzzy logic rules required for good control to 13 has been made in spite of the fact that the controller uses 6 input variables. The crisp output of the controller can be directly used as the input voltage of the actuator (motor). The controller, for most initial conditions of the system, is able to balance the flexible pole indefinitely and bring the cart to the centre of the track. The controller can accommodate external disturbances to the system (e.g., shaking the track randomly, elevating the end of the track, applying external forces and obstacles to the pole in any direction, etc.). The response of the controller is fast enough to balance the

flexible pole from an initial angle of 20 degrees. Nesting the fuzzy logic systems supported the filtering of noisy inputs.

This research confirms that a multiple fuzzy controller can be developed to control a complex and nonlinear system such as the flexible pole-cart balancing system without knowing its mathematical description. It should be observed, however, that such a system requires rule based additions to accommodate initial sensor offsets and extreme displacement conditions.

8.6. A Fuzzy-Genetic Controller for the Flexible Pole-Cart Balancing Problem

In this element of the work the author was able to apply a combination of fuzzy logic systems and genetic algorithm techniques to the control of the flexible pole-cart balancing problem. The controller developed uses the genetic algorithm as a trainer and the fuzzy logic system as a controller. The construction of the fuzzy logic system did not require the prior knowledge of a human expert because the genetic algorithm was able to achieve the optimal parameter entries necessary for the fuzzy logic system to perform and control the desired behaviour.

In this work, apart from being applied as a process controller, the fuzzy logic system is also being used as a universal approximator. The fuzzy logic system has been applied as a means of adjusting the system parameters so that the system output matches the training data (the desired behaviour of the flexible pole-cart balancing problem) with the aid of a genetic algorithm. This suggests that this system can be applied to a broad range of problems perhaps even including approximating a feedforward neural network (FNN). The advantage of this technique (when compared to the black box approach of FNN) is that, it is easy to look inside a FAM matrix and infer what it will do with the given input data. This is important for the user who requires transparent models of the control algorithms applied.

8.7. Summary

Table 8.1 shows the comparison of functions of the controllers developed and the route of their evolution. The following should be noted:

The neural network experiment determined that the pole deflection was the variable most prone to error. This was therefore measured directly in the fuzzy controller.

The fuzzy controller uses six inputs $x, \dot{x}, \theta_r, \dot{\theta}_r, d_e, \dot{d}_e$ while the neural network and evaluator only uses $x, \dot{x}, \theta_r, \theta_e$ (noting that $\theta_e = d_e / L$, where d_e is deflection and L is the length of the pole). The neural network structure does not use therefore $\dot{d}_e$ and $\dot{\theta}_r$ as input variables. A fuzzy controller using $x, \dot{x}, \theta_r$ and $\dot{d}_e$ as inputs was not able to balance the pole for infinite time. The 4-8-8-2 neural network structure built upon the understanding of the mechanics and control of the system determined during the construction of the simulation. It would appear that the 4-8-8-2 structure includes some implicit knowledge of the nonlinear dynamics of the system and the relationships between θ_e, $\dot{\theta}_r$, and $\dot{d}_e$. The weights structures of the network used to control the simulation and the test bed are different.

The fuzzy control system was the only system to use the input $\dot{d}_e$. This is a particularly noisy input. The cascaded approach was therefore implemented to remove the effect of the noise.

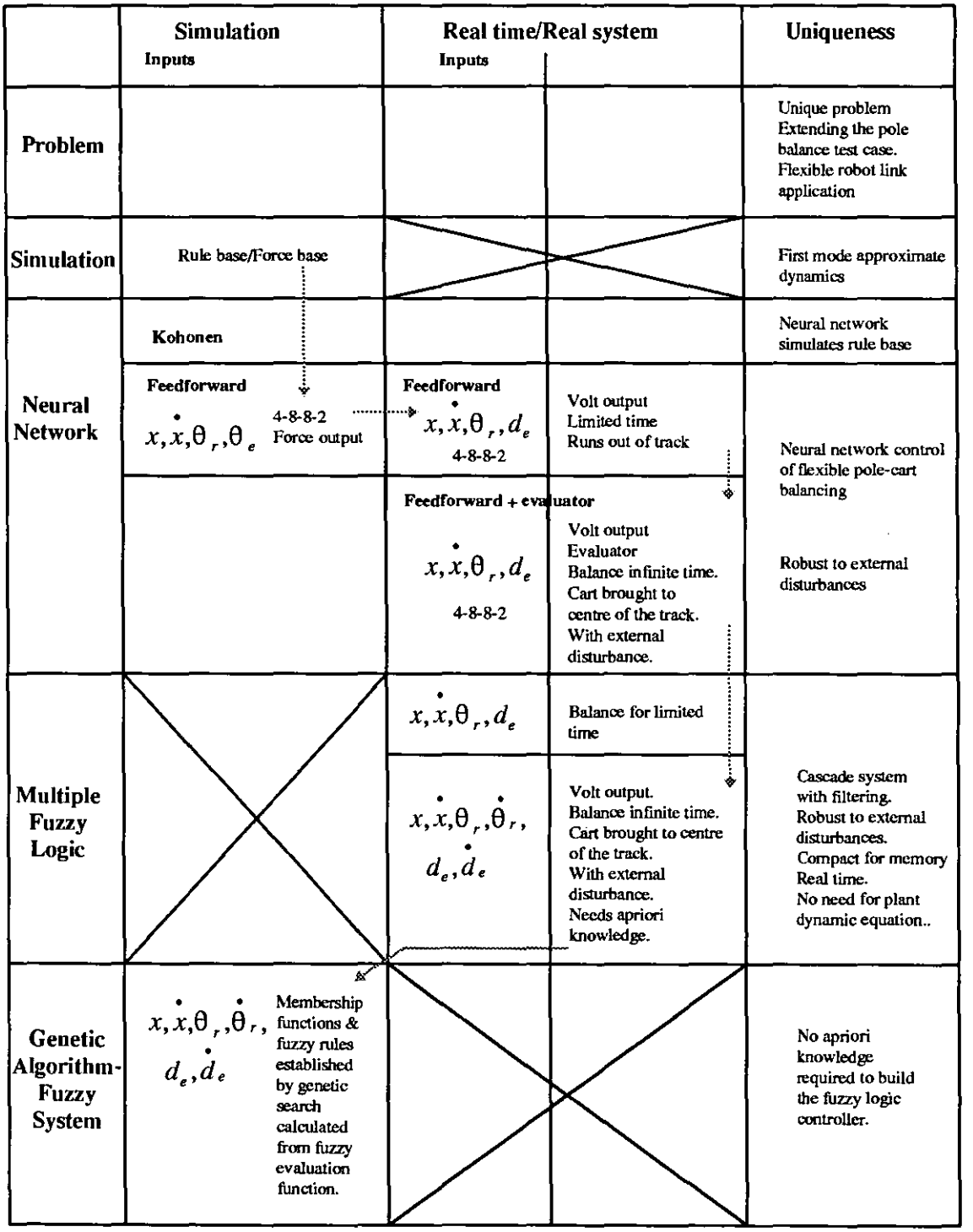


Table 8.1
Comparison of functions of controllers generated

8.8. Suggestions for Future Work

There are four areas that can be identified for further work.

1. An implementation of the fuzzy-genetic controller presented in Chapter 7 on the real physical flexible pole-cart balancing system. The major limitations in such an experiment is the availability of a fast processing machine, a parallel processing machine is most likely to be appropriate. If the real GA's learning process requires a long time (e.g., longer chromosomes, more input variables or more input fuzzy sets are required) then a delay might occur that results in failure. The mapping of simulated force output to real voltage output must also be resolved.

2. The construction of an on-line and off-line hybrid fuzzy-neural network controller to control the flexible pole-cart balancing system. Critical here is the determination of whether there is an improvement in the performance of the overall system by identifying new sets of parameters for fuzzy logic system decision making using gradient-descent optimisation techniques based on a neural network formulation of the problem. It is anticipated that learning can be take place in the neural network elements of the system and the implementation of control can be carried out by the fuzzy logic system.

3. The development of an intelligent controller for a flexible pole-cart balancing problem on which the flexible pole undergoes multi-mode vibrations by for example using a thinner beam with a higher elastic deflection. This adds further input parameters (variables) to the system. This problem may be resolved by adding a further fuzzy logic system to the multiple fuzzy logic controller.

4. Perhaps the most demanding task is to develop an intelligent controller that can control the position of the tip of the flexible pole irrespective of the position and

movement of the cart. This problem is much more representative of the true needs of the application of flexible robots. Determining the appropriate input-output variables for the task would make a good starting point for this work. It maybe possible to apply the multiple fuzzy logic systems technique described earlier. In this case, knowledge of the dynamic equations of the system may not be needed.

9. REFERENCES

1. A. Kandel, *"Fuzzy mathematical techniques with applications"*, New York: Adison-Wesley, 1986.
2. J. G. Kuschewski, S. H. Hui, S. H. Zak, *"Application of feedforward neural networks to dynamical system identification and control"*, IEEE Transactions of Control Systems Technology, Vol. 1, no. 1, pp. 37-49, March 1993.
3. D. H. Nguyen, B. Widrow *"Neural networks for self learning control system"*, IEEE Control System Magazine, pp. 18-23, April 1990.
4. C. W. Anderson, *"Learning to control an inverted pendulum using neural networks"*, IEEE Control Systems Magazine, Vol. 15, pp. 31-36, April 1989.
5. A. G. Barto, R. S. Sutton, and C. W. Anderson, *"Neuronlike adaptive elements that can solve difficult learning control problems"*, IEEE Systems Man. and Cybernetics, Vol. 13 pp. 834-846, 1983.
6. B. Widrow and F. W. Smith, *"Pattern recognition control systems"*, in Computer Info. Sci. (COINS) Symp. Proc. , Washington , DC, pp. 288-317, 1963.
7. G. Wang, D. K. Miu *"Unsupervised adaptation neural network control"*, Neural Network Joint Conf. , pp. 3-421-3-428, San Diego CA, 1990.
8. H. Ishibushi, R. Fujioka, H. Tanaka *"Neural networks that learn from fuzzy if then rules"*, IEEE Transactions on Fuzzy Systems, Vol. 1, no. 2, pp. 85-97, May 1993.
9. V. V. Tolat, B. Widrow, *"An adaptive broom balancer with visual inputs"*, Proc. Int. Conf. on Neural Networks, San Diego, Ca, pp. 2-641-2-647, July 1988.
10. S. Geva, J. Sitte *"A cartpole experiment benchmark for trainable controllers"*, IEEE Control Systems Magazine, pp. 40-51, Oct. 1993.
11. B. Zhang, *"Experiments in learning control using neural networks"*, Ph.D. Thesis Dissertation, University of Strathclyde, April 1991.
12. A. De Luca, G. Ulivi. *"Iterative learning control of robots with elastic joints"*, Proc. of 1992 IEEE International Conference on Robotics and Automation", NICE, France, pp. 1920-1925, May 1992.
13. R. M. Glorioso, *"Engineering Cybernetics"*, Prentice Hall Inc. Englewood Cliffs, New Jersey, 1975.

14. H. T. Milhorn, Jr. *"The application of control theory to physiological systems"*, W. B. Saunders Company, West Washington Square, Philadelphia, 1966.

15. M. M. Stransic, G. R. Pennock, *"A non degenerate orientation solution of a four jointed wrist"*, IEEE International Conference on Robotics and Automation, pp. 998-1003, March 1985.

16. S. Y. Oh, *"Control of redundant manipulators by inverse kinematics"*, Proc. of the IEEE Workshop on Intelligent Control, pp. 53-57, August 1985.

17. J. Sklansky, *"Adaptation, Learning, Self repair, and Feedback"*, IEEE Spectrum, Vol. 1, no. 5, pp. 172-174, May 1964.

18. W. R. Ashby, *"Design for a brain"*, 2nd ed., Wiley, New York, 1960.

19. I. Bratco, *"Qualitative Modelling: Learning and Control"*, Presented at 6th Czechoslovak Conf. AI, Prague, Czechoslovakia, pp. 97-112, June 1991.

20. R. H. Cannon, Jr., *"Dynamics of Physical Systems"*, New York: McGraw-Hill, 1967.

21. S. Horikawa, T. Furuhashi, S. Okuma, and Y. Uchikawa, *"A fuzzy controller using a neural network and its capability to learn expert's control rules"*, in Proc. IIZUKA'90, Vol. 1, pp. 103-106, 1990,.

22. C. T. Lin and C. S .G. Lee, *"Neural-network-based fuzzy logic control and decisions system"*, IEEE Transactions on Computers, Vol. 40, no. 12, pp. 1320-1336, 1991.

23. B. Kosko, *"Neural network and fuzzy systems"*, Quad Englewood Cliffs, NJ: Prentice Hall, 1992.

24. A. Kawamura, N. Watanabe, H. Okada, and K. Asakawa, *"A prototype of neuro-fuzzy cooperation system"*, in Proc. FUZZ-IEEE, pp. 1275-1282, 1992.

25. C. T. Lin and C. S. G. Lee, *"Real-time supervised structure/parameter learning for fuzzy neural network"*, in Proc. FUZZ-IEEE, pp. 1283-1291, 1992.

26. L. X. Wang and J. M. Mendel, *"Backpropagation fuzzy systems as nonlinear dynamic system identifier"*, in Proc. FUZZ-IEEE, pp. 1409-1419, 1992.

27. D. Michie and R. A. Chambers, *"BOXES : An experiment in adaptive control"*, in Machine Intelligence 2, E. Dale & D. Mitchie, Eds. Edinburgh, U.K: Oliver and Boyd, pp. 137-152, 1968.

28. D. Michie and R. A. Chambers, *"BOXES : As a model of pattern-formation"*, in Towards a Theoretical Biology, Vol. 1, Prolegomena C. H. Waddington, eds. Edinburgh: Edinburgh University Press, 1968.

29. C. J. Li, and T. S. Sankar, *"Systematic methods for efficient modelling and dynamics computation of flexible robot manipulators"*, IEEE Trans. on Systems, Man and Cybernetics, Vol. 23, no. 1, pp. 77-95, Jan/Feb. 1993.

30. C. H. Meng, and J. S. Chen, *"Dynamic modelling and payload-adaptive control of a flexible manipulator"*, Proc. IEEE International Conference on Robotics and Automation, pp. 488-493, 1988.

31. D. Wang, and M. Vidyasagar, *"Control of a flexible beam for optimum step response"*, Proc. IEEE International Conference. on Robotics and Automation, pp. 1567-1577, 1987.

32. J. S. Albus, *"Outline for a theory of intelligence"*, IEEE Transactions. on Systems, Man, and Cybernetics," Vol. 21, no. 3, pp. 473-509, May/June 1991.

33. R. H. Cannon, E. Schmitz, *"Initial experiments on the end-point control of a flexible one-link robot"*, International Journal of Robotic Research, Vol. 3, No 3, pp. 62-75, Fall 1984.

34. T. Fukuda, *"Flexibility control of elastic robotic arm"*, Journal of Robotic Systems, Vol. 2, no. 1, pp. 73-88, 1985.

35. G. G. Hastings, W. J. Book, *"A linear dynamic model for flexible robotic manipulators"*, IEEE Control System Magazine, Vol. 7, pp. 61-64, Feb. 1987.

36. G. G. Hastings, W. J. Book, *"Experiments in the optimal control of a flexible arm"*, Proc. of 1987 IEEE International Conference on Robotics and Automation, pp. 1024-1029, Spring, 1986.

37. W. B. Gervater, *"Basic relations for control of flexible vehicles"*, AIAA Journal., Vol. 8, no. 4, pp. 666-672, Apr. 1970.

38. B. V. Chapnik, G. R. Heppler, J. D. Aplevich, *"Modelling impact of a one-link flexible robotic arm"*, IEEE Transactions on Robotics and Automation. Vol. 7, no. 4, pp. 479-488, August 1991.

39. B. V. Chapnik, G. R. Heppler and J.D. Aplevich, *"Controlling the impact response of a one-link flexible robotic arm"*, IEEE Transactions on Robotics and Automation, Vol. 9, no. 3, pp. 346-351, June 1993.

40. J. Case, A. H. Chilver, *"Strength of materials and structures"*, 2nd ed., 1971.

41. S. P. Timoshenko, J. N. Goodler, *"Theory of elasticity"*, 3rd edition, McGraw-Hill Kogakusha Ltd., 1970.

42. B. Carnahan, H. A. Luther, J. O. Wilkes, *"Applied numerical methods"* , John Wiley and Sons, Inc., 1969.

43. F. Matsuno, T. Asano, and Y. Sakawa *"Modelling and quasi-static hybrid position/force control of constrained planar two-link flexible manipulators"*, IEEE Transactions on Robotics and Automation, Vol. 10, no. 3, pp. 287-297, June 1994.

44. C. Butler, Caudill. *"Understanding neural networks"*, A Bradford Book, The MIT Press, Cambridge, Massachusetts, London, England, 1992 .

45. C. H. Dagli. *"Artificial neural networks for intelligent manufacturing"*, Chapman & Hall, 1994.

46. A. U. Asar, J. R. Mcdonald, *"A specification of neural network applications in the load forecasting problem"*, IEEE Transactions on Control Systems Technology, Vol. 2, pp. 135-141, June 1994.

47. R. Hecht-Nielsen. *"Neurocomputing"*, Reading, MA: Addison-Wesley Publishing Company, p. 343, 1990.

48. J. Werbos. *"Beyond regression: new tools for prediction and analysis in the behavioral sciences"*, Doctoral Dissertation, Appl. Math., Harvard University, November 1974.

49. B. Parker, *"A comparison of algorithms for neuron like cells"*, Proc. Second Annual Conference On Neural Networks for Computing, pp. 327-332, Am. Inst. of Physics, New York, 1986.

50. E. Rumelhart, *"Parallel distributed processing: exploring in the microstructure of cognition"*, I & II, MIT Press, Cambridge MA, 1986.

51. T. Kohonen, *"Correlation matrix memories"*, IEEE Transactions on Computers, C-21, pp. 353-359,1972.

52. T. Kohonen, *"Self-organized formation of topology correct feature maps"*, Biol. Cyber., pp. 59-69, 1982.

53. B. Rao, H. V. Rao *"C++ neural networks and fuzzy logic"*, first ed., 1993.

54. T. Yamakawa, *"A fuzzy inference engine in nonlinear analog mode and Its application to a fuzzy logic control"*, IEEE Transactions on Neural Networks, Vol. 4, No. 3, pp. 496-522, May 1993.

55. L. A. Zadeh, *"Fuzzy sets"*, in Information and Control, New York: Academic Press, Vol. 8, pp. 338-353, 1965.

56. M. Mendel, *"Fuzzy logic systems for engineering: A tutorial"*, Proceedings of the IEEE, Vol. 83, No. 3, pp. 354-377, March 1995.

57. J. R. Jang, C. T. Sun, *"Neuro-fuzzy modelling and control"*, Proceedings of the IEEE, Vol. 83, No. 3, pp. 378-405, March 1995.

58. H. Mamdani, S. Assilian, *"An experiment in linguistic synthesis with a fuzzy logic controller"*, International Journal Man Mach. Studies, Vol. 7, no. 1, pp. 1-13, 1975.

59. H. Mamdani, *"Application of fuzzy logic to approximate reasoning using linguistic synthesis"*, IEEE Transactions on Computers, Vol. c-26, no. 12, pp. 1182-1191, 1977.

60. T. Welstead, *"Neural Networks and fuzzy logic applications"*, John Wiley and Sons Inc., 1994.

61. W. Pedrycz, *"Fuzzy control and fuzzy systems"*, New York: Wiley, 1989.

62. B. Kosko, *"Neural network and fuzzy systems"*, Englewood Cliffs, NJ: Prentice Hall, 1992.

63. L. Fu, *"Neural networks in computer intelligence"*, McGraw-Hill, Inc. p. 105, 1994.

64. A. Homaifar, E. McCornick, *"Simultaneous design of membership functions and rule sets for fuzzy controllers using genetic algorithms"*, IEEE Transactions on Fuzzy Systems, Vol. 3, No. 2, pp. 129-139, May 1995.

65. Y. M. Park, U. Moon, K. Y. Lee, *"A self-organising fuzzy logic controller for dynamic systems using a fuzzy auto-regressive moving average (FARMA) mode"*, IEEE Transactions on Fuzzy Systems, Vol. 3, No. 1, pp. 75-82, February 1995.

66. J. Kim, Y. Moon, B. P. Zeigler, *"Designing fuzzy net controllers using genetic algorithms"*, IEEE Control Systems, pp. 66-72, June 1995.

67. T. Welstead, *"Neural network and fuzzy logic applications in C/C++"*, John Wiley & Sons, Inc., 1994.

68. J. Janson, J. F. Frenzel, *"Training product unit neural networks with genetic algorithms"*, IEEE Expert, pp. 26-31, October 1993.

69. A. Varsek, T. Urbancic, B. Filipie, *"Genetic algorithms in controller design and tuning"*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 23, No. 5, pp. 1330-1339, Sept/Oct 1993.

70. K. Ogata, *"Discrete-time control systems"*, Prentice Hall, Englewood Cliffs, New Jersey, 2nd ed., 1995.

71. N. S., Nise *"Control Systems Engineering"*, The Benjamin/Cummings Publishing Company, Inc., 2nd ed., 1995.

72. E. P. Dadios, D. J. Williams, *"Flexible pole-cart balancing"*, Department of Manufacturing Engineering, Loughborough University of Technology, U.K, Processes Group Technical Report 94/9, 1994.

73. E. P. Dadios, D. J. Williams, *"Application of neural network to the flexible pole-cart balancing problem"*, Proc. of IEEE Systems Man and Cybernetics International Conference, Vancouver Canada, Vol. 3, pp. 2506-2511, October 22-25, 1995.

74. E. P. Dadios, D. J. Williams, *"Multiple fuzzy logic systems: A controller for the flexible pole-cart balancing problem"*, to be included in the Proc. of the IEEE Robotics and Automation International Conference, Minneapolis, Minnesota USA, April 24-26, 1996.

75. E. P. Dadios, D. J. Williams, *"A fuzzy-genetic controller for the flexible pole-cart balancing problem"*, to be included in the Proc. of the IEEE 3rd International Conference on Evolutionary Computation (ICEC'96), Nagoya, Japan, May 20-22, 1996.

Appendix A

Program MAYMAY source code

This is a Pascal language computer program that will simulate the mathematical model of the flexible pole cart-balancing system

The following are the options to choose for the user to run the program

1. Simulate new values for rigid pole.

2. Simulate new values for elastic pole.

3. Plot the graph: rigid pole angle vs. time without friction.

4. Plot the graph: rigid pole angle vs. time with friction.

7. Plot the graph cart displacement vs. time (elastic pole).

8. Plot the graph cart acceleration vs. time (elastic pole).

9. Plot the graph cart velocity vs. time (elastic pole).

0. To quit program.

Important procedures in the program:

1. **runge( )** - A procedure to solve differential equations using Runge-Kutta algorithm.
2. **check_pole_angle( )** - A procedure to check the position of the pole and apply the force necessary to balance the pole.
3. **find_elas_pole_acc_vel_ang( )** - A procedure to find the elastic pole velocity and acceleration.
4. **numerical_integration( )** - A procedure used to simulate the system by solving differential equations using numerical integration technique.
5. **find_elastic_angle( )** - A procedure to find the elastic pole angle.
6. **solve_cart_displacement( )** - A procedure to find the cart displacement and velocity.

7. **find_cart_acceleration( )** - A procedure to find the acceleration of the cart.
8. **start_g_r_wof( )** - A procedure used to plot the graphs of pole angle, cart displacement, cart velocity, cart acceleration versus time.
9. **draw_pole_cart( )** - A procedure to draw graphically the entire pole-cart balancing system in real time.

```
/** The main Program **?
program maymay;
uses crt,gra28_car,glob_dat;
const n=2;
type
    order = array[1..n] of real;          {array use to store the value of pole angle and velocity}
var
    f1,f2 :text;                          {text file use for storing data}
    T,in_force,c_force,c_f :REAL;         {T=time; in_force=initial force; c_force=the force at any
                                                    angle; c_f=coeficient of friction}
  H,TMAX,nume,dnume,t_limit:REAL;         {H=integration time step; TMAX=total simulation time;
                                          {nume=numerator; dnume=denomenator; t_limit=time limit}
  M,IFREQ,Icount,k,ctr      :INTEGER;
  F,V                 :order;             {V[1]=pole angle; V[2] = F[1]=pole velocity}
                                          {F[2]=pole acceleartion}
  ch                  :char;
  failure             :boolean;

{**** procedure to enter data ***}
procedure in_data(ch:char);
var x,y :integer;
begin
     clrscr; textcolor(yellow);
     write('Mass of the pole                = ');
     readln(mp);
     write('Total mass of the pole and the cart  = ');
     readln(mt);
     write('Total length of the pole        = ');
     readln(pl);
     if ch <> '1' then              {for elastic pole}
     begin
      write('Breadth of the pole            = ');
      readln(pb);
      write('Depth of the pole              = ');
      readln(pd);
```

```
      write('Elasticity of the pole          = ');
      readln(E);
    end
    else
    begin                    {this is for rigid pole}
       pb := 0.0; pd := 0.0; e := 0.0
    end;
    x := wherex; y := wherey;
    repeat
       gotoxy(x,y);
       write('Initial force applied (not zero)   = ');
       readln(in_force);
    until in_force <> 0.0;
    write('Coefficient of friction          = ');
    readln(c_f);
    write('Step size (H)                    = ');
    readln(H);
    write('Upper limit of integration (tmax)  = ');
    readln(tmax);
    write('Freq. intermediate printouts (Ifreq) = ');
    readln(IFREQ);
    T := 0.0;              {time}
    g := 9.81;             {acceleration due to gravity}
    write('Initial pole angle (degrees)       = ');
    readln(V[1]);
    V[1] := V[1]*6.283185/360;
    write('Limitation of pole angle (degrees)  = ');
    readln(t_limit);
    t_limit := t_limit*6.283185/360;
    V[2] := 0.0;           {initial velocity of the pole}
    Icount := 0;
    ctr := 1;              {first array element}
    c_force := in_force;
    c_ac[1] := in_force/mt;   {initial acceleration of the cart}
end;  {in_data}

{*** procedure to write initial values to external file **}
procedure write_init_values_to_file;
begin
   writeln(f1);
   writeln(f1,'     ***** Initial Data ******');
   writeln(f1,' Mass of the pole                =',mp:10:6);
   writeln(f1,' Total mass of the pole and the cart =',mt:10:6);
   writeln(f1,' Total length of the pole         =',pl:10:6);
   writeln(f1,' Breadth of the pole              =',pb:10:6);
   writeln(f1,' Depth of the pole                =',pd:10:6);
   writeln(f1,' Elasticity of the pole           =',e:12:2);
   writeln(f1,' Initial force applied            =',in_force:10:6);
   writeln(f1,' Coifficient of friction          =',c_f:10:6);
   writeln(f1,' Step size (H)                    =',h:10:6);
   writeln(f1,' Upper limit of integration (tmax)  =',tmax:10:6);
   writeln(f1,' Freq. intermediate printouts (Ifreq) =',ifreq:10);
   writeln(f1,' Initial time (t sec)             = 0.0');
```

```
    writeln(f1,' Initial pole angle (theta in rad)   =',V[1]:10:6);
    writeln(f1,' Limitations of pole angle           =',t_limit:10:6);
    writeln(f1,' Initial pole velocity (dtheta/dt)   = 0.0');
    writeln(f1,' Acceleration due to gravity         = 9.81 m/sq sec.');
    writeln(f1,'     OUTPUT VALUES:' );
end; {write_init_values}

{*** procedure to write initial values to external file **}
procedure write_init_values_to_file2;
begin
    writeln(f2);
    writeln(f2,'      ***** Initial Data ******');
    writeln(f2,' Mass of the pole                    =',mp:10:6);
    writeln(f2,' Total mass of the pole and the cart =',mt:10:6);
    writeln(f2,' Total length of the pole            =',pl:10:6);
    writeln(f2,' Breadth of the pole                 =',pb:10:6);
    writeln(f2,' Depth of the pole                   =',pd:10:6);
    writeln(f2,' Elasticity of the pole              =',e:12:2);
    writeln(f2,' Initial force applied               =',in_force:10:6);
    writeln(f2,' Coifficient of friction             =',c_f:10:6);
    writeln(f2,' Step size (H)                       =',h:10:6);
    writeln(f2,' Upper limit of integration (tmax)   =',tmax:10:6);
    writeln(f2,' Freq. intermediate printouts (Ifreq) =',ifreq:10);
    writeln(f2,' Initial time (t sec)                = 0.0');
    writeln(f2,' Initial pole angle (theta in rad)   =',V[1]:10:6);
    writeln(f2,' Limitations of pole angle           =',t_limit:10:6);
    writeln(f2,' Initial pole velocity (dtheta/dt)   = 0.0');
    writeln(f2,' Acceleration due to gravity         = 9.81 m/sq sec.');
    writeln(f2,'     OUTPUT VALUES:' );
end; {write_init_values}

{*** procedure to output data to external file *****}
procedure out_file;
begin
    write_init_values_to_file;
    writeln(f1,'Time':8,'C_accel':13,'Pole_ang':13,'Pole_ang_vel':14,'Force':7,'P_accel':14);
    write(f1,T:10:6,'  ',c_ac[1]:10:6,'  ',V[1]:10:6,'  ',V[2]:10:6,'  ',in_force:10:6);
    time_co[ctr] := T;
    angle_co[ctr] := V[1];
end;  {out_file}

{**** procedure to use runge kutta algorithm ***}
procedure runge(var m,k:integer;n:integer;var Y,F:order;var x,h:real);
var
  j :integer;
  savey,phi :array[1..50] of real;
begin
    m := m + 1;
    case m of
      1: k := 1;
      2: begin
           for j := 1 to n do
           begin
              savey[j] := Y[j];
```

```
            phi[j] := F[j];
            Y[j] := savey[j] + 0.5*H*F[j];
          end;
          x := x + 0.5 * H;
          k := 1;
        end;
      3: begin
          for j := 1 to n do
          begin
            phi[j] := phi[j] + 2.0*F[j];
            Y[j] := savey[j] + 0.5*H*F[j];
          end;
          k := 1;
        end;
      4: begin
          for j := 1 to n do
          begin
            phi[j] := phi[j] + 2.0*F[j];
            Y[j] := savey[j] + H*F[j];
          end;
          x := x + 0.5 * H;
          k := 1;
        end;
      5: begin
          for j := 1 to n do
            Y[j] := savey[j] + (phi[j] + F[j])*H/6.0;
          m := 0;
          k := 0;
        end;
    end; {case}
end; {*** runge **}

{*** procedure to check the location of the pole ****}
{procedure check_pole_angle;
begin
  if (V[1] > t_limit) or (V[1] < (-1)*t_limit) then
  begin
     clrscr;
     writeln('  Failure !!!! angle reached the limit (in radians) = ',t_limit:10:6);
     writeln('  Press return to exit .... ');
     failure := true;
     readln; close(f1);
     exit;
  end
  else if (V[1] > 0.0) then
    begin
      if c_force < 0.0 then
        c_force := c_force*(-1);
    end
  else if (V[1] < 0.0) then
    begin
      if c_force > 0.0 then
        c_force := c_force*(-1)
    end
```

```
end; }  {check_pole_angle}

{*** procedure to check the location of the pole ****}
procedure check_pole_angle;
begin
  if (V[1] > t_limit) or (V[1] < (-1)*t_limit) then
  begin
      clrscr;
      writeln('  Failure !!!! angle reached the limit (in radians) = ',t_limit:10:6);
      write('  Press return to exit .... ');
      failure := true;
      readln;
      exit;
  end;
  if ((V[1] >= -0.0009)and(V[1] <= 0.0009)) then
    c_force := 0.0              {theta is at 0 angle don't apply force}
  else if ((V[1] > -0.001)and(V[1] <= -0.0009)) then
    c_force := -0.1
  else if ((V[1] > 0.0009)and(V[1] <= 0.001)) then
    c_force := 0.1
  else if ((V[1] > -0.003)and(V[1] <= -0.001)) then
    c_force := -0.2
  else if ((V[1] > 0.001)and(V[1] <= 0.003)) then
    c_force := 0.2
  else if ((V[1] > -0.006)and(V[1] <= -0.003)) then
    c_force := -0.3
  else if ((V[1] > 0.003)and(V[1] <= 0.006)) then
    c_force := 0.3
  else if ((V[1] > -0.009)and(V[1] <= -0.006)) then
    c_force := -0.4
  else if ((V[1] > 0.006)and(V[1] <= 0.009)) then
    c_force := 0.4
  else if ((V[1] > -0.011)and(V[1] <= -0.009))  then
    c_force := -0.5
  else if ((V[1] > 0.009)and(V[1] <= 0.011))  then
    c_force := 0.5
  else if ((V[1] > -0.013)and(V[1] <= -0.011))  then
    c_force := -0.6
  else if ((V[1] > 0.011)and(V[1] <= 0.013))  then
    c_force := 0.6
  else if ((V[1] > -0.015)and(V[1] <= -0.013))  then
    c_force := -0.7
  else if ((V[1] > 0.013)and(V[1] <= 0.015))  then
    c_force := 0.7
  else if ((V[1] > -0.017)and(V[1] <= -0.015))  then
    c_force := -0.8
  else if ((V[1] > 0.015)and(V[1] <= 0.017))  then
    c_force := 0.8
  else if ((V[1] > -0.019)and(V[1] <= -0.017))  then
    c_force := -0.9
  else if ((V[1] > 0.017)and(V[1] <= 0.019))  then
    c_force := 0.9
  else if ((V[1] > -0.020)and(V[1] <= -0.019))  then
    c_force := -1.0
```

```
else if ((V[1] > 0.019)and(V[1] <= 0.020))  then
  c_force := 1.0
else if ((V[1] > -0.022)and(V[1] <= -0.020))  then
  c_force := -1.1
else if ((V[1] > 0.020)and(V[1] <= 0.022))  then
  c_force := 1.1
else if ((V[1] > -0.024)and(V[1] <= -0.022))  then
  c_force := -1.2
else if ((V[1] > 0.022)and(V[1] <= 0.024))  then
  c_force := 1.2
else if ((V[1] > -0.026)and(V[1] <= -0.024))  then
  c_force := -1.3
else if ((V[1] > 0.024)and(V[1] <= 0.026))  then
  c_force := 1.3
else if ((V[1] > -0.028)and(V[1] <= -0.026))  then
  c_force := -1.4
else if ((V[1] > 0.026)and(V[1] <= 0.028))  then
  c_force := 1.4
else if ((V[1] > -0.03)and(V[1] <= -0.028))  then
  c_force := -1.5
else if ((V[1] > 0.028)and(V[1] <= 0.030))  then
  c_force := 1.5
else if ((V[1] > -0.032)and(V[1] <= -0.030))  then
  c_force := -1.6
else if ((V[1] > 0.030)and(V[1] <= 0.032))  then
  c_force := 1.6
else if ((V[1] > -0.034)and(V[1] <= -0.032))  then
  c_force := -1.7
else if ((V[1] > 0.032)and(V[1] <= 0.034))  then
  c_force := 1.7
else if ((V[1] > -0.036)and(V[1] <= -0.034))  then
  c_force := -1.8
else if ((V[1] > 0.034)and(V[1] <= 0.036))  then
  c_force := 1.8
else if ((V[1] > -0.038)and(V[1] <= -0.036))  then
  c_force := -1.9
else if ((V[1] > 0.036)and(V[1] <= 0.038))  then
  c_force := 1.9
else if ((V[1] > -0.040)and(V[1] <= -0.038))  then
  c_force := -2.0
else if ((V[1] > 0.038)and(V[1] <= 0.040))  then
  c_force := 2.0
else if ((V[1] > -0.043)and(V[1] <= -0.040))  then
  c_force := -2.1
else if ((V[1] > 0.040)and(V[1] <= 0.043))  then
  c_force := 2.1
else if ((V[1] > -0.046)and(V[1] <= -0.043))  then
  c_force := -2.2
else if ((V[1] > 0.043)and(V[1] <= 0.046))  then
  c_force := 2.2
else if ((V[1] > -0.049)and(V[1] <= -0.046))  then
  c_force := -2.3
else if ((V[1] > 0.046)and(V[1] <= 0.049))  then
  c_force := 2.3
```

```
else if ((V[1] > -0.052)and(V[1] <= -0.049))  then
  c_force := -2.4
else if ((V[1] > 0.049)and(V[1] <= 0.052))  then
  c_force := 2.4
else if ((V[1] > -0.055)and(V[1] <= -0.052))  then
  c_force := -2.5
else if ((V[1] > 0.052)and(V[1] <= 0.055))  then
  c_force := 2.5
else if ((V[1] > -0.058)and(V[1] <= -0.055))  then
  c_force := -2.6
else if ((V[1] > 0.055)and(V[1] <= 0.058))  then
  c_force := 2.6
else if ((V[1] > -0.061)and(V[1] <= -0.058))  then
  c_force := -2.7
else if ((V[1] > 0.058)and(V[1] <= 0.061))  then
  c_force := 2.7
else if ((V[1] > -0.064)and(V[1] <= -0.061))  then
  c_force := -2.8
else if ((V[1] > 0.061)and(V[1] <= 0.064))  then
  c_force := 2.8
else if ((V[1] > -0.067)and(V[1] <= -0.064))  then
  c_force := -2.9
else if ((V[1] > 0.064)and(V[1] <= 0.067))  then
  c_force := 2.9
else if ((V[1] > -0.070)and(V[1] <= -0.067))  then
  c_force := -3.0
else if ((V[1] > 0.067)and(V[1] <= 0.070))  then
  c_force := 3.0
else if ((V[1] > -0.073)and(V[1] <= -0.070))  then
  c_force := -3.1
else if ((V[1] > 0.070)and(V[1] <= 0.073))  then
  c_force := 3.1
else if ((V[1] > -0.076)and(V[1] <= -0.073))  then
  c_force := -3.2
else if ((V[1] > 0.073)and(V[1] <= 0.076))  then
  c_force := 3.2
else if ((V[1] > -0.079)and(V[1] <= -0.076))  then
  c_force := -3.3
else if ((V[1] > 0.076)and(V[1] <= 0.079))  then
  c_force := 3.3
else if ((V[1] > -0.082)and(V[1] <= -0.079))  then
  c_force := -3.4
else if ((V[1] > 0.079)and(V[1] <= 0.082))  then
  c_force := 3.4
else if ((V[1] > -0.085)and(V[1] <= -0.082))  then
  c_force := -3.5
else if ((V[1] > 0.082)and(V[1] <= 0.085))  then
  c_force := 3.5
else if ((V[1] > -0.088)and(V[1] <= -0.085))  then
  c_force := -3.6
else if ((V[1] > 0.085)and(V[1] <= 0.088))  then
  c_force := 3.6
else if ((V[1] > -0.091)and(V[1] <= -0.088))  then
  c_force := -3.7
```

```
else if ((V[1] > 0.088)and(V[1] <= 0.091))  then
  c_force := 3.7
else if ((V[1] > -0.094)and(V[1] <= -0.091))  then
  c_force := -3.8
else if ((V[1] > 0.091)and(V[1] <= 0.094))  then
  c_force := 3.8
else if ((V[1] > -0.097)and(V[1] <= -0.094))  then
  c_force := -3.9
else if ((V[1] > 0.094)and(V[1] <= 0.097))  then
  c_force := 3.9
else if ((V[1] > -0.100)and(V[1] <= -0.097))  then
  c_force := -4.0
else if ((V[1] > 0.097)and(V[1] <= 0.100))  then
  c_force := 4.0
else if ((V[1] > -0.103)and(V[1] <= -0.100))  then
  c_force := -4.1
else if ((V[1] > 0.100)and(V[1] <= 0.103))  then
  c_force := 4.1
else if ((V[1] > -0.106)and(V[1] <= -0.103))  then
  c_force := -4.2
else if ((V[1] > 0.103)and(V[1] <= 0.106))  then
  c_force := 4.2
else if ((V[1] > -0.109)and(V[1] <= -0.106))  then
  c_force := -4.3
else if ((V[1] > 0.106)and(V[1] <= 0.109))  then
  c_force := 4.3
else if ((V[1] > -0.112)and(V[1] <= -0.109))  then
  c_force := -4.4
else if ((V[1] > 0.109)and(V[1] <= 0.112))  then
  c_force := 4.4
else if ((V[1] > -0.115)and(V[1] <= -0.112))  then
  c_force := -4.5
else if ((V[1] > 0.112)and(V[1] <= 0.115))  then
  c_force := 4.5
else if ((V[1] > -0.118)and(V[1] <= -0.115))  then
  c_force := -4.6
else if ((V[1] > 0.115)and(V[1] <= 0.118))  then
  c_force := 4.6
else if ((V[1] > -0.121)and(V[1] <= -0.118))  then
  c_force := -4.7
else if ((V[1] > 0.118)and(V[1] <= 0.121))  then
  c_force := 4.7
else if ((V[1] > -0.124)and(V[1] <= -0.121))  then
  c_force := -4.8
else if ((V[1] > 0.121)and(V[1] <= 0.124))  then
  c_force := 4.8
else if ((V[1] > -0.127)and(V[1] <= -0.124))  then
  c_force := -4.9
else if ((V[1] > 0.124)and(V[1] <= 0.127))  then
  c_force := 4.9
else if ((V[1] > -0.130)and(V[1] <= -0.127))  then
  c_force := -5.0
else if ((V[1] > 0.127)and(V[1] <= 0.130))  then
  c_force := 5.0
```

```
else if ((V[1] > -0.133)and(V[1] <= -0.130))  then
  c_force := -5.1
else if ((V[1] > 0.130)and(V[1] <= 0.133))  then
  c_force := 5.1
else if ((V[1] > -0.136)and(V[1] <= -0.133))  then
  c_force := -5.2
else if ((V[1] > 0.133)and(V[1] <= 0.136))  then
  c_force := 5.2
else if ((V[1] > -0.139)and(V[1] <= -0.136))  then
  c_force := -5.3
else if ((V[1] > 0.136)and(V[1] <= 0.139))  then
  c_force := 5.3
else if ((V[1] > -0.142)and(V[1] <= -0.139))  then
  c_force := -5.4
else if ((V[1] > 0.139)and(V[1] <= 0.142))  then
  c_force := 5.4
else if ((V[1] > -0.145)and(V[1] <= -0.142))  then
  c_force := -5.5
else if ((V[1] > 0.142)and(V[1] <= 0.145))  then
  c_force := 5.5
else if ((V[1] > -0.148)and(V[1] <= -0.145))  then
  c_force := -5.6
else if ((V[1] > 0.145)and(V[1] <= 0.148))  then
  c_force := 5.6
else if ((V[1] > -0.151)and(V[1] <= -0.148))  then
  c_force := -5.7
else if ((V[1] > 0.148)and(V[1] <= 0.151))  then
  c_force := 5.7
else if ((V[1] > -0.154)and(V[1] <= -0.151))  then
  c_force := -5.8
else if ((V[1] > 0.151)and(V[1] <= 0.154))  then
  c_force := 5.8
else if ((V[1] > -0.157)and(V[1] <= -0.154))  then
  c_force := -5.9
else if ((V[1] > 0.154)and(V[1] <= 0.157))  then
  c_force := 5.9
else if ((V[1] > -0.160)and(V[1] <= -0.157))  then
  c_force := -6.0
else if ((V[1] > 0.157)and(V[1] <= 0.160))  then
  c_force := 6.0
else if ((V[1] > -0.163)and(V[1] <= -0.160))  then
  c_force := -6.1
else if ((V[1] > 0.160)and(V[1] <= 0.163))  then
  c_force := 6.1
else if ((V[1] > -0.166)and(V[1] <= -0.163))  then
  c_force := -6.2
else if ((V[1] > 0.163)and(V[1] <= 0.166))  then
  c_force := 6.2
else if ((V[1] > -0.169)and(V[1] <= -0.166))  then
  c_force := -6.3
else if ((V[1] > 0.166)and(V[1] <= 0.169))  then
  c_force := 6.3
else if ((V[1] > -0.172)and(V[1] <= -0.169))  then
  c_force := -6.4
```

```
else if ((V[1] > 0.169)and(V[1] <= 0.172))  then
  c_force := 6.4
else if ((V[1] > -0.175)and(V[1] <= -0.172))  then
  c_force := -6.5
else if ((V[1] > 0.172)and(V[1] <= 0.175))  then
  c_force := 6.5
else if ((V[1] > -0.178)and(V[1] <= -0.175))  then
  c_force := -6.6
else if ((V[1] > 0.175)and(V[1] <= 0.178))  then
  c_force := 6.6
else if ((V[1] > -0.181)and(V[1] <= -0.178))  then
  c_force := -6.7
else if ((V[1] > 0.178)and(V[1] <= 0.181))  then
  c_force := 6.7
else if ((V[1] > -0.184)and(V[1] <= -0.181))  then
  c_force := -6.8
else if ((V[1] > 0.181)and(V[1] <= 0.184))  then
  c_force := 6.8
else if ((V[1] > -0.187)and(V[1] <= -0.184))  then
  c_force := -6.9
else if ((V[1] > 0.184)and(V[1] <= 0.187))  then
  c_force := 6.9
else if ((V[1] > -0.190)and(V[1] <= -0.187))  then
  c_force := -7.0
else if ((V[1] > 0.187)and(V[1] <= 0.190))  then
  c_force := 7.0
else if ((V[1] > -0.193)and(V[1] <= -0.190))  then
  c_force := -7.1
else if ((V[1] > 0.190)and(V[1] <= 0.193))  then
  c_force := 7.1
else if ((V[1] > -0.196)and(V[1] <= -0.193))  then
  c_force := -7.2
else if ((V[1] > 0.193)and(V[1] <= 0.196))  then
  c_force := 7.2
else if ((V[1] > -0.199)and(V[1] <= -0.196))  then
  c_force := -7.3
else if ((V[1] > 0.196)and(V[1] <= 0.199))  then
  c_force := 7.3
else if ((V[1] > -0.202)and(V[1] <= -0.199))  then
  c_force := -7.4
else if ((V[1] > 0.169)and(V[1] <= 0.202))  then
  c_force := 7.4
else if ((V[1] > -0.205)and(V[1] <= -0.202))  then
  c_force := -7.5
else if ((V[1] > 0.202)and(V[1] <= 0.205))  then
  c_force := 7.5
else if ((V[1] > -0.208)and(V[1] <= -0.205))  then
  c_force := -7.6
else if ((V[1] > 0.205)and(V[1] <= 0.208))  then
  c_force := 7.6
else if ((V[1] > -0.211)and(V[1] <= -0.208))  then
  c_force := -7.7
else if ((V[1] > 0.208)and(V[1] <= 0.211))  then
  c_force := 7.7
```

```
else if ((V[1] > -0.214)and(V[1] <= -0.211))  then
  c_force := -7.8
else if ((V[1] > 0.211)and(V[1] <= 0.214))  then
  c_force := 7.8
else if ((V[1] > -0.217)and(V[1] <= -0.214))  then
  c_force := -7.9
else if ((V[1] > 0.214)and(V[1] <= 0.217))  then
  c_force := 7.9
else if ((V[1] > -0.220)and(V[1] <= -0.217))  then
  c_force := -8.0
else if ((V[1] > 0.217)and(V[1] <= 0.220))  then
  c_force := 8.0
else if ((V[1] > -0.223)and(V[1] <= -0.220))  then
  c_force := -8.1
else if ((V[1] > 0.220)and(V[1] <= 0.223))  then
  c_force := 8.1
else if ((V[1] > -0.226)and(V[1] <= -0.223))  then
  c_force := -8.2
else if ((V[1] > 0.223)and(V[1] <= 0.226))  then
  c_force := 8.2
else if ((V[1] > -0.229)and(V[1] <= -0.226))  then
  c_force := -8.3
else if ((V[1] > 0.226)and(V[1] <= 0.229))  then
  c_force := 8.3
else if ((V[1] > -0.232)and(V[1] <= -0.229))  then
  c_force := -8.4
else if ((V[1] > 0.229)and(V[1] <= 0.232))  then
  c_force := 8.4
else if ((V[1] > -0.235)and(V[1] <= -0.232))  then
  c_force := -8.5
else if ((V[1] > 0.232)and(V[1] <= 0.235))  then
  c_force := 8.5
else if ((V[1] > -0.238)and(V[1] <= -0.235))  then
  c_force := -8.6
else if ((V[1] > 0.235)and(V[1] <= 0.238))  then
  c_force := 8.6
else if ((V[1] > -0.241)and(V[1] <= -0.238))  then
  c_force := -8.7
else if ((V[1] > 0.238)and(V[1] <= 0.241))  then
  c_force := 8.7
else if ((V[1] > -0.244)and(V[1] <= -0.241))  then
  c_force := -8.8
else if ((V[1] > 0.241)and(V[1] <= 0.244))  then
  c_force := 8.8
else if ((V[1] > -0.247)and(V[1] <= -0.244))  then
  c_force := -8.9
else if ((V[1] > 0.244)and(V[1] <= 0.247))  then
  c_force := 8.9
else if ((V[1] > -0.250)and(V[1] <= -0.247))  then
  c_force := -9.0
else if ((V[1] > 0.247)and(V[1] <= 0.250))  then
  c_force := 9.0
else if ((V[1] > -0.253)and(V[1] <= -0.250))  then
  c_force := -9.1
```

```
else if ((V[1] > 0.250)and(V[1] <= 0.253))  then
  c_force := 9.1
else if ((V[1] > -0.256)and(V[1] <= -0.253))  then
  c_force := -9.2
else if ((V[1] > 0.253)and(V[1] <= 0.256))  then
  c_force := 9.2
else if ((V[1] > -0.259)and(V[1] <= -0.256))  then
  c_force := -9.3
else if ((V[1] > 0.256)and(V[1] <= 0.259))  then
  c_force := 9.3
else if ((V[1] > -0.262)and(V[1] <= -0.259))  then
  c_force := -9.4
else if ((V[1] > 0.259)and(V[1] <= 0.262))  then
  c_force := 9.4
else if ((V[1] > -0.265)and(V[1] <= -0.262))  then
  c_force := -9.5
else if ((V[1] > 0.262)and(V[1] <= 0.265))  then
  c_force := 9.5
else if ((V[1] > -0.268)and(V[1] <= -0.265))  then
  c_force := -9.6
else if ((V[1] > 0.265)and(V[1] <= 0.268))  then
  c_force := 9.6
else if ((V[1] > -0.271)and(V[1] <= -0.268))  then
  c_force := -9.7
else if ((V[1] > 0.268)and(V[1] <= 0.271))  then
  c_force := 9.7
else if ((V[1] > -0.274)and(V[1] <= -0.271))  then
  c_force := -9.8
else if ((V[1] > 0.271)and(V[1] <= 0.274))  then
  c_force := 9.8
else if ((V[1] > -0.277)and(V[1] <= -0.274))  then
  c_force := -9.9
else if ((V[1] > 0.274)and(V[1] <= 0.277))  then
  c_force := 9.9;

if ((V[1] > -0.280)and(V[1] <= -0.277))  then
  c_force := -10.0
else if ((V[1] > 0.277)and(V[1] <= 0.280))  then
  c_force := 10.0
else if ((V[1] > -0.283)and(V[1] <= -0.280))  then
  c_force := -10.1
else if ((V[1] > 0.280)and(V[1] <= 0.283))  then
  c_force := 10.1
else if ((V[1] > -0.286)and(V[1] <= -0.283))  then
  c_force := -10.2
else if ((V[1] > 0.283)and(V[1] <= 0.286))  then
  c_force := 10.2
else if ((V[1] > -0.289)and(V[1] <= -0.286))  then
  c_force := -10.3
else if ((V[1] > 0.286)and(V[1] <= 0.289))  then
  c_force := 10.3
else if ((V[1] > -0.292)and(V[1] <= -0.289))  then
  c_force := -10.4
else if ((V[1] > 0.289)and(V[1] <= 0.292))  then
```

```
  c_force := 10.4
else if ((V[1] > -0.295)and(V[1] <= -0.292))  then
  c_force := -10.5
else if ((V[1] > 0.292)and(V[1] <= 0.295))  then
  c_force := 10.5
else if ((V[1] > -0.298)and(V[1] <= -0.295))  then
  c_force := -10.6
else if ((V[1] > 0.295)and(V[1] <= 0.298))  then
  c_force := 10.6
else if ((V[1] > -0.301)and(V[1] <= -0.298))  then
  c_force := -10.7
else if ((V[1] > 0.298)and(V[1] <= 0.301))  then
  c_force := 10.7
else if ((V[1] > -0.304)and(V[1] <= -0.301))  then
  c_force := -10.8
else if ((V[1] > 0.301)and(V[1] <= 0.304))  then
  c_force := 10.8
else if ((V[1] > -0.307)and(V[1] <= -0.304))  then
  c_force := -10.9
else if ((V[1] > 0.304)and(V[1] <= 0.307))  then
  c_force := 10.9
else if ((V[1] > -0.310)and(V[1] <= -0.307))  then
  c_force := -11.0
else if ((V[1] > 0.307)and(V[1] <= 0.310))  then
  c_force := 11.0
else if ((V[1] > -0.313)and(V[1] <= -0.310))  then
  c_force := -11.1
else if ((V[1] > 0.310)and(V[1] <= 0.313))  then
  c_force := 11.1
else if ((V[1] > -0.316)and(V[1] <= -0.313))  then
  c_force := -11.2
else if ((V[1] > 0.313)and(V[1] <= 0.316))  then
  c_force := 11.2
else if ((V[1] > -0.319)and(V[1] <= -0.316))  then
  c_force := -11.3
else if ((V[1] > 0.316)and(V[1] <= 0.319))  then
  c_force := 11.3
else if ((V[1] > -0.322)and(V[1] <= -0.319))  then
  c_force := -11.4
else if ((V[1] > 0.319)and(V[1] <= 0.322))  then
  c_force := 11.4
else if ((V[1] > -0.325)and(V[1] <= -0.322))  then
  c_force := -11.5
else if ((V[1] > 0.322)and(V[1] <= 0.325))  then
  c_force := 11.5
else if ((V[1] > -0.328)and(V[1] <= -0.325))  then
  c_force := -11.6
else if ((V[1] > 0.325)and(V[1] <= 0.328))  then
  c_force := 11.6
else if ((V[1] > -0.331)and(V[1] <= -0.328))  then
  c_force := -11.7
else if ((V[1] > 0.328)and(V[1] <= 0.331))  then
  c_force := 11.7
else if ((V[1] > -0.334)and(V[1] <= -0.331))  then
```

```
  c_force := -11.8
else if ((V[1] > 0.331)and(V[1] <= 0.334))  then
  c_force := 11.8
else if ((V[1] > -0.337)and(V[1] <= -0.334))  then
  c_force := -11.9
else if ((V[1] > 0.334)and(V[1] <= 0.337))  then
  c_force := 11.9
else if ((V[1] > -0.340)and(V[1] <= -0.337))  then
  c_force := -12.0
else if ((V[1] > 0.337)and(V[1] <= 0.340))  then
  c_force := 12.0
else if ((V[1] > -0.343)and(V[1] <= -0.340))  then
  c_force := -12.1
else if ((V[1] > 0.340)and(V[1] <= 0.343))  then
  c_force := 12.1
else if ((V[1] > -0.346)and(V[1] <= -0.343))  then
  c_force := -12.2
else if ((V[1] > 0.343)and(V[1] <= 0.346))  then
  c_force := 12.2
else if ((V[1] > -0.349)and(V[1] <= -0.346))  then
  c_force := -12.3
else if ((V[1] > 0.346)and(V[1] <= 0.349))  then
  c_force := 12.3
else if ((V[1] > -0.352)and(V[1] <= -0.349))  then
  c_force := -12.4
else if ((V[1] > 0.349)and(V[1] <= 0.352))  then
  c_force := 12.4
else if ((V[1] > -0.355)and(V[1] <= -0.352))  then
  c_force := -12.5
else if ((V[1] > 0.352)and(V[1] <= 0.355))  then
  c_force := 12.5
else if ((V[1] > -0.358)and(V[1] <= -0.355))  then
  c_force := -12.6
else if ((V[1] > 0.355)and(V[1] <= 0.358))  then
  c_force := 12.6
else if ((V[1] > -0.361)and(V[1] <= -0.358))  then
  c_force := -12.7
else if ((V[1] > 0.358)and(V[1] <= 0.361))  then
  c_force := 12.7
else if ((V[1] > -0.364)and(V[1] <= -0.361))  then
  c_force := -12.8
else if ((V[1] > 0.361)and(V[1] <= 0.364))  then
  c_force := 12.8
else if ((V[1] > -0.367)and(V[1] <= -0.364))  then
  c_force := -12.9
else if ((V[1] > 0.364)and(V[1] <= 0.367))  then
  c_force := 12.9
else if ((V[1] > -0.370)and(V[1] <= -0.367))  then
  c_force := -13.0
else if ((V[1] > 0.367)and(V[1] <= 0.370))  then
  c_force := 13.0
else if ((V[1] > -0.373)and(V[1] <= -0.370))  then
  c_force := -13.1
else if ((V[1] > 0.370)and(V[1] <= 0.373))  then
```

```
  c_force := 13.1
else if ((V[1] > -0.376)and(V[1] <= -0.373))  then
  c_force := -13.2
else if ((V[1] > 0.373)and(V[1] <= 0.376))  then
  c_force := 13.2
else if ((V[1] > -0.379)and(V[1] <= -0.376))  then
  c_force := -13.3
else if ((V[1] > 0.376)and(V[1] <= 0.379))  then
  c_force := 13.3
else if ((V[1] > -0.382)and(V[1] <= -0.379))  then
  c_force := -13.4
else if ((V[1] > 0.379)and(V[1] <= 0.382))  then
  c_force := 13.4
else if ((V[1] > -0.385)and(V[1] <= -0.382))  then
  c_force := -13.5
else if ((V[1] > 0.382)and(V[1] <= 0.385))  then
  c_force := 13.5
else if ((V[1] > -0.388)and(V[1] <= -0.385))  then
  c_force := -13.6
else if ((V[1] > 0.385)and(V[1] <= 0.388))  then
  c_force := 13.6
else if ((V[1] > -0.391)and(V[1] <= -0.388))  then
  c_force := -13.7
else if ((V[1] > 0.388)and(V[1] <= 0.391))  then
  c_force := 13.7
else if ((V[1] > -0.394)and(V[1] <= -0.391))  then
  c_force := -13.8
else if ((V[1] > 0.391)and(V[1] <= 0.394))  then
  c_force := 13.8
else if ((V[1] > -0.397)and(V[1] <= -0.394))  then
  c_force := -13.9
else if ((V[1] > 0.394)and(V[1] <= 0.397))  then
  c_force := 13.9
else if ((V[1] > -0.400)and(V[1] <= -0.397))  then
  c_force := -14.0
else if ((V[1] > 0.397)and(V[1] <= 0.400))  then
  c_force := 14.0
else if ((V[1] > -0.403)and(V[1] <= -0.400))  then
  c_force := -14.1
else if ((V[1] > 0.400)and(V[1] <= 0.403))  then
  c_force := 14.1
else if ((V[1] > -0.406)and(V[1] <= -0.403))  then
  c_force := -14.2
else if ((V[1] > 0.403)and(V[1] <= 0.406))  then
  c_force := 14.2
else if ((V[1] > -0.409)and(V[1] <= -0.406))  then
  c_force := -14.3
else if ((V[1] > 0.406)and(V[1] <= 0.409))  then
  c_force := 14.3
else if ((V[1] > -0.412)and(V[1] <= -0.409))  then
  c_force := -14.4
else if ((V[1] > 0.409)and(V[1] <= 0.412))  then
  c_force := 14.4
else if ((V[1] > -0.415)and(V[1] <= -0.412))  then
```

```
    c_force := -14.5
  else if ((V[1] > 0.412)and(V[1] <= 0.415))   then
    c_force := 14.5
  else if ((V[1] > -0.418)and(V[1] <= -0.415))   then
    c_force := -14.6
  else if ((V[1] > 0.415)and(V[1] <= 0.418))   then
    c_force := 14.6
  else if ((V[1] > -0.421)and(V[1] <= -0.418))   then
    c_force := -14.7
  else if ((V[1] > 0.418)and(V[1] <= 0.421))   then
    c_force := 14.7
  else if ((V[1] > -0.424)and(V[1] <= -0.421))   then
    c_force := -14.8
  else if ((V[1] > 0.421)and(V[1] <= 0.424))   then
    c_force := 14.8
  else if ((V[1] > -0.427)and(V[1] <= -0.424))   then
    c_force := -14.9
  else if ((V[1] > 0.424)and(V[1] <= 0.427))   then
    c_force := 14.9

  else if ((V[1] < -0.427) or (V[1] > 0.427))  then
  begin
  {    c_force := 0 + in_force;}
    c_force := 15;
    if (V[1] > 0.0) then
    begin                          {the pole angle is positive going down}
       if c_force < 0.0 then       {force currently is going left}
         c_force := c_force*(-1);    {change direction}
     end
    else if (V[1] < 0.0) then         {the pole angle is negative going down}
     begin
       if c_force > 0.0 then        {force currently is going right}
          c_force := c_force*(-1)     {change direction}
     end;
  end;                          {theta is not zero}
  if mt < 0.6 then                {reduce force since mass is very small}
    c_force := c_force*0.2         {reduce size of experiment}
  else if mt < 1.0 then
    c_force := c_force*0.5
    else if mt > 1.1 then
      c_force := c_force*mt
end;  {check_pole_angle}


{*** procedure to find the elastic pole velocity and acceleration **}
procedure find_elas_pole_acc_vel_ang(k:real;var e_acc,e_vel,t_angle:real);
var n1,n2,n3:real;
begin
    e_vel := V[2] + V[2]*k*cos(V[1])/(1 + k*sin(V[1])*k*sin(V[1]));
    n1 := 1 + (k*cos(V[1])/(1 + k*sin(V[1])*k*sin(V[1])));
    n2 := -k*sin(V[1])*(1 + k*k*(1 + cos(V[1])*cos(V[1])));
    n3 := (1 + (k*sin(V[1])*k*sin(V[1])))*(1 + (k*sin(V[1])*k*sin(V[1])));
    e_acc := F[2]*n1 + V[2]*V[2]*(n2/n3);
end;  {find_elas_pole_acc_vel_angle}
```

```
{*** procedure to find cart acceleration **}
procedure find_cart_acceleration(ch:char);
var k,e_acc,e_vel,n1,n2,n3,t_angle,c_dis :real;
begin
   n1 := c_f*mp*g;
   if ch = '2' then        {elastic pole}
   begin
    k := (12*pl*pl*mp*g)/(8*E*pb*pd*pd*pd);
    find_elas_pole_acc_vel_ang(k,e_acc,e_vel,t_angle);
    t_angle := V[1] + arctan(k*sin(V[1]));    {total elastic angle in radians}
    n2 := e_acc*mp*pl/2*(cos(t_angle)-c_f*sin(t_angle));
    n3 := e_vel*e_vel*mp*pl/2*(-sin(t_angle)-c_f*cos(t_angle));
    c_ac[ctr] := (c_force-(n1+n2+n3))/mt;
    writeln(f2,T:10:6,' ',t_angle*57.2957795:10:6,' ',V[1]*57.2957795:10:6,' ',c_ac[ctr]:10:6);
   end
   else  {rigid pole}
   begin
    n2 := F[2]*mp*pl/2*(cos(V[1])-c_f*sin(V[1]));
    n3 := V[2]*V[2]*mp*pl/2*(-sin(V[1])-c_f*cos(V[1]));
    c_ac[ctr] := (c_force-(n1+n2+n3))/mt;
   end;
end; {find_cart_acceleration}

{*** proceudre to solve differential equation using numerical integration*}
procedure numerical_integration(ch:char);
var t_angle:real;   {total elastic angle}
begin  {main}
   in_data(ch);
   failure := false;
   case ch of
    '1' : assign(f1,'c:\research\n_i\r_out.dat'); {rigid pole}
    '2' : begin
          assign(f1,'c:\research\n_i\e_out.dat'); {elastic pole}
          assign(f2,'c:\research\n_i\r_angle.dat');
          rewrite(f2);
          write_init_values_to_file2;
          writeln(f2,'Time':8,'Elastic pole':15,'Rigid pole':12,'Cart':7);
          writeln(f2,'angle':20,'angle':12,'acceleration':18);
          t_angle := V[1] + arctan(sin(V[1])*(12*pl*pl*mp*g)/(8*E*pb*pd*pd*pd));    {total elastic
angle}
          writeln(f2,T:10:6,' ',t_angle*57.2957795:10:6,' ',V[1]*57.2957795:10:6,' ',c_ac[1]:10:6);
         end;
   end;  {case}
   rewrite(f1);
   out_file;            {print data to file}
   writeln('Time':8,'C_accel':13,'Pole_ang':13,'Pole_ang_vel':14,'Force':7,'P_accel':14);
   write(T:10:6,' ',c_ac[1]:10:6,' ',V[1]*57.2957795:10:6,' ',V[2]:10:6,' ',in_force:10:6);
   repeat
         m := 0;
         runge(M,K,2,V,F,T,H);
         while k = 1 do
         begin
            F[1] := V[2];
```

```
            check_pole_angle;
            if failure then
            begin
              close(f1); close(f2);
              exit;
            end;
            dnume := (4/3*mt*pl/2 - mp*pl/2*cos(V[1])*cos(V[1]) + c_f*mp*pl/2*cos(V[1])*sin(V[1]));
            nume := mt*g*sin(V[1]) - cos(V[1])*(c_force - (c_f*mp*g -
mp*pl/2*V[2]*V[2]*(sin(V[1])+c_f*cos(V[1]))));
            F[2] := nume/dnume;
            if T = 0.0 then
            begin
              writeln(' ',F[2]:10:6);
              writeln(f1,' ',F[2]:10:6);
            end;
            runge(M,K,2,V,F,T,H);
          end;
          if (T <= tmax) then
          begin
            icount := icount + 1;
            if (icount = ifreq) then
            begin
              icount := 0;
              ctr := ctr+1;  {total number of elements in array}
              find_cart_acceleration(ch);   {finds elastic angle,velocity,accelaration,displacemen}
writeln(T:10:6,' ',c_ac[ctr]:10:6,' ',V[1]*57.295779:10:6,' ',V[2]:10:6,' ',c_force:10:6,' ',F[2]:10:6);
writeln(f1,T:10:6,' ',c_ac[ctr]:10:6,' ',V[1]*57.295779:10:6,' ',V[2]:10:6,' ',c_force:10:6,' ',F[2]:10:6);
              time_co[ctr] := T; angle_co[ctr] := V[1];
            end;
          end;
    until (T >tmax);
    writeln;
    write('     Thank you for waiting. Just press enter to continue ... ');
    readln;
    close(f1);
    if ch <> '1' then close(f2);
end;  {numerical_integration}

{this is the menu to choosr topic on the program}
procedure menu(var ch:char);
VAR  y:integer;
begin
    clrscr; gotoxy(1,3);
    textcolor(blue);
    writeln('*************************************************':63);
    textcolor(red+blinK);
    writeln('*        CART-POLE BALANCING SYSTEM        *':63);
    textcolor(blue);
    writeln('*        Computer simulation using         *':63);
    WRITELN('*        Fourth order runge-kutta          *':63);
    textcolor(magenta+blink);
    writeln('*           By: Elmer P. Dadios            *':63);
    textcolor(blue);
    writeln('*************************************************':63);
```

```
    writeln;
    textcolor(green);
    writeln('*************************************************':64);
    writeln('*              MAIN MENU                  *':64);
    writeln('* 1. Simulate new values for rigid pole       *':64);
    writeln('* 2. Simulate new values for elastic pole     *':64);
{   writeln('* 3. Plot the graph: rigid pole without friction *':64);}
    writeln('* 4. Plot the graph: rigid pole angle vs. time  *':64);
    writeln('* 5. Plot the graph: elastic pole angle vs.time  *':64);
    writeln('* 6. Real time elastic pole cart simulation     *':64);
    writeln('* 7. Cart displacement vs. time (elastic pole)  *':64);
    writeln('* 8. Cart acceleration vs. time (elastic pole)  *':64);
    writeln('* 9. Cart velocity vs. time    (elastic pole)  *':64);
    writeln('* 0. To quit program.                          *':64);
    writeln('*                                    *':64);
    writeln('*************************************************':64);
    y := wherey;
    repeat
      gotoxy(27,y);
      write('Your choice please : ');
      read(ch);
    until ch in ['0','1','2','4','5','6','7','8','9'];
end;   {menu}

{*** procedure to find the elastic angle ****}
procedure find_elastic_angle;
var k,t_angle :real;
   i  :integer;
begin
    assign(f1,'c:\research\n_i\r_angle.dat');
    rewrite(f1);
    write_init_values_to_file;
    writeln(f1,'Time':8,'t_pole_ang':15,'r_pole_ang':12,'C_accel':10);
    k := pl*pl*mp*g*12/(8*E*pb*pd*pd*pd);
    for i := 1 to ctr do
    begin
        t_angle := angle_co[i] + arctan(k*sin(angle_co[i]));
        writeln(f1,time_co[i]:10:6,'  ',t_angle:10:6,'  ',angle_co[i]:10:6,'  ',c_ac[i]:10:6);
    end;
    close(f1);
end;   {find_elastic_angle}

BEGIN      {MAIN PROGRAM }
  repeat
    MENU(ch);
    case ch of
      '1' : numerical_integration(ch);
      '2' : numerical_integration(ch);
      '4','5','7','8','9': start_g_r_wof(ch);   {call program that will graph theta vs time}
      '6' : draw_pole_cart('6');
    end;      {case}
  until ch = '0';
  textcolor(white);
end.      {MAIN PROGRM}
```

```
/*** This is the file that contain the procedures for displaying graphically the ***/
                    /*** dynamic behaviour of the system ***/
unit gra28_car;  {graph of moving pole and cart}
INTERFACE
uses crt,graph,glob_dat;
type fillpaterrnType = array[1..8] of byte;
const  gray50: fillpatterntype = ($AA,$55,$AA,$55,$AA,$55,$AA,$55);
var
  grdriver,grmode,errcode,x,y,i,time_sc,j    :integer;
  scale :longint;
  f1 : text;
  t_angle :array[1..n] of real;
  failure :boolean;
  in_force,c_f,ac_amp :real;

procedure get_pole_data(var data:real);
procedure get_freq_neg(i:integer;var t1,t2:real;var cycles:integer);
procedure solve_cart_displacement(ch:char);
procedure get_ex_data(choice:char);
procedure plot_line(choice:char);
procedure write_theta(the_pos,the_neg:real);
procedure write_time(time:integer);
procedure plot_points;
procedure init_graph;
procedure write_strings(ch:char);
procedure start_g_r_wof(choice:char);
procedure write_cart_pole_heading;
procedure draw_track;
procedure draw_cart(change:real);
procedure draw_hinge(change:real);
procedure draw_elastic_pole(L1,change:real;n:integer;var TL:real);
procedure draw_system_at_any_time(n,scale:integer;var change:real);
procedure draw_pole(change:real);
procedure draw_pole_cart(ch:char);

IMPLEMENTATION

{*** procedure to initialize global data ***}
procedure init_global_data;
begin
   for i := 1 to n do
   begin
      time_co[i] := 0.0;
      t_angle[i] := 0.0;
      angle_co[i] := 0.0;
      c_ac[i]  := 0.0;
   end;
end;    {init_global_data}
{*** procedure to read poles data from file ***}
procedure get_pole_data(var data:real);
var ch:char;
begin
   repeat
```

```
      read(f1,ch);
    until ch = '=';
    readln(f1,data);
end;  {get_pole_data}

{*** procedure to get frequency that start at the first negative side ***}
procedure get_freq_neg(i:integer;var t1,t2:real;var cycles:integer);
begin
    cycles := cycles + 1;
    if cycles = 1 then t2 := 0.0
    else t2 := t2 + (time_co[i]-t1);
    t1 := time_co[i];
end;  {get_freq_neg}

{*** procedure to find the cart displacement and velocity ***}
procedure solve_cart_displacement(ch:char);
var t1,t2,freq,cart_vel:real;
   i,cycles:integer;
   flag :boolean;
begin
   assign(f1,'c:\research\n_i\c_dis.dat');
   rewrite(f1);
   writeln(f1,'Time':8,'Elas_angle':15,'Cart_accel':12,'Cart_velocity':14,'Cart_displacement':18);
   writeln(f1,time_co[1]:10:6,'  ',t_angle[1]*57.2957795:10:6,'  ',c_ac[1]:10:6,'  ',0.0:10:6,'    ',0.0:10:6);
   t1 := 0.0; cycles := 0;
   if in_force > 0.0 then
   begin
      flag := true;
      for i := 1 to j do
      begin
         if (flag) and (c_ac[i] < 0.0) then   {first negative}
         begin
            get_freq_neg(i,t1,t2,cycles);
            flag := false;
         end
         else if (not flag) and (c_ac[i] > 0.0) then {first positive}
         begin
            t2 := t2 + (time_co[i]-t1);
            t1 := time_co[i];
            flag := true;
         end;
      end;    {for}
   end        {if in_force > 0.0}
   else
   begin
      flag := false;
      for i := 1 to j do
      begin
         if (not flag) and (c_ac[i] > 0.0) then   {first negative}
         begin
            get_freq_neg(i,t1,t2,cycles);
            flag := true;
         end
         else if (flag) and (c_ac[i] < 0.0) then {first positive}
```

```
        begin
          t2 := t2 + (time_co[i]-t1);
          t1 := time_co[i];
          flag := false;
        end;
    end;    {for}
  end;      {if in_force < 0.0}
  freq := t2/cycles;
  for i := 2 to j do        {at t > 0}
  begin                     {values for cart displacement}
    cart_vel := ac_amp*sin(6.283185*freq*time_co[i])/(6.283185*freq);  {Ksin(wt)/w = cart velocity}
    write(f1,time_co[i]:10:6,' ',t_angle[i]*57.2957795:10:6,' ',c_ac[i]:10:6,' ',cart_vel:10:6);
    if ch = '6' then        {draw the entire system}
    begin
      c_ac[i] := -c_ac[i]/(freq*freq*39.4784);  {cart displacement = -acceleration/omega*omega}
      writeln(f1,'   ',c_ac[i]:10:6);
    end
    else if ch = '7' then       {plot cart displacement vs time}
    begin
      angle_co[i] := -c_ac[i]/(freq*freq*39.4784);
      writeln(f1,' ',angle_co[i]:10:6);
    end
    else if ch = '9' then       {plot cart velocity vs time}
    begin
      angle_co[i] := cart_vel;  {in plotting points angle_co is always used}
      writeln(f1);
    end;
  end;   {for}
  if ch = '6' then c_ac[1] := 0.0;   {initial displacement in plotting the entire system}
  if (ch = '9')or(ch='7') then
    angle_co[1] := 0.0;          {initial velocity and displacement}
  close(f1);
end;   {solve_cart_displacement}

{*** procedure to read data from external file ***}
procedure get_ex_data(choice:char);
var ch :char;
  amp_p,amp_n:real;       {minimimum and maximum value of amplitude}
begin
   init_global_data;
   case choice of
      '3','4': assign (f1,'c:\research\n_i\r_out.dat');  {rigid pole w/o friction}
{       '4': assign (f1,'c:\research\n_i\r_angle.dat');}  {rigid pole w/ friction}
      '5','6','7','8','9': assign (f1,'c:\research\n_i\r_angle.dat');  {rigid pole w/ friction}
   end;
   reset(f1);
{   for i := 1 to 20 do   }
   for i := 1 to 2 do
     readln(f1);
     get_pole_data(mp);          {mass of the pole}
     get_pole_data(mt);          {total mass of the pole and the cart}
     get_pole_data(pl);          {length of the pole}
     get_pole_data(pb);          {breadth of the pole}
     get_pole_data(pd);          {depth of the pole}
```

```
    get_pole_data(E);          {elasticity of the pole}
    get_pole_data(in_force);
    get_pole_data(c_f);
    for i := 1 to 11 do
     readln(f1);
  j := 0; amp_p := 0; amp_n := 0;
  while (not eof(f1)) do
  begin
   j := j+1;
   for i := 1 to 2 do
    read(f1,ch);
   read(f1,time_co[j]);     {the time}
   for i := 1 to 2 do
    read(f1,ch);
   read(f1,t_angle[j]);              {the total elastic angle}
   t_angle[j] := t_angle[j]/57.2957795;   {convert to radians}
   for i := 1 to 2 do
    read(f1,ch);
   read(f1,angle_co[j]);             {the rigid pole angle}
   angle_co[j] := angle_co[j]/57.2957795;
   for i := 1 to 2 do
    read(f1,ch);
   readln(f1,c_ac[j]);       {cart acceleration}
   if (j>2) then
   begin       {exclude at t=0}
     if (c_ac[j] < amp_n) then
       amp_n := c_ac[j]      {max negative amplitude of acceleration}
     else if (c_ac[j] > amp_p) then
       amp_p := c_ac[j];     {max positive amplitude of acceleration}
   end;
  end;     {while}
  ac_amp := (amp_p + abs(amp_n))/2;   {actual amplitude of acceleration}
  close(f1);
  if (choice='6') or (choice='7') or (choice='9') then     {this is for real time cart pole simulation}
      solve_cart_displacement(choice)
  else if choice = '8' then     {for cart acceleration}
   for i := 1 to j do
      angle_co[i] := c_ac[i]
  else if choice = '5' then     {this is for elastic pole angle_co = total deflection}
     for i := 1 to j do
      angle_co[i] := t_angle[i];
end;     {get_ex_data}

{**** procedure to draw center line ****}
procedure plot_line(choice:char);
begin
   x := round(getmaxx/2-270);
   y := round(getmaxy/2);
   for i := x to getmaxx-10 do
     Putpixel(i,y,blue);              {horizontal line}
   for i := 5 to y*2-5 do
     putpixel(x-1,i,blue);            {vertical line}
   setcolor(brown);
   SetTextstyle(defaultFont,Horizdir,2);  {charsize =1}
```

```
  SetTextjustify(centertext,centertext);
  case choice of
   '4':begin
       if c_f = 0.0 then
         OutTextXY(x+290,y-200,'RIGID POLE WITHOUT FRICTION') {THE TITLE}
       else
         OutTextXY(x+290,y-200,'RIGID POLE WITH FRICTION'); {THE TITLE}
      end;
   '5':begin
       if c_f <> 0.0 then
         OutTextXY(x+290,y-200,'ELASTIC POLE WITH FRICTION') {THE TITLE}
       else
         OutTextXY(x+290,y-200,'ELASTIC POLE WITHOUT FRICTION');
      end;
   '7': OutTextXY(x+290,y-200,'TIME VS. CART DISPLACEMENT'); {THE TITLE}
   '8': OutTextXY(x+290,y-200,'TIME VS. CART ACCELERATION'); {THE TITLE}
   '9': OutTextXY(x+290,y-200,'TIME VS. CART VELOCITY'); {THE TITLE}
  END;
end; {plot_line}

{*** procedure to plot the coordinates of the anlge ***}
procedure write_theta(the_pos,the_neg:real);
begin
  x := round(getmaxx/2-270);
  {*** for positive angle ****}
  y := round(scale*the_pos);
  y := round(getmaxy/2 - y);                {coordinate of theta}
  for i := (x-1) to (x+1) do
    putpixel(i,y,green);                    {draw horizintal line}
  x := x-30;
    if the_pos = 0.0 then
      OutTextXY(x,y,'0.0')
    else if the_pos <= 0.00175 then
      OutTextXY(x,y,'0.0018')
    else if the_pos <= 0.00345 then
      OutTextXY(x,y,'0.0035')
    else if the_pos <=0.005232 then
      OutTextXY(x,y,'0.0052')
    else if the_pos <= 0.00698 then
      OutTextXY(x,y,'0.007')
    else if the_pos <= 0.00873 then
      OutTextXY(x,y,'0.0087')
    else if the_pos <= 0.01047 then
      OutTextXY(x,y,'0.0105')
    else if the_pos <= 0.01220 then
      OutTextXY(x,y,'0.0122')
    else if the_pos <= 0.01396 then
      OutTextXY(x,y,'0.014')
    else if the_pos <= 0.01570 then
      OutTextXY(x,y,'0.0157')
    else if the_pos <= 0.01745 then
      OutTextXY(x,y,'0.0175')
    else if the_pos <= 0.01919 then
      OutTextXY(x,y,'0.0192')
```

```
else if the_pos <= 0.02090 then
  OutTextXY(x,y,'0.0209')
else if the_pos <= 0.02269 then
  OutTextXY(x,y,'0.0227')
else if the_pos <= 0.02443 then
  OutTextXY(x,y,'0.0244')
else if the_pos <= 0.02618 then
  OutTextXY(x,y,'0.0262')
else if the_pos <= 0.02793 then
  OutTextXY(x,y,'0.0279')
else if the_pos <= 0.02967 then
  OutTextXY(x,y,'0.0297')
else if the_pos <= 0.03142 then
  OutTextXY(x,y,'0.0314')
else if the_pos <= 0.03316 then
  OutTextXY(x,y,'0.0332')
else if the_pos <= 0.03491 then
  OutTextXY(x,y,'0.0349')
else if the_pos <= 0.03665 then
  OutTextXY(x,y,'0.0367')
else if the_pos <= 0.03839 then
  OutTextXY(x,y,'0.0384')
else if the_pos <= 0.04014 then
  OutTextXY(x,y,'0.0401')
else if the_pos <= 0.04189 then
  OutTextXY(x,y,'0.0419')
else if the_pos <= 0.04363 then
  OutTextXY(x,y,'0.0436')
else if the_pos <= 0.04538 then
  OutTextXY(x,y,'0.0454')
else if the_pos <= 0.04712 then
  OutTextXY(x,y,'0.0471')
else if the_pos <= 0.04887 then
  OutTextXY(x,y,'0.0489')
else if the_pos <= 0.05061 then
  OutTextXY(x,y,'0.0506')
else if the_pos <= 0.05236 then
  OutTextXY(x,y,'0.0524')
else if the_pos <= 0.05410 then
  OutTextXY(x,y,'0.0541')
else if the_pos <= 0.05585 then
  OutTextXY(x,y,'0.0559')
else if the_pos <= 0.05760 then
  OutTextXY(x,y,'0.0576')
else if the_pos <= 0.05934 then
  OutTextXY(x,y,'0.0593')
else if the_pos <= 0.06100 then
  OutTextXY(x,y,'0.061')
else if the_pos <= 0.06283 then
  OutTextXY(x,y,'0.0628')
else if the_pos <= 0.06458 then
  OutTextXY(x,y,'0.0646')
else if the_pos <= 0.06632 then
  OutTextXY(x,y,'0.0663')
```

```
else if the_pos <= 0.06807 then
  OutTextXY(x,y,'0.0681')
else if the_pos <= 0.06981 then
  OutTextXY(x,y,'0.0698')
else if the_pos <= 0.07156 then
  OutTextXY(x,y,'0.0716')
else if the_pos <= 0.07330 then
  OutTextXY(x,y,'0.0733')
else if the_pos <= 0.07500 then
  OutTextXY(x,y,'0.075')
else if the_pos <= 0.07679 then
  OutTextXY(x,y,'0.0768')
else if the_pos <= 0.07853 then
  OutTextXY(x,y,'0.0785')
else if the_pos <= 0.08028 then
  OutTextXY(x,y,'0.0803')
else if the_pos <= 0.08203 then
  OutTextXY(x,y,'0.082')
else if the_pos <= 0.08378 then
  OutTextXY(x,y,'0.0838')
else if the_pos <= 0.08552 then
  OutTextXY(x,y,'0.0855')
else if the_pos <= 0.087273 then
  OutTextXY(x,y,'0.0873')
else if the_pos <= 0.08901 then
  OutTextXY(x,y,'0.089')
else if the_pos <= 0.09076 then
  OutTextXY(x,y,'0.0908')
else if the_pos <= 0.09250 then
  OutTextXY(x,y,'0.0925')
else if the_pos <= 0.09425 then
  OutTextXY(x,y,'0.0943')
else if the_pos <= 0.09599 then
  OutTextXY(x,y,'0.096')
else if the_pos <= 0.09777 then
  OutTextXY(x,y,'0.0978')
else if the_pos <= 0.09948 then
  OutTextXY(x,y,'0.0995')
else if the_pos <= 0.10123 then
  OutTextXY(x,y,'0.1012')
else if the_pos <= 0.10297 then
  OutTextXY(x,y,'0.103')
else if the_pos <= 0.10472 then
  OutTextXY(x,y,'0.1047')
else if the_pos <= 0.10821 then
  OutTextXY(x,y,'0.1082')
else if the_pos <= 0.11170 then
  OutTextXY(x,y,'0.1117')
else if the_pos <= 0.11519 then
  OutTextXY(x,y,'0.1152')
else if the_pos <= 0.11868 then
  OutTextXY(x,y,'0.1187')
else if the_pos <= 0.12217 then
  OutTextXY(x,y,'0.1222')
```

```
else if the_pos <= 0.12566 then
  OutTextXY(x,y,'0.1257')
else if the_pos <= 0.12915 then
  OutTextXY(x,y,'0.1292')
else if the_pos <= 0.13264 then
  OutTextXY(x,y,'0.1326')
else if the_pos <= 0.13614 then
  OutTextXY(x,y,'0.1361')
else if the_pos <= 0.13962 then
  OutTextXY(x,y,'0.1396')
else if the_pos <= 0.14312 then
  OutTextXY(x,y,'0.1431')
else if the_pos <= 0.14661 then
  OutTextXY(x,y,'0.1466')
else if the_pos <= 0.15010 then
  OutTextXY(x,y,'0.1501')
else if the_pos <= 0.153689 then
  OutTextXY(x,y,'0.1537')
else if the_pos <= 0.15708 then
  OutTextXY(x,y,'0.1571')
else if the_pos <= 0.16057 then
  OutTextXY(x,y,'0.1606')
else if the_pos <= 0.16406 then
  OutTextXY(x,y,'0.1641')
else if the_pos <= 0.16755 then
  OutTextXY(x,y,'0.1676')
else if the_pos <= 0.17104 then
  OutTextXY(x,y,'0.171')
else if the_pos <= 0.17453 then
  OutTextXY(x,y,'0.1745')
else if the_pos <= 0.191986 then
  OutTextXY(x,y,'0.192')
else if the_pos <= 0.2094395 then
  OutTextXY(x,y,'0.2094')
else if the_pos <= 0.226892 then
  OutTextXY(x,y,'0.2269')
else if the_pos <= 0.24435 then
  OutTextXY(x,y,'0.2444')
else if the_pos <= 0.26180 then
  OutTextXY(x,y,'0.2618')
else if the_pos <= 0.27925 then
  OutTextXY(x,y,'0.2793')
else if the_pos <= 0.29671 then
  OutTextXY(x,y,'0.2967')
else if the_pos <= 0.31416 then
  OutTextXY(x,y,'0.3142')
else if the_pos <= 0.33161 then
  OutTextXY(x,y,'0.3316')
else if the_pos <= 0.34907 then
  OutTextXY(x,y,'0.3491')
else if the_pos <= 0.36652 then
  OutTextXY(x,y,'0.3665')
else if the_pos <= 0.38397 then
  OutTextXY(x,y,'0.384')
```

```
else if the_pos <= 0.40143 then
  OutTextXY(x,y,'0.4014')
else if the_pos <= 0.41888 then
  OutTextXY(x,y,'0.4189')
else if the_pos <= 0.43633 then
  OutTextXY(x,y,'0.4363')
else if the_pos <= 0.453786 then
  OutTextXY(x,y,'0.4538')
else if the_pos <= 0.47124 then
  OutTextXY(x,y,'0.4712')
else if the_pos <= 0.48870 then
  OutTextXY(x,y,'0.4887')
else if the_pos <= 0.506145 then
  OutTextXY(x,y,'0.5061')
else if the_pos <= 0.523599 then
  OutTextXY(x,y,'0.5236');

 {*** for negative ****}
x := x+30;                          {for horizontal line marker}
y := round(scale*the_neg);
y := round(abs(y) + getmaxy/2);          {coordinate of theta}
for i := (x-1) to (x+1) do
putpixel(i,y,green);                {draw horizintal line}
x := x-30;                          {for theta coordinate}
if the_neg = 0.0 then
  OutTextXY(x,y,'0.0')
else if the_neg >= -0.00175 then
  OutTextXY(x,y,' 0.0018')
else if the_neg >= -0.00345 then
  OutTextXY(x,y,' 0.0035')
else if the_neg >= -0.00523 then
  OutTextXY(x,y,'0.0052')
else if the_neg >= -0.00698 then
  OutTextXY(x,y,'0.007')
else if the_neg >= -0.00873 then
  OutTextXY(x,y,'0.0087')
else if the_neg >= -0.01047 then
  OutTextXY(x,y,'0.0105')
else if the_neg >= -0.01220 then
  OutTextXY(x,y,'0.0122')
else if the_neg >= -0.01396 then
  OutTextXY(x,y,'0.014')
else if the_neg >= -0.01570 then
  OutTextXY(x,y,'0.0157')
else if the_neg >= -0.01745 then
  OutTextXY(x,y,'0.0175')
else if the_neg >= -0.01919 then
  OutTextXY(x,y,'0.0192')
else if the_neg >= -0.02090 then
  OutTextXY(x,y,'0.0209')
else if the_neg >= -0.02269 then
  OutTextXY(x,y,'0.0227')
else if the_neg >= -0.02443 then
  OutTextXY(x,y,'0.0244')
```

```
else if the_neg >= -0.02618 then
  OutTextXY(x,y,'0.0262')
else if the_neg >= -0.02793 then
  OutTextXY(x,y,'0.0279')
else if the_neg >= -0.02967 then
  OutTextXY(x,y,'0.0297')
else if the_neg >= -0.03142 then
  OutTextXY(x,y,'0.0314')
else if the_neg >= -0.03316 then
  OutTextXY(x,y,'0.0332')
else if the_neg >= -0.03491 then
  OutTextXY(x,y,'0.0349')
else if the_neg >= -0.03665 then
  OutTextXY(x,y,'0.0367')
else if the_neg >= -0.03839 then
  OutTextXY(x,y,'0.0384')
else if the_neg >= -0.04014 then
  OutTextXY(x,y,'0.0401')
else if the_neg >= -0.04189 then
  OutTextXY(x,y,'0.0419')
else if the_neg >= -0.04363 then
  OutTextXY(x,y,'0.0436')
else if the_neg >= -0.04538 then
  OutTextXY(x,y,'0.0454')
else if the_neg >= -0.04712 then
  OutTextXY(x,y,'0.0471')
else if the_neg >= -0.04887 then
  OutTextXY(x,y,'0.0489')
else if the_neg >= -0.05061 then
  OutTextXY(x,y,'0.0506')
else if the_neg >= -0.05236 then
  OutTextXY(x,y,'0.0524')
else if the_neg >= -0.05410 then
  OutTextXY(x,y,'0.0541')
else if the_neg >= -0.05585 then
  OutTextXY(x,y,'0.0559')
else if the_neg >= -0.05760 then
  OutTextXY(x,y,'0.0576')
else if the_neg >= -0.05934 then
  OutTextXY(x,y,'0.0593')
else if the_neg >= -0.06100 then
  OutTextXY(x,y,'0.061')
else if the_neg >= -0.06283 then
  OutTextXY(x,y,'0.0628')
else if the_neg >= -0.06458 then
  OutTextXY(x,y,'0.0646')
else if the_neg >= -0.06632 then
  OutTextXY(x,y,'0.0663')
else if the_neg >= -0.06807 then
  OutTextXY(x,y,'0.0681')
else if the_neg >= -0.06981 then
  OutTextXY(x,y,'0.0698')
else if the_neg >= -0.07156 then
  OutTextXY(x,y,'0.0716')
```

```
else if the_neg >= -0.07330 then
  OutTextXY(x,y,'0.0733')
else if the_neg >= -0.07500 then
  OutTextXY(x,y,'0.075')
else if the_neg >= -0.07679 then
  OutTextXY(x,y,'0.0768')
else if the_neg >= -0.07853 then
  OutTextXY(x,y,'0.0785')
else if the_neg >= -0.08028 then
  OutTextXY(x,y,'0.0803')
else if the_neg >= -0.08203 then
  OutTextXY(x,y,'0.082')
else if the_neg >= -0.08378 then
  OutTextXY(x,y,'0.0838')
else if the_neg >= -0.08552 then
  OutTextXY(x,y,'0.0855')
else if the_neg >= -0.087273 then
  OutTextXY(x,y,'0.0873')
else if the_neg >= -0.08901 then
  OutTextXY(x,y,'0.089')
else if the_neg >= -0.09076 then
  OutTextXY(x,y,'0.0908')
else if the_neg >= -0.09250 then
  OutTextXY(x,y,'0.0925')
else if the_neg >= -0.09425 then
  OutTextXY(x,y,'0.0943')
else if the_neg >= -0.09599 then
  OutTextXY(x,y,'0.096')
else if the_neg >= -0.09777 then
  OutTextXY(x,y,'0.0978')
else if the_neg >= -0.09948 then
  OutTextXY(x,y,'0.0995')
else if the_neg >= -0.10123 then
  OutTextXY(x,y,'0.1012')
else if the_neg >= -0.10297 then
  OutTextXY(x,y,'0.103')
else if the_neg >= -0.10472 then
  OutTextXY(x,y,'0.1047')
else if the_neg >= -0.10821 then
  OutTextXY(x,y,'0.1082')
else if the_neg >= -0.11170 then
  OutTextXY(x,y,'0.1117')
else if the_neg >= -0.11519 then
  OutTextXY(x,y,'0.1152')
else if the_neg >= -0.11868 then
  OutTextXY(x,y,'0.1187')
else if the_neg >= -0.12217 then
  OutTextXY(x,y,'0.1222')
else if the_neg >= -0.12566 then
  OutTextXY(x,y,'0.1257')
else if the_neg >= -0.12915 then
  OutTextXY(x,y,'0.1292')
else if the_neg >= -0.13264 then
  OutTextXY(x,y,'0.1326')
```

```
else if the_neg >= -0.13614  then
  OutTextXY(x,y,'0.1361')
else if the_neg >= -0.13962  then
  OutTextXY(x,y,'0.1396')
else if the_neg >= -0.14312  then
  OutTextXY(x,y,'0.1431')
else if the_neg >= -0.14661  then
  OutTextXY(x,y,'0.1466')
else if the_neg >= -0.15010  then
  OutTextXY(x,y,'0.1501')
else if the_neg >= -0.153689 then
  OutTextXY(x,y,'0.1537')
else if the_neg >= -0.15708  then
  OutTextXY(x,y,'0.1571')
else if the_neg >= -0.16057  then
  OutTextXY(x,y,'0.1606')
else if the_neg >= -0.16406  then
  OutTextXY(x,y,'0.1641')
else if the_neg >= -0.16755  then
  OutTextXY(x,y,'0.1676')
else if the_neg >= -0.17104  then
  OutTextXY(x,y,'0.171')
else if the_neg >= -0.17453  then
  OutTextXY(x,y,'0.1745')
else if the_neg >= -0.191986 then
  OutTextXY(x,y,'0.192')
else if the_neg >= -0.2094395 then
  OutTextXY(x,y,'0.2094')
else if the_neg >= -0.226892 then
  OutTextXY(x,y,'0.2269')
else if the_neg >= -0.24435  then
  OutTextXY(x,y,'0.2444')
else if the_neg >= -0.26180  then
  OutTextXY(x,y,'0.2618')
else if the_neg >= -0.27925  then
  OutTextXY(x,y,'0.2793')
else if the_neg >= -0.29671  then
  OutTextXY(x,y,'0.2967')
else if the_neg >= -0.31416  then
  OutTextXY(x,y,'0.3142')
else if the_neg >= -0.33161  then
  OutTextXY(x,y,'0.3316')
else if the_neg >= -0.34907  then
  OutTextXY(x,y,'0.3491')
else if the_neg >= -0.36652  then
  OutTextXY(x,y,'0.3665')
else if the_neg >= -0.38397  then
  OutTextXY(x,y,'0.384')
else if the_neg >= -0.40143  then
  OutTextXY(x,y,'0.4014')
else if the_neg >= -0.41888  then
  OutTextXY(x,y,'0.4189')
else if the_neg >= -0.43633  then
  OutTextXY(x,y,'0.4363')
```

```
      else if the_neg >= -0.453786 then
        OutTextXY(x,y,'0.4538')
      else if the_neg >= -0.47124  then
        OutTextXY(x,y,'0.4712')
      else if the_neg >= -0.48870  then
        OutTextXY(x,y,'0.4887')
      else if the_neg >= -0.506145 then
        OutTextXY(x,y,'0.5061')
      else if the_neg >= -0.523599 then
        OutTextXY(x,y,'0.5236');

end;    {write_theta}

{*** procedure to print time covered ***}
procedure write_time(time:integer);
var l,xt,yt:integer;
begin
    xt := round(getmaxx/2-270+time*time_sc);        {x coordinate of time}
    yt := round(getmaxy/2+5);                       {y coordinate of time}
    for l := round(getmaxy/2-1) to round(getmaxy/2+1) do
              putpixel(xt,l,green);
    setcolor(yellow);
    SetTextstyle(defaultfont,horizdir,1);           {charsize =1}
    SetTextjustify(LeftText,Toptext);
     case time of
        1: OutTextXY(xt-2,yt,'1');
        2: OutTextXY(xt-2,yt,'2');
        3: OutTextXY(xt-2,yt,'3');
        4: OutTextXY(xt-2,yt,'4');
        5: OutTextXY(xt-2,yt,'5');
        6: OutTextXY(xt-2,yt,'6');
        7: OutTextXY(xt-2,yt,'7');
        8: OutTextXY(xt-2,yt,'8');
        9: OutTextXY(xt-2,yt,'9');
        10: OutTextXY(xt-2,yt,'10');
        11: OutTextXY(xt-2,yt,'11');
        12: OutTextXY(xt-2,yt,'12');
        13: OutTextXY(xt-2,yt,'13');
        14: OutTextXY(xt-2,yt,'14');
        15: OutTextXY(xt-2,yt,'15');
        16: OutTextXY(xt-2,yt,'16');
        17: OutTextXY(xt-2,yt,'17');
        18: OutTextXY(xt-2,yt,'18');
        19: OutTextXY(xt-2,yt,'19');
        20: OutTextXY(xt-2,yt,'20');
    end;
end;  {write_time}

{**** procedure to plot points on the graph ****}
procedure plot_points;
var timer :integer;
   max_the_pos,max_the_neg :real;
begin
    timer := 1;                         {value of time}
```

```
    max_the_pos := 0;                    {maximum positive angle}
    max_the_neg := 0;
    for i := 1 to j do
    begin
       x := round(getmaxx/2-270 + time_sc*time_co[i]);      {time}
       y := round(scale*angle_co[i]);
       if (angle_co[i] > 0) then                              {positive angle}
       begin
          if angle_co[i] > max_the_pos then
            max_the_pos := angle_co[i];
           y := round(getmaxy/2 - y);                  {coordinate of theta}
       end
       else if (angle_co[i] < 0) then
       begin              {negative angle}
          if angle_co[i] < max_the_neg then
            max_the_neg := angle_co[i];
          y := round(abs(y) + getmaxy/2)
       end
       else
          y := round(getmaxy/2);
       putpixel(x,y,red);                  {at 0 angle}
       if (round(time_co[i]) = timer) then    {check the value of time}
       begin
         write_time(timer);                {plot time coordinate}
         timer := timer +1;
       end;
    end;
    write_theta(max_the_pos,max_the_neg);  {plot anlge coordinates}
end;   {plot_points}

{*** procedure to initialize graphics mode ****}
procedure init_graph;
begin
    grdriver := detect;
    Initgraph(grdriver,grmode,'c:\tp6\bgi ');
    errcode := graphresult;
    if errcode <> grok then
    begin
      writeln('Graphics erro : ',GraphErrorMsg(Errcode));
      readln;
      halt(1);
    end
end; {init_graph}

{*** procedure to write string values on screen ****}
procedure write_strings(ch:char);
begin
    setcolor(green);
    SetTextstyle(defaultFont,Horizdir,2);  {charsize =1}
    SetTextjustify(centertext,centertext);
    OutTextXY(350,280,'Time (seconds)');                {CP is updated}
    setcolor(green);
    SetTextstyle(defaultfont,Vertdir,2);          {charsize =1}
    SetTextjustify(centertext,centertext);
```

```
   if ch = '7' then
     OutTextXY(10,240,'(-) Cart displacement [m] (+)')
   else if ch = '8' then
   begin
     OutTextXY(10,240,'(-) Cart acceleration (+)');
     OutTextXY(25,240,'[m/sqs]')
   end
   else if ch = '9' then
     OutTextXY(10,240,'(-) Cart velocity [m/s] (+)')
   else
     OutTextXY(10,240,'(-) Theta (radians) (+)');
end; {write_strings}

{** procedure to start graph of rigid pole without friction***}
procedure start_g_r_wof(choice:char);
var i :integer;
begin
   clrscr;
   write('Scale factor of the vertical coordinate (1-1000) = ');
   readln(scale);
   write('Scale factor of the horizontal coordinate (1-50) = ');
   readln(time_sc);
   get_ex_data(choice);

   init_graph;
   plot_line(choice);
   plot_points;
   write_strings(choice);
   readln;
   closegraph;
end; {START_G_R_WOF}

{*** procedure to write heading on the pole cart balancing system ***}
procedure write_cart_pole_heading;
var i:integer;
begin
   setcolor(cyan);
   SetTextstyle(defaultFont,Horizdir,2); {charsize =1}
   SetTextjustify(centertext,centertext);
   OutTextXY(round(getmaxx/2),round(getmaxy/2+95),'CART POLE BALANCING SYSTEM');
   setcolor(13);
   SetTextstyle(defaultFont,Horizdir,2); {charsize =1}
   SetTextjustify(centertext,centertext);
   OutTextXY(round(getmaxx/2),round(getmaxy/2+115),'Real time simulation');
   for i := 1 to 80 do
     putpixel(round(getmaxx/2-150+i),round(getmaxy/2+150),green);
   setcolor(green);
   SetTextstyle(defaultFont,Horizdir,0); {charsize =1}
   SetTextjustify(centertext,centertext);
   OutTextXY(round(getmaxx/2),round(getmaxy/2+150),'Elastic pole');
   for i := 1 to 80 do
     putpixel(round(getmaxx/2-150+i),round(getmaxy/2+165),blue);
   setcolor(blue);
   SetTextstyle(defaultFont,Horizdir,0); {charsize =1}
```

```
   SetTextjustify(centertext,centertext);
   OutTextXY(round(getmaxx/2),round(getmaxy/2+165),'Rigid pole');
end;  {write_cart-pole_headng}

{*** procedure to draw the track ****}
procedure draw_track;
begin
   write_cart_pole_heading;
   setcolor(white);
   setfillpattern(gray50,brown);
   bar3d(round(getmaxx/2-
280),round(getmaxy/2+35),round(getmaxx/2+270),round(getmaxy/2+55),10,topoff);
   bar(round(getmaxx/2-280),round(getmaxy/2-35),round(getmaxx/2-270),round(getmaxy/2+45));
   bar(round(getmaxx/2+270),round(getmaxy/2-35),round(getmaxx/2+280),round(getmaxy/2+55));
end;  {draw_track}

{*** procedure to draw the cart ***}
procedure draw_cart(change:real);
begin
  setfillpattern(gray50,red);
  if mt < 0.6 then
     bar(round(getmaxx/2-
25+change),round(getmaxy/2+10),round(getmaxx/2+25+change),round(getmaxy/2+30))
  else
     bar(round(getmaxx/2-50+change),round(getmaxy/2-
10),round(getmaxx/2+50+change),round(getmaxy/2+30));
end;  {draw_cart}
{**** procedure to draw the wheel **}
procedure draw_wheel(change:real);
begin
   setcolor(white);
   setfillpattern(gray50,white);
   if mt < 0.6 then
   begin
     pieslice(round(getmaxx/2-15+change),round(getmaxy/2+30),0,360,5);
     pieslice(round(getmaxx/2+15+change),round(getmaxy/2+30),0,360,5);
   end
   else
   begin
     pieslice(round(getmaxx/2-30+change),round(getmaxy/2+30),0,360,9);
     pieslice(round(getmaxx/2+30+change),round(getmaxy/2+30),0,360,9);
   end;
end;  {draw_wheel}

{*** procedure to draw the hinge **}
procedure draw_hinge(change:real);
begin
   setcolor(yellow);
   if mt < 0.6 then
   begin
      arc(round(getmaxx/2+change),round(getmaxy/2+10),0,180,9);
      circle(round(getmaxx/2+change),round(getmaxy/2+6),3);
      circle(round(getmaxx/2+change),round(getmaxy/2+6),1);
   end
```

```
   else
   begin
      arc(round(getmaxx/2+change),round(getmaxy/2-10),0,180,10);
      circle(round(getmaxx/2+change),round(getmaxy/2-15),3);
      circle(round(getmaxx/2+change),round(getmaxy/2-15),1);
   end;
end;  {draw_hinge}

{*** procedure to draw the elastic pole ***}
procedure draw_elastic_pole(L1,change:real;n:integer;var TL:real);
var  L,x2,y2,g,elong,L2,k,i:real;
   scale :integer; ratio:real;
begin
   ratio := getmaxy/getmaxx;
   scale := 200;
   g := 9.81;
   k := pb*pd*pd*pd*E/12;      {EI}
   L := pl;
   elong := (mp*g*sin(angle_co[n])/L)/24 * (6*L*L*L1*L1 - 4*L*L1*L1*L1 + L1*L1*L1*L1)/k;
   L2 := sqrt(L1*L1 + elong*elong);  {distance of elastic pole at any point I}
   if L1 <> 0 then
     TL := arctan(elong/L1)      {the angle due to elasticity}
   else tl := 0.0;
    x2 := sin(angle_co[n]+TL)*L2*scale*ratio;
   y2 := cos(angle_co[n]+TL)*L2*scale;
{
    x2 := sin(T_ANGLE[N])*L2*scale*ratio;
   y2 := cos(T_angle[N])*L2*scale;
}
   if mt < 0.6 then       {small cart}
     putpixel(round(getmaxx/2+change+x2),round(getmaxy/2+6-y2),green)
   else
     putpixel(round(getmaxx/2+change+x2),round(getmaxy/2-15-y2),green);
end;   {draw_elastic_pole}

{*** procedure to calculate the displacement of the cart ***}
procedure find_cart_displacement(n,scale:integer;var change:real);
var ch :char;
begin
   change := (scale)*c_ac[n];
         { c_ac[n] X = distance covered by cart = cart acceleration}
   if (change <= -220.0) or (change >= 220.0) then
   begin                        {failure cart hit limit of track}
      setcolor(RED);
      SetTextstyle(defaultFont,Horizdir,1);  {charsize =1}
      SetTextjustify(centertext,centertext);
      OutTextXY(round(getmaxx/2),round(getmaxy-25),'FAILURE !!! CART HIT THE TRACK
LIMIT');
      OutTextXY(round(getmaxx/2),round(getmaxy-15),'Press return for main menu.');
      failure := true;
      readln(ch);
   end;
end;  {find_cart_displacement}
```

```
{*** procedure to draw the entire system at any time n ***}
{*** this will calculate the position of the cart ***}
procedure draw_system_at_any_time(n,scale:integer;var change:real);
begin
   find_cart_displacement(n,scale,change);
   draw_track;
   draw_cart(change);
    draw_wheel(change);
   draw_hinge(change);
end; {draw_system_at_any_time}

{**** procedure to draw the pole ****}
procedure draw_pole(change:real);
var  x,y,L1,L,TL,ta,ratio:real;
   n,scale : integer;
begin
   ratio := getmaxy/getmaxx;
   L := pl;  {1.0;}
   scale := 200;
   for n := 1 to j do   {j is the total number of elements}
   begin
     L1 := 0.004;
     repeat
        x := sin(angle_co[n])*L1*ratio;
        y := cos(angle_co[n])*L1;
        if mt < 0.6 then
         putpixel(round(getmaxx/2+change+x*scale),round(getmaxy/2+6-y*scale),blue)
        else
         putpixel(round(getmaxx/2+change+x*scale),round(getmaxy/2-15-y*scale),blue);
        draw_elastic_pole(L1,change,n,TL);
        L1 := L1 + 0.004;
     until L1 >= L;
{      if time_co[n] = 0.02 then
     begin
        setcolor(RED);
        SetTextstyle(defaultFont,Horizdir,1);

        SetTextjustify(centertext,centertext);
        OutTextXY(round(getmaxx/2),round(getmaxy-175),' Time   Cart displacement  Elastic pole
angle   Rigid pole angle');
        OutTextXY(round(getmaxx/2),round(getmaxy-165),'0.02      -0.2994        30.52 degrees
9.924 degrees');
        readln; clearviewport;
     end;
}
{      ta := angle_co[n] + TL;  }   {total elastic pole angle}
{      writeln(f1,time_co[n]:10:6,'  ',tl:10:6,'  ',angle_co[n]:10:6); }
     delay(3000);
{      RESTORECRTMODE;
     GOTOXY(10,24); WRITE('X = ',CHANGE/scale:10:6,'  Time = ',time_co[n]:10:6);
     delay(1000);
     SETGRAPHMODE(getgraphmode);
}
```

```
      setviewport(round(getmaxx/2-269),round(getmaxy/2-235),(getmaxx-
51),round(getmaxy/2+35),clipon);
      clearviewport;
      setviewport(0,0,getmaxx,getmaxy,clipon);
      draw_system_at_any_time(n+1,scale,change);
      if failure then exit;
    end; {for}
end; {draw_pole}

{*** procedure to draw the cart and the pole ****}
procedure draw_pole_cart(ch:char);
var i :integer;
   change : real;   {X displacement of the cart}
begin
    get_ex_data(ch);
    failure := false;
    init_graph;
{    change := 0;}
    write_cart_pole_heading;
    draw_system_at_any_time(1,200,change);
{    draw_track;
    draw_cart(change);
    draw_wheel(change);
    draw_hinge(change);
}
    draw_pole(change);;
    if (not failure) then
    begin
     setcolor(RED);
     SetTextstyle(defaultFont,Horizdir,1);  {charsize =1}
     SetTextjustify(centertext,centertext);
     OutTextXY(round(getmaxx/2),round(getmaxy-25),'SUCCESSFUL !!! SIMULATION TIME
FINISHED');
     OutTextXY(round(getmaxx/2),round(getmaxy-15),'Press return for main menu.');
     readln(ch);
    end;
    closegraph;
end;      {DRAW_POLE_CART}
end.      {UNIT GRA28}
```

APPENDIX B

The Quanzer Consulting Incorporated Controller for the Flexible Pole-Cart Balancing Problem

1. Description

The Quanzer company make a range of devices for control engineering experiment. This experiment designed for this thesis combines the inverted pendulum and the flexible link module to obtain an interesting variation of the classical inverted pendulum.

The system is assembled as shown in figure B.1. Note that you can either use the full pendulum or just a small shaft to couple the flexible link to the cart. The two masses supplied with the system must also be attached to the tip of the link as shown in figure B.1.

This appendix describes the problem for deriving the proprietory control system used by Quanzer.

2. Mathematical model

Consider the simplified diagram shown in figure B.2. The stiffness of the link is assumed to be collocated at the point of attachment to the pendulum. The rotational stiffness is represented by K_s. The masses of the moving elements are as shown in the figure. In order to derive the differential equations of the system, we need to obtain the kinetic and potential energy for each element in the system. These are obtained as follows:

Consider the coordinate frames defined in the figure B.3. Using transformation matrices, we have the following transformations:

$$T^{01} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T^{12} = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & h\sin(\alpha) \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 1 & \cos(\alpha) & h\cos(\alpha) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T^{32} = \begin{bmatrix} \cos(\beta) & 0 & -\sin(\beta) & L_b \sin(\beta) \\ 0 & 1 & 0 & 0 \\ \sin(\beta) & 1 & \cos(\beta) & L_b \cos(\beta) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The last column in each matrix represents the position of the frame relative to the previous frame. The transformation $T^{02} = T^{01}T^{12}$ represents the position and orientation of the camera relative to the base frame and $T^{03} = T^{01}T^{12}T^{23}$ is the position and orientation of the load attached at the tip (bulb plus two masses) relative to the base frame.

Defining

$$T^{01}T^{12} = \begin{bmatrix} & & & P_c^x \\ & [R^{12}] & & 0 \\ & & & P_c^z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Take note that there is no motion along the 'y' direction. Then the kinetic energy of the camera is given by:

$$KE_{camera} = 0.5M_c\left(\left[\frac{\delta P_c^x}{\delta t}\right]^2 + \left[\frac{\delta P_c^z}{\delta t}\right]^2\right) + 0.5J_c\left[\frac{\delta \alpha}{\delta t}\right]^2$$

and the potential energy of the camera is given by:

$$PE_{camera} = M_c g P_c^z$$

Similarly for the load at the end of the link,

$$T^{01}T^{12}T^{23} = \begin{bmatrix} & & & P_b^x \\ & [R^{23}] & & 0 \\ & & & P_b^z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Then, the kinetic energy of the load is:

$$KE_{bulb} = 0.5M_b\left(\left[\frac{\delta P_b^x}{\delta t}\right]^2 + \left[\frac{\delta P_b^z}{\delta t}\right]^2\right)$$

and the potential energy of the load is given by:

$$PE_{bulb} = M_c g P_b^{\,z}$$

The potential energy in the spring (equivalent stiffness of the link collocated at the mounting to the pendulum) is:

$$PE_{spring} = 0.5 K_s \beta^2$$

The potential energy of the pendulum is derived as for the camera by substituting L_p for h in T^{12} and M_p for M_c in the energy equations. It is assumed the pendulum is a point mass located at a distance L_p from the joint (L_p is half the actual physical length of the pendulum).

The kinetic energy of the cart is given by:

$$KE_{cart} = 0.5 M_{cart} x^2$$

All of the above equations are implemented in a MAPLE program that computes the Langrarian about each independent axis and derives the nonlinear differential equations. The nonlinear differential equations are written to disk.. A second program reads the nonlinear equations and linearizes them about the operating point (0,0,0). The linearized model results in the matrix equation:

$$\begin{bmatrix} \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \\ \ddot{x} \\ \ddot{\alpha} \\ \ddot{\beta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & a_{42} & a_{43} & 0 & 0 & 0 \\ 0 & a_{52} & a_{53} & 0 & 0 & 0 \\ 0 & a_{62} & a_{63} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \alpha \\ \beta \\ \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix} F$$

The values for the constants in the matrix are then used in MATLAB to design the controller.

3. Control system design

The design proceeds as with the inverted pendulum experiment. Substituting system parameters into the matrix equation obtained:

$$\begin{bmatrix} \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \\ \ddot{x} \\ \ddot{\alpha} \\ \ddot{\beta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -5.8 & -4.7 & 0 & 0 & 0 \\ 0 & 49.6 & 92.8 & 0 & 0 & 0 \\ 0 & -11.47 & -94.7 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \alpha \\ \beta \\ \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1.3 \\ -4.3 \\ 0.96 \end{bmatrix} F$$

The force output must be converted to a voltage input since the motor is driven by a voltage:

$$F = \frac{T}{r} = \frac{K_m K_g I_m}{r} = \frac{K_m K_g}{Rr} V - \frac{K_m{}^2 K_g{}^2}{Rr^2} \dot{x}$$

substituting parameter values into the matrix equation results in:

$$\begin{bmatrix} \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \\ \ddot{x} \\ \ddot{\alpha} \\ \ddot{\beta} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -5.8 & -4.7 & -10.1 & 0 & 0 \\ 0 & 49.6 & 92.8 & 32 & 0 & 0 \\ 0 & -11.47 & -94.7 & -7.4 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \alpha \\ \beta \\ \dot{x} \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1.3 \\ -4.3 \\ 0.96 \end{bmatrix} V$$

A controller is then generated using the LQR method within MATLAB. The Q and the r weighting factors chosen are:

$Q = diag(0.1 \ 1 \ 0 \ 0 \ 0 \ 0.1)$

$r = 0.001$

resulting in the feedback gains:

$K = [-10 \ 62.7 \ 48.5 \ -16 \ -11.4 \ -10.7]$ for units in metres and radians.

and

$K = [-0.1 \ -1.1 \ 0.85 \ -0.16 \ -0.2 \ -0.19$ for units in centimetres and degrees.

The closed loop eigenvalues for the above gain are:

$[-15.4 +- j6.2]$

$[-1.7 +- j7.9]$

$[-1.4 +- j0.92]$

Figure FP4 compares the response of the modelled system to a step command in cart position using the gains obtained above and a set of gains with *K3* and *K6* set to zero. Clearly, the feedback gains *K3* and *K6* are necessary to stabilized the system.

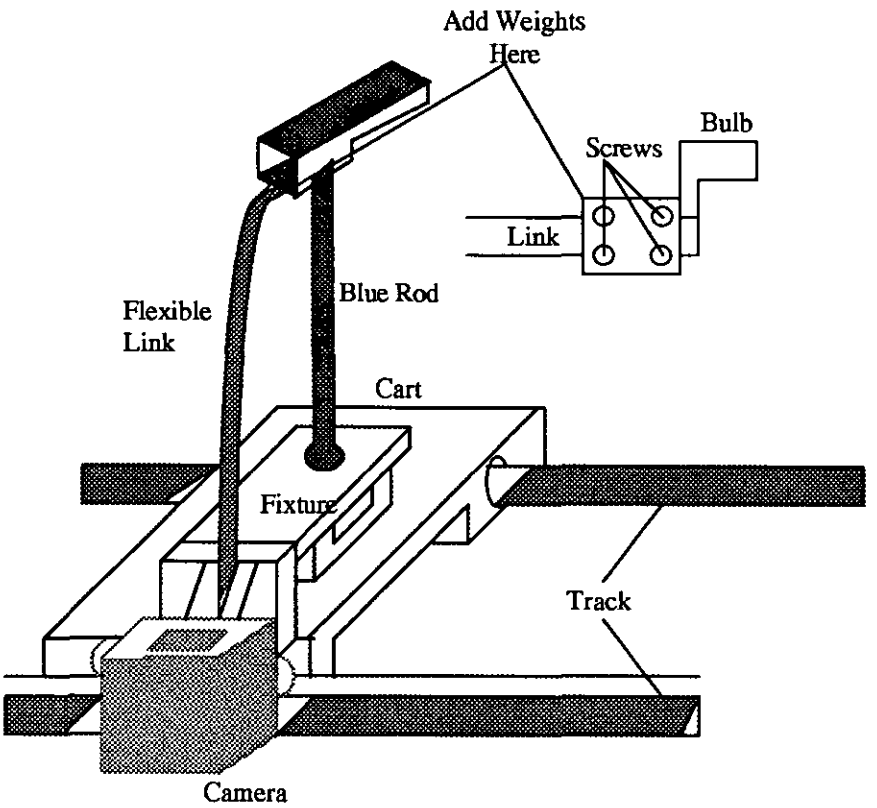


Figure B.1
Assembly of flexible inverted pendulum

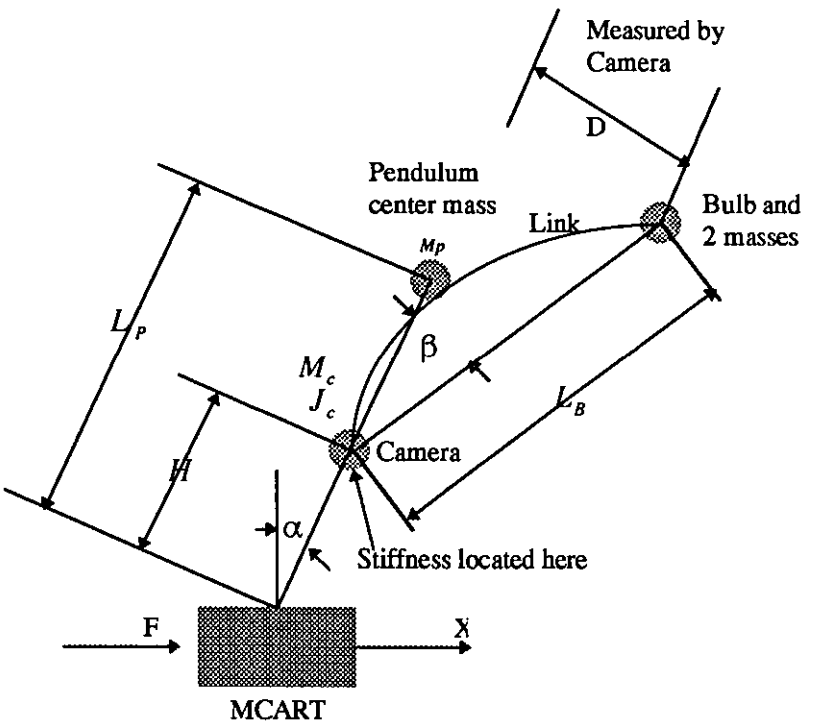


Figure B.2
Simplified model of the flexible inverted pendulum

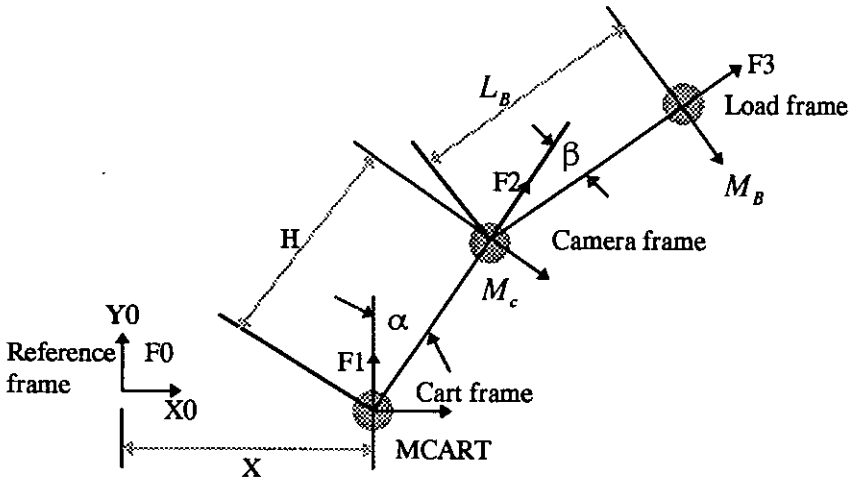


Figure B.3
Coordinate frame definitions

4. Results

The above controller is implemented on an actual system and works well. Figure B.4 compares the response of the modelled system to a step command in cart position using the gains obtained above and a set of gains with *K3* and *K6* set to zero. Figure B.5 shows the deflection response to a tap to the pendulum. Note that the system does not stabilize but there is a limit cycle due to friction and other nonlinearities. Figure B.6 shows the response of the system when the feedback gains *K3* and *K6* are set to zero.

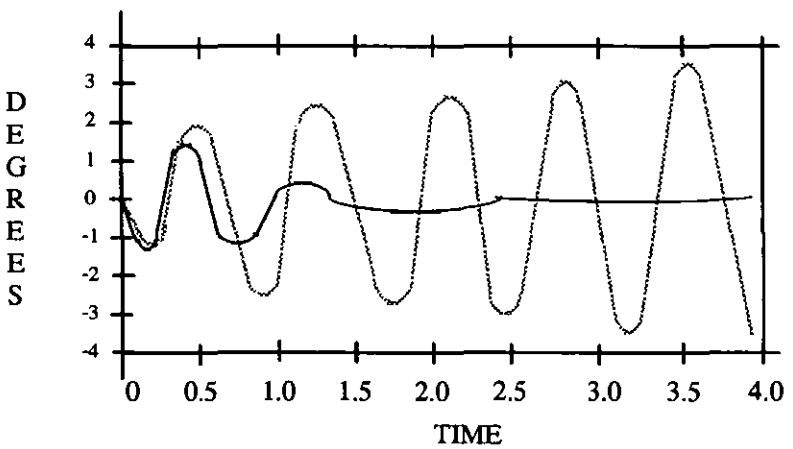


Figure B.4
Deflection (β) Response with and without camera feedback

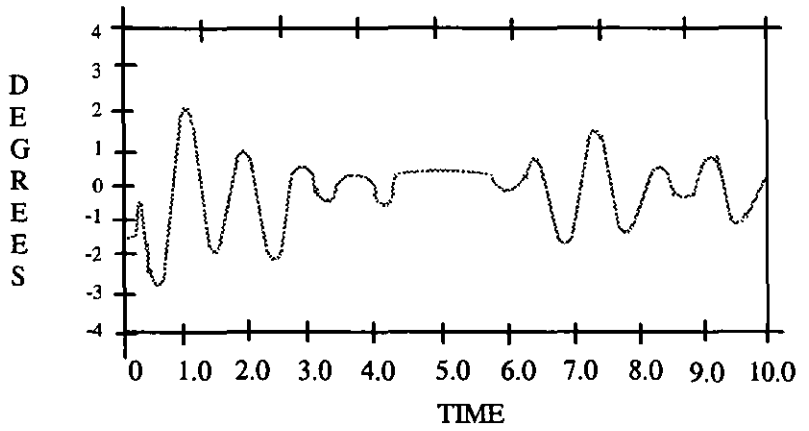


Figure B.5
Deflection (β) Response to a tap on the pendulum using full state feedback

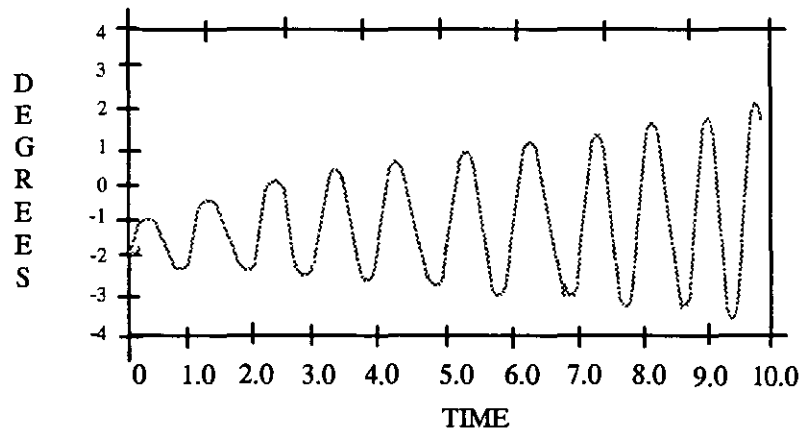


Figure B.6
Deflection (β) Response when K3 and K6 are set to zero

5. Reference

1. P. P. Richard "Robot Manipulators: Mathematics, Programming and Control ,", The MIT press, 1981.

APPENDIX C

The Neural Network On-Line Program for the Flexible Pole-Cart Balancing System

Important procedures in the program:

1. **get_neural_net( )** - A procedure used to set up the architecture of the neural network.
2. **get_layer_info( )** - A procedure used to get the neural network parameters.
3. **set_up_network( )** - A procedure used to construct and interconnect the layers of the neural network. This includes memory allocation of the neural network to the computer.
4. **read_weights( )** - A procedure used to get the trained weight values of the neural network from an external file wed2_111.dat and allocate it to each layer connections.
5. **main_loop( )** - A procedure used to operate the controller with an interaction of the user.
6. **reset_ad_da_con( )** - A procedure used to prepare the analog/digital digital/analog converter for operation.
7. **initialize_data( )** - A procedure used to initialize the values of the sensors.
8. **set_clock_frequency( )** - A procedure used to set the clock frequency for real time operation.
9. **newtimer( )** - A tc++ built in procedure used to instruct the interrupt vectors to operate in real time.
10. **enable( )** - A tc++ buit in procedure used to enable the interrupt service routine (isr). At this point newtimer( ) start operating in real time and sensors are getting data from the plant.
11. **get_voltage_from_neural_net( )** - A procedure used to get the voltage needed to operate the actuator and control the system.

12. **get_data_from_acd( )** - A procedure used to assign the normalized data taken from the sensors to the neural network input buffers.
13. **forward_prop( )** - A procedure used to obtain the required output of the neural network to control the system by forwardly propagating the sum of the weights and the input values of each layer. This uses a sigmoid function in procedure calc_out( ) to a value between 0 to 1.
14. **get_final_value( )** - A procedure for determining the actual magnitude and direction of the final output of the neural network controller.
15. **main_menu( )** - A procedure to display user options on operations the controller.
16. **printval( )** - A procedure use to display the status of the system. (i.g. position and velocity of the cart, the pole angle, and the pole deflection).
17. **save_data_to_file( )** - A procedure used to save the informations needed to examine the performance of the controller. The data are save in MATLAB format and are ready for MATLAB graphical representation.

C++ Object Oriented Class/Structures Used in the Program

```
/*
This classes are stored inside file layer4.h
*/

#define MAX_LAYERS      5
#define MAX_VECTORS     500


class network;
class layer                                     // The components of a layer
{
        protected:
                int num_inputs;
                int num_outputs;
                float *outputs;                 // pointer to array of outputs
                float *inputs;                  // pointer to array of inputs, which are outputs of
                                                // some other layer
        friend network;
        public:
                virtual void calc_out()=0;
};

class input_layer: public layer                 // The components of input layer
{
        private:
                float noise_factor;             // Noise parameter applied to input data
                float * orig_outputs;
        public:
                input_layer(int, int);
                ~input_layer();
                virtual void calc_out();
                void set_NF(float);
        friend network;
};

class middle_layer;                             // The components of hidden layers
class output_layer:         public layer
{
protected:
        float * weights;                        // Pointers of weight values
        float * output_errors;                  // array of errors at output
        float * back_errors;                    // array of errors back-propagated
        float * expected_values;                // to inputs
        float * cum_deltas;                     // for momentum
        float * past_deltas;                    // for momentum
friend network;
```

```
public:
        output_layer(int, int);
        ~output_layer();
        virtual void calc_out();
        void list_weights();
        void read_weights(int, FILE *);
        void list_outputs();
};

class middle_layer:        public output_layer
{
private:
public:
        middle_layer(int, int);
        ~middle_layer();
        void calc_error();
};

class network                                   // The components of FNN architecture
{
private:
        layer *layer_ptr[MAX_LAYERS];           // Pointer for every layer
        int number_of_layers;                   // Actual number of layers in the network
        int layer_size[MAX_LAYERS];
        float *buffer;                          // Input data storage
        fpos_t position;                        // Flag for status of system
        unsigned training;                      // Flag for testing or training operation
public:
        network();
        ~network();
        void set_training(const unsigned &);
        unsigned get_training_value();
        void get_layer_info();
        void set_up_network();
        void read_weights(FILE *);
        float get_final_output();
        void write_outputs(FILE *);
        void list_outputs();
        void forward_prop();
        int get_data_from_adc(float a,float x_d,float a_d,float d_d);
        void set_up_pattern(int);
};
```

```
/*
        This is the main program to test capability of feedforward neural network
        controller to control the flexible pole-cart balancing problem.
        By: Elmer P. Dadios
        Manufacturing Engineering Department
        Loughborough University of Technology, UK


#pragma inline                                  // Inline assembly declaration
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <bios.h>
#include <dos.h>
#include <io.h>
#include <float.h>
#include <stdlib.h>
#include <graphics.h>
#include <d:\research\neu\dt2811.drv>           // Use for data translation AD/DA converter
extern void interrupt newtimer(...);            // Declare other functions for real time operarion
static void interrupt (*oldtimer)(...);

#include "layer4.cpp"
#define WEIGHTS_FILE "wed2_111.dat"
network backp;   // create a network object
long int vectors_in_buffer;

const int no_of_data = 3000;
int i,ctr,ctr1;
float x_data[3000],a_data[3000],d_data[3000],v_data[3000];
FILE * weights_file_ptr;

float a_n,d_n,x_n,x_dn,a_dn,d_dn;               // Normalized values of data from the plant
float alpha,alpha_d,x,x_d,def,def_d,voltage,vf,force,t_mass;
float alpha_pf,alpha_p,alpha_f,x_pf,x_p,x_f,def_pf,def_p,def_f,def_df;      // Raw data
float cal_constant_x,cal_constant_alpha,cal_constant_def;                   // Sensors constant value
float wcut,fcut,kf1,kf2,ts,fsamp;                                           // For low pass filter
int ivolts,alpha_int,x_int,def_int,motor_off;   // Integer values of raw data for AD/DA converter
float x_bias,alpha_bias,def_bias;               // Offset/initial value of sensor in volts
int gdriver,gmode,cdiv,cdiv_lo,cdiv_hi;         // Graphics & clock devider
float timen,timep,u_freq,del_time,base_freq;        // Frequency for printing realtime data to
screen
float volts_o;                                  // Actual voltage value for actuator
int volt_int;                                   // Integer voltage value for AD/DA converter

void initialize_data(void)
{
        // sampling frequency
        fsamp = 200.0;
        ts = 1/fsamp;
        ctr = 0;
```

```
        // setup  printval() frequency
        u_freq = 5; /* refresh realtime data on screen at 5 Hz */
        del_time = 1./u_freq;

        //sensor calibration constant
        cal_constant_x = 91/10.00;              // 91 cm over entire range
        cal_constant_alpha = 352.0/24;          // full turn over 24 volts
        cal_constant_def = 2.54;  // 1 inch per volt = 2.54 cm per volt but you should calibrate first
        x_bias = 0;                             // cart displacement offset
        alpha_bias = -0.6;                              // Pole angle offset
        def_bias =  -1.95;                      // Pole deflection offset

        // other parameters
        motor_off = 1;                          // start with motor off
        timen = 0;                              // initialize real time
        timep = 0.1;

        // lowpass filter
        fcut = 2.0;
        wcut = 2.0*3.14*fcut;
        kf1 = wcut*ts/(2+wcut*ts);
        kf2 = (wcut*ts-2)/(wcut*ts+2);
}


// procedure to print the values
void printval(void)
{
        textcolor(GREEN);
        gotoxy(1,1); cprintf("    REAL TIME DATA");
        gotoxy(1,2); cprintf("Real time (sec)    = %6.1f",timen);
        gotoxy(1,3); cprintf("Voltage applied    = %6.4f",voltage);
        gotoxy(1,4); cprintf("Cart position (cm)  = %6.4f  x_n  = %6.4f",x,x_n);
        gotoxy(1,5); cprintf("Pole anlge (deg)   = %6.4f  a_n  = %6.4f",alpha,a_n);
        gotoxy(1,6); cprintf("Cart velocity (cm)  = %6.4f  x_dn = %6.4f",x_d,x_dn);
        gotoxy(1,7); cprintf("Pole velocity (d/s) = %6.4f  a_dn = %6.4f",alpha_d,a_dn);
        gotoxy(1,8); cprintf("Pole deflection    = %6.4f  d_n  = %6.4f",def,d_n);
        gotoxy(1,9); cprintf("Velocity of def    = %6.4f  d_dn = %6.4f",def_d,d_dn);
        gotoxy(15,20);cprintf("ctr = %d",ctr);
        gotoxy(15,18); /* move cursor to the choice position */

}       /* printval */


// procedure to reset the AD/DA controllers
void reset_ad_da_con(void)
{
        reset_ad();
        reset_da();
        daout(0,2048);
        daout(1,2048);
        daout(2,2048);
}
```

```
// Procedure  to set up the architecture of the neural network
void set_neural_net(void)
{
        backp.get_layer_info();                 // get layer information
        backp.set_up_network();                 // set up the network connections

        // read in the weight matrix defined by a backpropagation simulator
        if ((weights_file_ptr=fopen(WEIGHTS_FILE,"r"))==NULL)
        {
                cout << "problem opening weights file\n";
                exit(1);
        }
        backp.read_weights(weights_file_ptr);
        fclose(weights_file_ptr);
}       // set_neural_net

void get_voltage_from_neural_net(void)
{
        a_n = alpha/45.0;                               // normalized angle
        x_n = x/50.0;                           // normalized cart position
        a_dn = alpha_d/50.0;                    // normalized angular velocity
        x_dn = x_d/50.0;                        // normalized cart velocity
        d_dn = def_d/50.0;                      // normalized deflection velcity

        vectors_in_buffer = backp.get_data_from_adc(a_n,x_dn,a_dn,d_dn);
        for (i=0; i<vectors_in_buffer; I++)     // Assign data from sensor to FNN buffer
        {                                       // process vectors
                backp.set_up_pattern(i);        // get next pattern
                backp.forward_prop();           // forward propagate
                voltage = backp.get_final_output()*5;
        }
        if ((x>14)&&(alpha>1.01)&&(x_d>0.01))
         voltage += 2.0;  // bring the cart to the center of the track
        if ((x>14)&&(alpha>1.01)&&(x_d<-0.01))
         voltage += 1.1;  // bring the cart to the center of the track
        if ((x<-14)&&(alpha<-1.01)&&(x_d<-0.01))
         voltage += -0.5;  // bring the cart to the center of the track
        if ((x<-14)&&(alpha<-1.01)&&(x_d>0.01))
         voltage += -0.2;  // bring the cart to the center of the track

        if ((alpha>1.01)&&(alpha_d>0.01))
                voltage += 3.0;  //balance the pole
        if ((alpha<-1.01)&&(alpha_d<-0.01))
                voltage += -3.0;  //balance the pole

        if (alpha>2.31)
                voltage += 5.5;  //balance the pole
        if (alpha<-2.31)
                voltage += -5.5;  //balance the pole

}       // get voltage from neural network
```

```
extern void interrupt far newtimer(...)                    // ISR real time operation
{
        char *data87[94];
        asm fsave data87
        _clear87();
        timen= timen+ts;

        // Get data from sensors measurement
        x_pf = x_f;                                        // save previous filtered data
        x_p = x;                                           // save previous raw data
        x_int = adin(4);                                   // sample the A to D channel
        x = itov(x_int)*cal_constant_x - x_bias;           // actual cart displacement
        x_f = kf1*(x+x_p)-kf2*x_f;                         // digital low pass filtering
        x_d = (x_f-x_pf)/ts;                               // actual cart velocity

        alpha_pf = alpha_f;                                // save previous filtered data
        alpha_p = alpha;                                   // save previous raw data
        alpha_int = adin(5);                               // sample the A to D channel
        alpha = itov(alpha_int)*cal_constant_alpha - alpha_bias; // actual pole angle
        alpha_f = kf1*(alpha+alpha_p)-kf2*alpha_f;                 // digital low pass filtering
        alpha_d = (alpha_f-alpha_pf)/ts;                   // actual velocity of the pole

        def_pf = def_f;                                    // save previous filtered data
        def_p = def;                                       // save previous raw data
        def_int = adin(10);                                // sample the A to D channel
        def = itov(def_int)*cal_constant_def -def_bias;    // actual deflection of the pole
        def_f = kf1*(def+def_p)-kf2*def_f;                 // digital low pass filtering
        def_df = def_df+def_d*wcut*ts;
        def_d = wcut*def - def_df;                 // actual velocity of pole deflection

        get_voltage_from_neural_net();                     // procedure to get actual voltage required to
                                                           // control the sytem from FNN
        if(motor_off >0)
                voltage = 0;
        // make sure that it will not exceed the capacity of the actuator
        if(voltage > 4.95) voltage = 4.95;
        if(voltage < -4.95) voltage = -4.95;

        // prepare for storing results to external file
        if ((abs(ctr) < no_of_data)&&(motor_off < 0))
        {
                ctr++;
                ctr1 = ctr;
                x_data[ctr1] = x;
                a_data[ctr1] = alpha;
                d_data[ctr1] = def;
                v_data[ctr1] = voltage;
        }
```

```
        // output the voltage calculated to the actuator for operation via DA converter
        volts_o = voltage;
        volt_int = volts_o*2048/5.0+2047;
        daout(0,volt_int);
        asm frstor data87
}       // end of newtimer


// procedure to go back to old timer
void get_old_timer(void)
{
        disable();
        setvect(0xc1,oldtimer);
        enable();
        outportb(0x43,0x36);
        outportb(0x40,0xff);
        outportb(0x40,0xff);
        daout(0,2048);
        daout(1,2048);
        daout(2,2048);
}       /* get_old_timer */

/* procedure to set the clock frequency for real time */
void set_clock_frequency(void)
{
        base_freq = 1193000.0;          // 1.193 MHz is the base frequency of the clock in an AT
        cdiv = ceil(base_freq*ts);      // setup clock divider
        outportb(0x43,0x36);            // setup clock number 2
        cdiv_hi = cdiv / 255;           // high byte of cdiv
        cdiv_lo = fmod(cdiv,255);       //low byte of cdiv
        outportb(0x40,cdiv_lo);         // write out low byte
        outportb(0x40,cdiv_hi);         // then high byte

        disable();                      // diable interrupts
        oldtimer = getvect(0x1c);       //save old isr address
        setvect(0x1c,newtimer);         // setup the new isr
        enable();                               // enable interrupts, at this point newtimer starts
running
}       set_clock_frequency

// procedure to display the main menu for user options
void main_menu(void)
{
        clrscr();
        textcolor(RED + BLINK);
        gotoxy(1,15);
        if (motor_off > 0)
                cprintf("MOTOR IS OFF");
        else
                cprintf("MOTOR IS ON");
        textcolor(BLUE);
        gotoxy(1,16);
        cprintf("[Q] to quit");
        gotoxy(1,17);
```

```
        cprintf("[S] to start/stop motor");
        gotoxy(1,18); textcolor(CYAN);
        cprintf("Your choice =  ");
}       // main_menu


// Procedure to operate the controller with the interaction of the user
void main_loop(void)
{  char choice;

        clrscr();
        reset_ad_da_con();
        _fpreset();                                     // reset floating point processor
        initialize_data();
        set_clock_frequency();
        main_menu();
        do
        {
                choice = '';
                if (kbhit())
                {
                        choice = getch();               // get the character
                        flushall();                     // flush the keyboard buffer
                }
                if ((choice=='Q')||(choice=='q'))       // quit program
                        get_old_timer();
                if ((choice=='S')||(choice=='s'))       // stop operation
                {
                        motor_off = -1*motor_off;// initially motor is off
                        main_menu();
                }
                if ((timen-timep)>del_time)             // frequency of printing realtime data to
                                                        // screen
                {
                        timep = timen;
                        printval();
                }
        } while((choice!='Q')&&(choice!='q'));
        textcolor(LIGHTGRAY);
        clrscr();
}       /* main_loop */


// Procedure in puting data to external file
void save_data_to_file(void)
{
        FILE *f1;
        int i;

        if ((f1 = fopen("xn_dat22.m","w"))==NULL)
        {
                puts("\ncannot open file xn_data");
                exit(1);
        }
```

```
        for (i=1;i<ctr1;i++)
                fprintf(f1,"%5.3f\n",x_data[i]);
        fclose(f1);

        if ((f1 = fopen("an_dat22.m", "w"))==NULL)
        {
                puts("\ncannot open file an_data");
                exit(1);
        }
        for (i = 1;i<ctr1;i++)
                fprintf(f1,"%5.3f\n",a_data[i]);
        fclose(f1);

        if ((f1 = fopen("dn_dat22.m", "w"))==NULL)
        {
                puts("\ncannot open file dn_data");
                exit(1);
        }
        for (i = 1;i<ctr1;i++)
                fprintf(f1,"%5.3f\n",d_data[i]);
        fclose(f1);

        if ((f1 = fopen("vn_dat22.m", "w"))==NULL)
        {
                puts("\ncannot open file dn_data");
                exit(1);
        }
        for (i = 1;i<ctr1;i++)
                fprintf(f1,"%5.3f\n",v_data[i]);
        fclose(f1);

}

// This is the main body of the program
void main()
{
        set_neural_net();
        main_loop();
        save_data_to_file();
}       // end main
```

```
// This is the file that contains the neural network procedures "Layer4.cpp"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "layer4.h"


// This is the squashing function that uses sigmoid
inline float squash(float input)
{
        if (input < -50)
                return 0.0;
        else    if (input > 50)
                return 1.0;
        else return (float)(1/(1+exp(-(double)input)));
}


// This a random number generator that will return a floating point value between -1 and 1
inline float randomweight(unsigned init)
{
        int num;
        if (init==1)                                    // seed the generator
        srand ((unsigned)time(NULL));
        num=rand() % 100;
        return 2*(float(num/100.00))-1;
}


// This function is needed for Turbo C++ and Borland C++ to link in the appropriate functions for fscanf
// floating point formats:
static void force_fpf()
{
        float x, *y;
        y=&x;
        x=*y;
}


// This is for the input layer
input_layer::input_layer(int i, int o)
{
        num_inputs=i;
        num_outputs=o;
        outputs = new float[num_outputs];
        orig_outputs = new float[num_outputs];
        if ((outputs==0)||(orig_outputs==0))
        {
                cout << "not enough memory\n";
                cout << "choose a smaller architecture\n";
                exit(1);
        }
```

```
        noise_factor=0;
}

input_layer::~input_layer()
{
        delete [num_outputs] outputs;
        delete [num_outputs] orig_outputs;
}


void input_layer::calc_out()
{
        // This will add noise to inputs  randomweight returns a random number between -1 and 1
        int i;
        for (i=0; i<num_outputs; i++)
                outputs[i] =orig_outputs[i]*
                        (1+noise_factor*randomweight(0));
}


// This is for the output layer
output_layer::output_layer(int ins, int outs)
{
        int i, j, k;
        num_inputs=ins;
        num_outputs=outs;
        weights = new float[num_inputs*num_outputs];
        output_errors = new float[num_outputs];
        back_errors = new float[num_inputs];
        outputs = new float[num_outputs];
        expected_values = new float[num_outputs];
        cum_deltas = new float[num_inputs*num_outputs];
        past_deltas = new float[num_inputs*num_outputs];

        if ((weights==0)||(output_errors==0)||(back_errors==0)
                ||(outputs==0)||(expected_values==0) || (past_deltas==0)||(cum_deltas==0))
        {
                cout << "not enough memory\n";
                cout << "choose a smaller architecture\n";
                exit(1);
        }

        // zero cum_deltas and past_deltas matrix
        for (i=0; i< num_inputs; i++)
        {
                k=i*num_outputs;
                for (j=0; j< num_outputs; j++)
                {
                        cum_deltas[k+j]=0;
                        past_deltas[k+j]=0;
                }
        }
}
```

```
output_layer::~output_layer()
{
        delete [num_outputs*num_inputs] weights;
        delete [num_outputs] output_errors;
        delete [num_inputs] back_errors;
        delete [num_outputs] outputs;
        delete [num_outputs*num_inputs] past_deltas;
        delete [num_outputs*num_inputs] cum_deltas;
}


void output_layer::calc_out()
{
        int i,j,k;
        float accumulator=0.0;
        for (j=0; j<num_outputs; j++)
        {
                for (i=0; i<num_inputs; i++)
                {
                        k=i*num_outputs;
                        if (weights[k+j]*weights[k+j] > 1000000.0)
                        {
                                cout << "weights are blowing up\n";
                                cout << "try a smaller learning constant\n";
                                cout << "e.g. beta=0.02    aborting...\n";
                                exit(1);
                        }
                        outputs[j]=weights[k+j]*(*(inputs+i));
                        accumulator+=outputs[j];
                }
                // use the sigmoid squash function
                outputs[j]=squash(accumulator);
                accumulator=0;
        }
}


void output_layer::read_weights(int layer_no,FILE * weights_file_ptr)
{
        int i, j, k;
        while (1)
        {
                fscanf(weights_file_ptr,"%i",&j);
                if ((j==layer_no)|| (feof(weights_file_ptr)))
                        break;
                else
                {
                        while (fgetc(weights_file_ptr) != '\n')
                                {;}                             // get rest of line
                }
        }
```

```
        if (!(feof(weights_file_ptr)))
        {       // continue getting first line
                i=0;
                for (j=0; j< num_outputs; j++)
                        fscanf(weights_file_ptr,"%f",&weights[j]);   // i*num_outputs = 0
                fscanf(weights_file_ptr,"\n");

                // now get the other lines
                for (i=1; i< num_inputs; i++)
                {
                        fscanf(weights_file_ptr,"%i",&layer_no);
                        k=i*num_outputs;
                        for (j=0; j< num_outputs; j++)
                                scanf(weights_file_ptr,"%f",&weights[k+j]);
                }
                fscanf(weights_file_ptr,"\n");
        }
        else cout << "end of file reached\n";
}


void output_layer::list_outputs()
{
        int j;
        for (j=0; j< num_outputs; j++)
                cout << "outputs["<<j<<"] is: "<<outputs[j]<<"\n";
}


// This is for the middle layer
middle_layer::middle_layer(int i, int o):
        output_layer(i,o)
{

}

middle_layer::~middle_layer()
{
        delete [num_outputs*num_inputs] weights;
        delete [num_outputs] output_errors;
        delete [num_inputs] back_errors;
        delete [num_outputs] outputs;
}


void middle_layer::calc_error()
{
        int i, j, k;
        float accumulator=0;
        for (i=0; i<num_inputs; i++)
        {
                k=i*num_outputs;
                for (j=0; j<num_outputs; j++)
                {
```

```
                    back_errors[i]=
                    weights[k+j]*(*(output_errors+j));
                    accumulator+=back_errors[i];
                }
                back_errors[i]=accumulator;
                accumulator=0;
                // now multiply by derivative of sigmoid squashing function, which is
                // just the input*(1-input)
                back_errors[i]*=(*(inputs+i))*(1-(*(inputs+i)));
        }
}


network::network()
{
        position=0L;
}

network::~network()
{
        int i,j,k;
        i=layer_ptr[0]->num_outputs;                    // inputs
        j=layer_ptr[number_of_layers-1]->num_outputs;    //outputs
        k=MAX_VECTORS;
        delete [(i+j)*k]buffer;
}


void network::set_training(const unsigned & value)
{
        training=value;
}
unsigned network::get_training_value()
{
        return training;
}


void network::get_layer_info()
{
        int i;
        // Get layer sizes for the network
        number_of_layers = 4;
        layer_size[0] = 4;
        layer_size[1] = 8;
        layer_size[2] = 8;
        layer_size[3] = 2;

/*
cout <<"\n  Enter the total number of layers for your network [3-5].";
cout <<"\n  3 means 1 hidden layer, 4 means 2, 5 means 3     = ";
cin >> number_of_layers;

cout <<"\n          Enter the layer sizes separated by spaces.";
```

```
cout <<"\n  Example; for 3 layers having 3 input neurons, 6 hidden";
cout <<"\n  neurons, and 1 output neuron just type 3 6 1 ";
cout <<"\n  Enter please                                = ";

for (i=0; i<number_of_layers; i++)
        {
        cin >> layer_size[i];
        }
*/
// -------------------------------------------------------
// size of layers:
// input_layer          layer_size[0]
// output_layer         layer_size[number_of_layers-1]
// middle_layers        layer_size[1]
//optional:             layer_size[number_of_layers-3]
//optional:             layer_size[number_of_layers-2]
//-------------------------------------------------------
}


void network::set_up_network()
{
        int i,j,k;
        // Construct the layers
        layer_ptr[0] = new input_layer(0,layer_size[0]);
        for (i=0;i<(number_of_layers-1);i++)
        {
                layer_ptr[i+1] =
                new middle_layer(layer_size[i],layer_size[i+1]);
        }
        layer_ptr[number_of_layers-1] = new
        output_layer(layer_size[number_of_layers-2],layer_size[number_of_layers-1]);
        for (i=0;i<(number_of_layers-1);i++)
        {
                if (layer_ptr[i] == 0)
                {
                        cout << "insufficient memory\n";
                        cout << "use a smaller architecture\n";
                        exit(1);
                }
        }

        // Connect the layers
        // set inputs to previous layer outputs for all layers, except the input layer
        for (i=1; i< number_of_layers; i++)
                layer_ptr[i]->inputs = layer_ptr[i-1]->outputs;

        // for back_propagation, set output_errors to next layer
        //back_errors for all layers except the output layer and input layer
        for (i=1; i< number_of_layers -1; i++)
                ((output_layer *)layer_ptr[i])->output_errors =
                        ((output_layer *)layer_ptr[i+1])->back_errors;
```

```
        // define the IObuffer that caches data from  the datafile
        i=layer_ptr[0]->num_outputs;                    // inputs
        j=layer_ptr[number_of_layers-1]->num_outputs;   //outputs
        k=MAX_VECTORS;
        buffer=new
        float[(i+j)*k];
        if (buffer==0)
        {
                cout << "insufficient memory for buffer\n";
                exit(1);
        }
}


void network::read_weights(FILE * weights_file_ptr)
{
        int i;
        for (i=1; i<number_of_layers; i++)
                ((output_layer *)layer_ptr[i])->read_weights(i,weights_file_ptr);
}


void network::list_outputs()
{
        int i;
        for (i=1; i<number_of_layers; i++)
        {
                cout << "layer number : " <<i<< "\n";
                ((output_layer *)layer_ptr[i])->list_outputs();
        }
}


void network::write_outputs(FILE *outfile)
{
        int i, ins, outs;
        ins=layer_ptr[0]->num_outputs;
        outs=layer_ptr[number_of_layers-1]->num_outputs;
        float temp;
        fprintf(outfile,"for input vector:\n");
        printf("for input vector:\n");
        for (i=0; i<ins; i++)
        {
                temp=layer_ptr[0]->outputs[i];
                fprintf(outfile,"%f ",temp);
                printf("%f ",temp);
        }
        fprintf(outfile,"\noutput vector is:\n");
        printf("\noutput vector is:\n");
        for (i=0; i<outs; i++)
        {
                temp=layer_ptr[number_of_layers-1]->outputs[i];
                fprintf(outfile,"%f ",temp);
                printf("%f ",temp);
```

```
        }
        if (training==1)
        {
                fprintf(outfile,"\nexpected output vector is:\n");
                printf("\nexpected output vector is:\n");
                for (i=0; i<outs; i++)
                {
                        temp=((output_layer *)(layer_ptr[number_of_layers-1]))->expected_values[i];
                        fprintf(outfile,"%f  ",temp);
                        printf("%f  ",temp);
                }
        }
        fprintf(outfile,"\n----------\n");
        printf("\n----------\n");
}


// This is the procedure to get the final output value of the network..
//Note that  this is the magnitude and direction of the applied voltage.
float network::get_final_output()
{
        float voltage,sign;
        sign=layer_ptr[number_of_layers-1]-> outputs[1];
        if (sign > 0.5)
                sign = 1.0;
        else
                sign = -1.0;
        voltage=layer_ptr[number_of_layers-1]->    outputs[0]*sign;
        return voltage;
}


// A procedure to get real data from adc
int network::get_data_from_adc(float a,float x_d,float a_d,float d_d)
{
        buffer[0] = a;
        buffer[1] = x_d;
        buffer[2] = a_d;
        buffer[3] = d_d;
        return (1);
}


void network::set_up_pattern(int buffer_index)
{
// read one vector into the network
        int i, k;
        int ins, outs;
        ins=layer_ptr[0]->num_outputs;
        outs=layer_ptr[number_of_layers-1]->num_outputs;
        if (training==1)
                k=buffer_index*(ins+outs);
        else
                k=buffer_index*ins;
```

```
        for (i=0; i<ins; i++)
                ((input_layer*)layer_ptr[0])->orig_outputs[i]=buffer[k+i];
        if (training==1)
        {
                for (i=0; i<outs; i++)
                        ((output_layer *)layer_ptr[number_of_layers-1])->
                                expected_values[i]=buffer[k+i+ins];
        }
}


void network::forward_prop()
{
        int i;
        for (i=0; i<number_of_layers; i++)
        {
                layer_ptr[i]->calc_out();               //A polymorphic function
        }
}
```

APPENDIX D

The Fuzzy Logic On-Line Program for the Flexible Pole-cart Balancing System

Important procedures in the program:

1. **init_flbrm_fuzzy_system( )** - A procedure to initialise all the parameters used in fuzzy logic system.
2. **init_flbrm_rules( )** - A procedure to initialise values of fuzzy logic rules.
3. **init_flbrm_mem_fns( )** - A procedure to initialise values of fuzzy logic membership functions.
4. **init_trapz( )** - A procedure to assign locations of points of trapezoid used as membership function.
5. **main_loop( )** - A procedure used to operate the controller and interact with the user.
6. **reset_ad_da_con( )** - A procedure used to prepare the analog/digital digital/analog converter for operation.
7. **initialize_data( )** - A procedure used to initialize the values of the sensors.
8. **set_clock_frequency( )** - A procedure used to set the clock frequency for real time operation.
9. **newtimer( )** - A tc++ built in procedure used to instruct the interrupt vectors to operate in real time.
10. **enable( )** - A tc++ buit in procedure used to enable the interrupt service routine (isr). At this point newtimer( ) start operating in real time and sensors are getting data from the plant.
11. **get_voltage_from_fuzzy_con( )** - A procedure to get the voltage needed to operate the actuator and control the system from a fuzzy logic controller.
12. **fuzzy_logic_force( )** - A procedure that will assign appropriate input variables to fuzzify and obtain the required output value.

13. **fuzzy_system( )** - A procedure used to obtain the required output value by deffuzzification using centroid method.
14. **main_menu( )** - A procedure to display user options on operations the controller.
15. **printval( )** - A procedure to display the status of the system. (i.g. position and velocity of the cart, the pole angle, and the pole deflection).
16. **save_data_to_file( )** - A procedure used to save the informations needed to examine the performance of the controller. The data are saved in MATLAB format and are ready for MATLAB graphical representation.

C Language Class/Structures Used in the Program

```
/* File FLOGIC5.H  Fuzzy logic header file  */

#ifndef FLOGIC5_H
#define FLOGIC5_H

#include "uttypes.h"

#define MAX_NO_OF_INPUTS 8                      // The number of input variables that a
fuzzy
                                                // logic controller operates.
#define MAX_NO_OF_INP_REGIONS 5                 // The number of membership functions that
                                                // the controller operates.
#define MAX_NO_OF_OUTPUT_VALUES 8               // The number of possible fuzzy logic
                                                // output  value.


typedef enum {regular,left,right} trapz_type;           // The specific sides of a trapezoid
use as a
                                                // membership function.
typedef struct trapezoid                        // Components of a trapezoid.
{
        trapz_type tp;
        float a,b,c,d;                          // Exact end points of a trapezoid.
        float l_slope,r_slope;                  // Slope of the left and right lines.
};


typedef struct rule                             // The components of a rule.
{
        short inp_index[MAX_NO_OF_INPUTS],      // Holds the index number of an input
                                                //  variables.
        inp_fuzzy_set[MAX_NO_OF_INPUTS],        // Holds the index number of fuzzy rules.
        out_fuzzy_set;                          // The fuzzy set output value.
};
```

```
typedef struct fuzzy_system_rec                    // The complete structure of the fuzzy
{                                                  // logic system
        short allocated;
        struct trapezoid inp_mem_fns [MAX_NO_OF_INPUTS] [MAX_NO_OF_INP_REGIONS];
        far struct rule *rules;
        int no_of_inputs,no_of_inp_regions,no_of_rules,no_of_outputs;
        float output_values[MAX_NO_OF_OUTPUT_VALUES];
};


typedef struct p_cart_state_rec                    // Holds the data of the plant
{
        float ang,                                 // Pole angle
        ang_vel,                                   // Pole velocity
        x_pos,                                     // Cart position
        x_dot,                                     // Cart velocity
        deflec,                                    // Pole deflection
        def_vel;                                   // Pole deflection velocity
};
```

The Fuzzy Logic Controller Program

by:
Elmer P. Dadios
Manufacturing Engineering Department
Loughborough University of technology, UK
August 1995

```
#pragma inline                                  // declaration for inline assembly
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <bios.h>
#include <dos.h>
#include <io.h>
#include <float.h>
#include <stdlib.h>
#include <graphics.h>
#include <d:\research\neu\dt2811.drv>           // For data translation AD/DA converter
extern void interrupt newtimer();               // Declare other functions for real time operation
static void interrupt (*oldtimer)();

#include "fl_fpb9.c"                            // Contains fuzzy logic procedures

const int no_of_data = 3000;                    // Number of result values to store
int i;
unsigned int ctr,ctr1;
float x_data[3000],a_data[3000],d_data[3000],v_data[3000];  // Array for result values

float alpha,alpha_d,x,x_d,def,def_d,voltage;                    // Plant variables
float alpha_pf,alpha_p,alpha_f,x_pf,x_p,x_f,def_pf,def_p,def_f,def_df;      //filtered and raw data
float cal_constant_x,cal_constant_alpha,cal_constant_def;       // sensor constant value
float wcut,fcut,kf1,kf2,ts,fsamp;               // for low pass filter */
int ivolts,alpha_int,x_int,def_int,motor_off;   // Integer values of raw data for AD/DA converter
float x_bias,alpha_bias,def_bias;               // offset/initial value of sensor in volts
int gdriver,gmode,cdiv,cdiv_lo,cdiv_hi;         // graphics & clock devider
float timen,timep,u_freq,del_time,base_freq;            // frequency for printing realtime data to
screen
float volts_o;                                  // Actual voltage value for actuator
int volt_int;                                   // Integer voltage value for AD/DA converter


void initialize_data(void)
{
        // sampling frequency
        fsamp = 200.0;
        ts = 1/fsamp;
        ctr = 0; ctr1 = 0;

        /* setup  printval() frequency */
        u_freq = 5;                             // Refresh realtime data on screen at 5 Hz
        del_time = 1./u_freq;
```

```
        // sensor calibration constant
        cal_constant_x = 91/10.00;                  // 91 cm over entire range
        cal_constant_alpha = 352.0/24;              // full turn over 24 volts
        cal_constant_def = 2.54;  //1 inch per volt = 2.54 cm per volt but you should calibrate first
        x_bias = 0;                                 // cart displacement offset
        alpha_bias = -0.1;                                  // Pole angle offset
        def_bias =  -0.6;                           // Pole deflection offset

        // other parameters
        motor_off = 1;                              // start with motor off
        timen = 0;                                  // initialize real time
        timep = 0.1;

        /* lowpass filter */
        fcut = 2.0;
        wcut = 2.0*3.14*fcut;
        kf1 = wcut*ts/(2+wcut*ts);
        kf2 = (wcut*ts-2)/(wcut*ts+2);
}


// procedure to display plant data on screen
void printval(void)
{
        textcolor(GREEN);
        gotoxy(1,1); cprintf("    REAL TIME DATA");
        gotoxy(1,2); cprintf("Real time (sec)    = %6.1f",timen);
        gotoxy(1,3); cprintf("Voltage applied    = %6.4f",voltage);
        gotoxy(1,4); cprintf("Cart position (cm) = %6.4f",x);
        gotoxy(1,5); cprintf("Cart velocity (cm) = %6.4f",x_d);
        gotoxy(1,6); cprintf("Pole angle (deg)   = %6.4f",alpha);
        gotoxy(1,7); cprintf("P_angle velocity   = %6.4f",alpha_d);
        gotoxy(1,8); cprintf("Pole deflection    = %6.4f",def);
        gotoxy(1,9); cprintf("Velocity of def    = %6.4f",def_d);
        if ((abs(ctr) < no_of_data)&&(motor_off < 0))
        {
                gotoxy(15,20);
                cprintf("ctr = %d",ctr);
        }
        gotoxy(15,18); /* move cursor to the choice position */
}       /* printval */


/* procedure to reset the ad da controllers */
void reset_ad_da_con(void)
{
        reset_ad();
        reset_da();
        daout(0,2048);
        daout(1,2048);
        daout(2,2048);
}
```

```
extern void interrupt far newtimer()
{
        char *data87[94];
        asm fsave data87
        _clear87();
        timen= timen+ts;

        // Sensor measurements
        x_pf = x_f;                                     // save previous filtered data
        x_p = x;                                        // save previous raw data
        x_int = adin(4);                                // sample the A to D channel
        x = itov(x_int)*cal_constant_x - x_bias;        // actual cart displacement
        x_f = kf1*(x+x_p)-kf2*x_f;                      // digital low pass filtering
        x_d = (x_f-x_pf)/ts;                            // actual cart velocity

        alpha_pf = alpha_f;                             // save previous filtered data
        alpha_p = alpha;                                // save previous raw data
        alpha_int = adin(5);                            // sample the A to D channel
        alpha = itov(alpha_int)*cal_constant_alpha - alpha_bias;      // actual pole angle
        alpha_f = kf1*(alpha+alpha_p)-kf2*alpha_f;      // digital low pass filtering
        alpha_d = (alpha_f-alpha_pf)/ts;                // actual pole angle velocity

        def_pf = def_f;                                 // save previous filtered data
        def_p = def;                                    // save previous raw data
        def_int = adin(10);                             // sample the A to D channel
        def = itov(def_int)*cal_constant_def -def_bias; // actual pole deflection
        def_f = kf1*(def+def_p)-kf2*def_f;              // digital low pass filtering
        def_df = def_df+def_d*wcut*ts;
        def_d = wcut*def - def_df;                      // actual pole deflection

        // run the fuzzy logic controller
        get_voltage_from_fuzzy_con(alpha,alpha_d,x,x_d,def,def_d,&voltage);

        // Insure the voltage to be on actuator's capacity
        if(voltage > 4.95) voltage = 4.95;
        if(voltage < -4.95) voltage = -4.95;

        // prepare for saving results to external file
        if ((abs(ctr) < no_of_data)&&(motor_off < 0))
        {
                        ctr++;
                        ctr1 = ctr;
                        x_data[ctr1] = x;
                        a_data[ctr1] = alpha;
                        d_data[ctr1] = def;
                        v_data[ctr1] = voltage;
        }

        if(motor_off >0)
                voltage = 0;
```

```
        // apply voltage to actuator
        volts_o = voltage;
        volt_int = volts_o*2048/5.0+2047;
        daout(0,volt_int);

        asm frstor data87
}


// procedure to go back to old timer
void get_old_timer(void)
{
        disable();
        setvect(0xc1,oldtimer);
        enable();
        outportb(0x43,0x36);
        outportb(0x40,0xff);
        outportb(0x40,0xff);
        daout(0,2048);
        daout(1,2048);
        daout(2,2048);
}       /* get_old_timer */


// procedure to set the clock frequency for real time
void set_clock_frequency(void)
{
        base_freq = 1193000.0;          // 1.193 MHz is the base frequency of the clock in an AT
        cdiv = ceil(base_freq*ts);      // setup clock divider
        outportb(0x43,0x36);            // setup clock number 2
        cdiv_hi = cdiv / 255;           // high byte of cdiv
        cdiv_lo = fmod(cdiv,255);       // low byte of cdiv
        outportb(0x40,cdiv_lo);         // write out low byte
        outportb(0x40,cdiv_hi);         // then high byte

        disable();                      // diable interrupts
        oldtimer = getvect(0x1c);       // save old isr address
        setvect(0x1c,newtimer);         // setup the new isr
        enable();                               // enable interrupts, at this point newtimer starts
running
}       // set_clock_frequency


// procedure to write menu for user options
void main_menu(void)
{
        clrscr();
        textcolor(RED + BLINK);
        gotoxy(1,15);
        if (motor_off > 0)
                cprintf("MOTOR IS OFF");
        else
                cprintf("MOTOR IS ON");
        textcolor(BLUE);
```

```
        gotoxy(1,16);
        cprintf("[Q] to quit");
        gotoxy(1,17);
        cprintf("[S] to start/stop motor");
        gotoxy(1,18); textcolor(CYAN);
        cprintf("Your choice =  ");
}       /* main_menu */


// procedure running the system with the interaction of the user
void main_loop(void)
{   char choice;

        clrscr();
        reset_ad_da_con();
        _fpreset();                                     // reset floating point processor
        initialize_data();
        set_clock_frequency();
        main_menu();
        do
        {
                choice = '';
                if (kbhit())
                {
                        choice = getch();               // get the character
                        flushall();                     // flush the keyboard buffer
                }
                if ((choice=='Q')||(choice=='q'))       // quit program
                        get_old_timer();
                if ((choice=='S')||(choice=='s'))       // stop operationt
                {
                        motor_off = -1*motor_off;// initially motor is off
                        main_menu();
                }
                if ((timen-timep)>del_time)             // frequency of printing realtime data to
                                                        //screen */
                {
                        timep = timen;
                        printval();
                }
        } while((choice!='Q')&&(choice!='q'));
        textcolor(LIGHTGRAY);
        clrscr();
}       // main_loop


/* Procedure in puting data to external file */
void save_data_to_file(void)
{
        int i;
        FILE *f1;

        if ((f1 = fopen("x_data16.m","w"))==NULL)
        {
```

```
            puts("\ncannot open file x_data");
            exit(1);
        }
        for (i = 1;i<ctr1;i++)
            fprintf(f1,"%5.3f\n",x_data[i]);
        fclose(f1);

        if ((f1 = fopen("a_data16.m", "w"))==NULL)
        {
            puts("\ncannot open file a_data");
            exit(1);
        }
        for (i = 1;i<ctr1;i++)
            fprintf(f1,"%5.3f\n",a_data[i]);
        fclose(f1);

        if ((f1 = fopen("d_data16.m", "w"))==NULL)
        {
            puts("\ncannot open file d_data");
            exit(1);
        }
        for (i = 1;i<ctr1;i++)
            fprintf(f1,"%5.3f\n",d_data[i]);
        fclose(f1);

        if ((f1 = fopen("v_data16.m", "w"))==NULL)
        {
            puts("\ncannot open file d_data");
            exit(1);
        }
        for (i = 1;i<ctr1;i++)
            fprintf(f1,"%5.3f\n",v_data[i]);
        fclose(f1);
}


// This is main body of the program
void main(void)
{
        init_flbrm_fuzzy_system(&g_fuzzy_system);
        main_loop();
        save_data_to_file();
}
```

```
// This is the file that contain the fuzzy logic procedures
// File FL_FPB9.C

#ifndef FL_FPB9_C
#define FL_FPB9_C

#include <math.h>
#include <conio.h>
#include <alloc.h>
#include <stdio.h>
#include "flbrm5.h"                    // Holds the header files and structures used
#include "flprcs5.c"                   // Holds the process procedures
#include "flbini27.c"                  // Holds initialization procedures
#include "flogic5.h"                   // Holds the header files and structures used


float fuzzy_logic_force (struct p_cart_state_rec the_state)
{
        float x[6],y[3],force;

        // assigning input variable
        x[in_theta] = the_state.ang;
        x[in_theta_dot] = the_state.ang_vel;
        x[in_x] = the_state.x_pos;
        x[in_x_dot] = the_state.x_dot;
        x[in_d] = the_state.deflec;
        x[in_d_dot] = the_state.def_vel;

        // get output result for each set of input variables
        y[in_theta] = fuzzy_system(x,g_fuzzy_system,0,13);      // theta & theta_d fuzzy result
        y[in_theta_dot] = fuzzy_system(x,g_fuzzy_system,13,26); // x & x_d fuzzy result
        y[in_x] = fuzzy_system(x,g_fuzzy_system,26,39);         // def & def_d fuzzy result
        y[in_x_dot] = fuzzy_system(y,g_fuzzy_system,39,52);     // y[in_theta] & y[in_x] fuzzy
                                                                // result
        force = fuzzy_system(y,g_fuzzy_system,52,65);           // y[in_theta_dot] & y[in_x_dot]
                                                                //fuzzy result

        return force;
}


void get_voltage_from_fuzzy_con(float a,float a_d,float x,float x_d,
                                float def,float def_d,float *voltage)
{
        struct p_cart_state_rec sys_state_rec;
        float volt;

        sys_state_rec.ang = a;
        sys_state_rec.ang_vel = a_d;
        sys_state_rec.x_pos = x;
        sys_state_rec.x_dot = x_d;
        sys_state_rec.deflec = def;
        sys_state_rec.def_vel = def_d;
```

```
        volt = fuzzy_logic_force(sys_state_rec);

        if ((x>10)&&(a>1.01)&&(x_d>0.01))
          volt += 2.0;  // bring the cart to the center of the track
        if ((x>10)&&(a>1.01)&&(x_d<-0.01))
          volt += 1.1;  // bring the cart to the center of the track
        if ((x<-10)&&(a<-1.01)&&(x_d<-0.01))
          volt += -1.0;  // bring the cart to the center of the track
        if ((x<-10)&&(a<-1.01)&&(x_d>0.01))
          volt += -0.6;  // bring the cart to the center of the track

        if ((a>1.01)&&(a_d>0.01))
                volt += 3.0; //balance the pole
        if ((a<-1.01)&&(a_d<-0.01))
                volt += -3.0; //balance the pole

        if (a>2.31)
                volt += 5.5; //balance the pole
        if (a<-2.31)
                volt += -5.5; //balance the pole


        *voltage = volt;
}
#endif
```

```
// File FLPRCS5.C - this is the file that hold the fuzzy logi c procedures and functions  */

#ifndef FLPRCS5_C
#define FLPRCS5_C

#define TOO_SMALL 1e-6

#include <stdlib.h>
#include <math.h>
#include <conio.h>

#include "flogic5.h"

struct fuzzy_system_rec g_fuzzy_system;

struct trapezoid init_trapz (float x1,float x2,float x3,float x4,trapz_type typ);
float fuzzy_system (float inputs[],struct fuzzy_system_rec fl,int ctr1,int ctr2);
void free_fuzzy_rules (struct fuzzy_system_rec *fz);


// Implementation
struct trapezoid init_trapz (float x1,float x2,float x3,float x4,trapz_type typ)
{
        struct trapezoid trz;

        trz.a = x1;
        trz.b = x2;
        trz.c = x3;
        trz.d = x4;
        trz.tp = typ;
        switch (trz.tp)
        {
        case regular:
                trz.l_slope = 1.0/(trz.b - trz.a);
                trz.r_slope = 1.0/(trz.c - trz.d);
                break;
        case left:
                trz.r_slope = 1.0/(trz.a - trz.b);
                trz.l_slope = 0.0;
                break;
        case right:
                trz.l_slope = 1.0/(trz.b - trz.a);
                trz.r_slope = 0.0;
                break;
        }       // end switch
        return trz;
}       // end function
```

```
float trapz (float x, struct trapezoid trz)
{
        switch (trz.tp)
        {
        case left:
                if (x <= trz.a)
                        return 1.0;
                if (x >= trz.b)
                        return 0.0;
                // a < x < b
                return trz.r_slope * (x - trz.b);
        case right:
                if (x <= trz.a)
                        return 0.0;
                if (x >= trz.b)
                        return 1.0;
                // a < x < b
                return trz.l_slope * (x - trz.a);
        case regular:
                if ((x <= trz.a) || (x >= trz.d))
                        return 0.0;
                if ((x >= trz.b) && (x <= trz.c))
                        return 1.0;
                if ((x >= trz.a) && (x <= trz.b))
                        return trz.l_slope * (x - trz.a);
                if ((x >= trz.c) && (x <= trz.d))
                        return  trz.r_slope * (x - trz.d);
        }       //End switch
        return 0.0;         // should not get to this point
}       // End function


float min_of (float values[],int no_of_inps)
{
        int i;
        float val;
        val = values [0];
        for (i = 1;i < no_of_inps;i++)
        {
                if (values[i] < val)
                val = values [i];
        }
        return val;
}
```

```
float fuzzy_system (float inputs[],struct fuzzy_system_rec fz,int ctr1,int ctr2)
{
        int i,j;
        short variable_index,fuzzy_set;
        float sum1 = 0.0,sum2 = 0.0,weight;
        float m_values[MAX_NO_OF_INPUTS];
        for (i=ctr1;i<ctr2;i++)
        {
                for (j = 0;j < fz.no_of_inputs;j++)
                {
                        variable_index = fz.rules[i].inp_index[j];
                        fuzzy_set = fz.rules[i].inp_fuzzy_set[j];
                        m_values[j] = trapz(inputs[variable_index],
                        fz.inp_mem_fns[variable_index][fuzzy_set]);
                }       // end j
                weight = min_of (m_values,fz.no_of_inputs);
                sum1 += weight * fz.output_values[fz.rules[i].out_fuzzy_set];
                sum2 += weight;
        }       // end i
        if (fabs(sum2) < TOO_SMALL)
                return 0.0;
        return (sum1/sum2);
}       // end fuzzy_system


#endif
```

```
// FLBINIT7.C - this file contains the Initialization procedures

#ifndef FLBINIT7_H
#define FLBINIT7_H

#include <alloc.h>
#include "flbrm5.h"

void init_flbrm_rules (struct fuzzy_system_rec *fl)
{
        const int
        no_of_x_rules = 13;
        int i;
        for (i = 0;i < no_of_x_rules;i++)
        {
                fl->rules[i].inp_index[0] = in_theta;
                fl->rules[i].inp_index[1] = in_theta_dot;
        }

        // Regions for theta and theta_dot:
         fl->rules[0].inp_fuzzy_set[0] = in_negl;
         fl->rules[0].inp_fuzzy_set[1] = in_negl;
         fl->rules[0].out_fuzzy_set = out_nl;
         fl->rules[1].inp_fuzzy_set[0] = in_negl;
         fl->rules[1].inp_fuzzy_set[1] = in_ze;
         fl->rules[1].out_fuzzy_set = out_nm;
         fl->rules[2].inp_fuzzy_set[0] = in_negl;
         fl->rules[2].inp_fuzzy_set[1] = in_posl;
         fl->rules[2].out_fuzzy_set = out_ze;

         fl->rules[3].inp_fuzzy_set[0] = in_negm;
         fl->rules[3].inp_fuzzy_set[1] = in_negm;
         fl->rules[3].out_fuzzy_set = out_nm;
         fl->rules[4].inp_fuzzy_set[0] = in_negm;
         fl->rules[4].inp_fuzzy_set[1] = in_posm;
         fl->rules[4].out_fuzzy_set = out_ns;

         fl->rules[5].inp_fuzzy_set[0] = in_ze;
         fl->rules[5].inp_fuzzy_set[1] = in_negl;
         fl->rules[5].out_fuzzy_set = out_ns;
         fl->rules[6].inp_fuzzy_set[0] = in_ze;
         fl->rules[6].inp_fuzzy_set[1] = in_ze;
         fl->rules[6].out_fuzzy_set = out_ze;
         fl->rules[7].inp_fuzzy_set[0] = in_ze;
         fl->rules[7].inp_fuzzy_set[1] = in_posl;
         fl->rules[7].out_fuzzy_set = out_ps;

         fl->rules[8].inp_fuzzy_set[0] = in_posm;
         fl->rules[8].inp_fuzzy_set[1] = in_negm;
         fl->rules[8].out_fuzzy_set = out_ps;
         fl->rules[9].inp_fuzzy_set[0] = in_posm;
         fl->rules[9].inp_fuzzy_set[1] = in_posm;
         fl->rules[9].out_fuzzy_set = out_pm;
```

```
fl->rules[10].inp_fuzzy_set[0] = in_posl;
fl->rules[10].inp_fuzzy_set[1] = in_negl;
fl->rules[10].out_fuzzy_set = out_ze;
fl->rules[11].inp_fuzzy_set[0] = in_posl;
fl->rules[11].inp_fuzzy_set[1] = in_ze;
fl->rules[11].out_fuzzy_set = out_pm;
fl->rules[12].inp_fuzzy_set[0] = in_posl;
fl->rules[12].inp_fuzzy_set[1] = in_posl;
fl->rules[12].out_fuzzy_set = out_pl;

for (i = 0;i < no_of_x_rules;i++)
{
        fl->rules[i + no_of_x_rules].inp_index[0] = in_x;
        fl->rules[i + no_of_x_rules].inp_index[1] = in_x_dot;
}
/* Regions for x and x_dot: */
fl->rules[13].inp_fuzzy_set[0] = in_negl;
fl->rules[13].inp_fuzzy_set[1] = in_negl;
fl->rules[13].out_fuzzy_set = out_nl;
fl->rules[14].inp_fuzzy_set[0] = in_negl;
fl->rules[14].inp_fuzzy_set[1] = in_ze;
fl->rules[14].out_fuzzy_set = out_nm;
fl->rules[15].inp_fuzzy_set[0] = in_negl;
fl->rules[15].inp_fuzzy_set[1] = in_posl;
fl->rules[15].out_fuzzy_set = out_ps;

fl->rules[16].inp_fuzzy_set[0] = in_negm;
fl->rules[16].inp_fuzzy_set[1] = in_negm;
fl->rules[16].out_fuzzy_set = out_nm;
fl->rules[17].inp_fuzzy_set[0] = in_negm;
fl->rules[17].inp_fuzzy_set[1] = in_posm;
fl->rules[17].out_fuzzy_set = out_ns;

fl->rules[18].inp_fuzzy_set[0] = in_ze;
fl->rules[18].inp_fuzzy_set[1] = in_negl;
fl->rules[18].out_fuzzy_set = out_ns;
fl->rules[19].inp_fuzzy_set[0] = in_ze;
fl->rules[19].inp_fuzzy_set[1] = in_ze;
fl->rules[19].out_fuzzy_set = out_ze;
fl->rules[20].inp_fuzzy_set[0] = in_ze;
fl->rules[20].inp_fuzzy_set[1] = in_posl;
fl->rules[20].out_fuzzy_set = out_ps;

fl->rules[21].inp_fuzzy_set[0] = in_posm;
fl->rules[21].inp_fuzzy_set[1] = in_negm;
fl->rules[21].out_fuzzy_set = out_ps;
fl->rules[22].inp_fuzzy_set[0] = in_posm;
fl->rules[22].inp_fuzzy_set[1] = in_posm;
fl->rules[22].out_fuzzy_set = out_pm;

fl->rules[23].inp_fuzzy_set[0] = in_posl;
fl->rules[23].inp_fuzzy_set[1] = in_negl;
fl->rules[23].out_fuzzy_set = out_ns;
fl->rules[24].inp_fuzzy_set[0] = in_posl;
```

```
fl->rules[24].inp_fuzzy_set[1] = in_ze;
fl->rules[24].out_fuzzy_set = out_pm;
fl->rules[25].inp_fuzzy_set[0] = in_posl;
fl->rules[25].inp_fuzzy_set[1] = in_posl;
fl->rules[25].out_fuzzy_set = out_pl;

for (i = 0;i < no_of_x_rules;i++)
{
        fl->rules[i + 2*no_of_x_rules].inp_index[0] = in_d;
        fl->rules[i + 2*no_of_x_rules].inp_index[1] = in_d_dot;
}
/* Regions for deflection and deflection velocity: */
fl->rules[26].inp_fuzzy_set[0] = in_negl;
fl->rules[26].inp_fuzzy_set[1] = in_negl;
fl->rules[26].out_fuzzy_set = out_nl;
fl->rules[27].inp_fuzzy_set[0] = in_negl;
fl->rules[27].inp_fuzzy_set[1] = in_ze;
fl->rules[27].out_fuzzy_set = out_nm;
fl->rules[28].inp_fuzzy_set[0] = in_negl;
fl->rules[28].inp_fuzzy_set[1] = in_posl;
fl->rules[28].out_fuzzy_set = out_ze;

fl->rules[29].inp_fuzzy_set[0] = in_negm;
fl->rules[29].inp_fuzzy_set[1] = in_negm;
fl->rules[29].out_fuzzy_set = out_nm;
fl->rules[30].inp_fuzzy_set[0] = in_negm;
fl->rules[30].inp_fuzzy_set[1] = in_posm;
fl->rules[30].out_fuzzy_set = out_ns;

fl->rules[31].inp_fuzzy_set[0] = in_ze;
fl->rules[31].inp_fuzzy_set[1] = in_negl;
fl->rules[31].out_fuzzy_set = out_ns;
fl->rules[32].inp_fuzzy_set[0] = in_ze;
fl->rules[32].inp_fuzzy_set[1] = in_ze;
fl->rules[32].out_fuzzy_set = out_ze;
fl->rules[33].inp_fuzzy_set[0] = in_ze;
fl->rules[33].inp_fuzzy_set[1] = in_posl;
fl->rules[33].out_fuzzy_set = out_ps;

fl->rules[34].inp_fuzzy_set[0] = in_posm;
fl->rules[34].inp_fuzzy_set[1] = in_negm;
fl->rules[34].out_fuzzy_set = out_ps;
fl->rules[35].inp_fuzzy_set[0] = in_posm;
fl->rules[35].inp_fuzzy_set[1] = in_posm;
fl->rules[35].out_fuzzy_set = out_pm;

fl->rules[36].inp_fuzzy_set[0] = in_posl;
fl->rules[36].inp_fuzzy_set[1] = in_negl;
fl->rules[36].out_fuzzy_set = out_ze;
fl->rules[37].inp_fuzzy_set[0] = in_posl;
fl->rules[37].inp_fuzzy_set[1] = in_ze;
fl->rules[37].out_fuzzy_set = out_pm;
fl->rules[38].inp_fuzzy_set[0] = in_posl;
fl->rules[38].inp_fuzzy_set[1] = in_posl;
```

```
fl->rules[38].out_fuzzy_set = out_pl;

for (i = 0;i < no_of_x_rules;i++)
{
        fl->rules[i + 3*no_of_x_rules].inp_index[0] = in_theta; //this is Y[0]=theta+theta_dot
        fl->rules[i + 3*no_of_x_rules].inp_index[1] = in_x; //this is Y[2]=def+def_dot
}
/* Regions for combined theta_theta_d and d_d_d: */
fl->rules[39].inp_fuzzy_set[0] = in_negl;
fl->rules[39].inp_fuzzy_set[1] = in_negl;
fl->rules[39].out_fuzzy_set = out_nl;
fl->rules[40].inp_fuzzy_set[0] = in_negl;
fl->rules[40].inp_fuzzy_set[1] = in_ze;
fl->rules[40].out_fuzzy_set = out_nm;
fl->rules[41].inp_fuzzy_set[0] = in_negl;
fl->rules[41].inp_fuzzy_set[1] = in_posl;
fl->rules[41].out_fuzzy_set = out_ns;

fl->rules[42].inp_fuzzy_set[0] = in_negm;
fl->rules[42].inp_fuzzy_set[1] = in_negm;
fl->rules[42].out_fuzzy_set = out_nm;
fl->rules[43].inp_fuzzy_set[0] = in_negm;
fl->rules[43].inp_fuzzy_set[1] = in_posm;
fl->rules[43].out_fuzzy_set = out_ns;

fl->rules[44].inp_fuzzy_set[0] = in_ze;
fl->rules[44].inp_fuzzy_set[1] = in_negl;
fl->rules[44].out_fuzzy_set = out_ns;
fl->rules[45].inp_fuzzy_set[0] = in_ze;
fl->rules[45].inp_fuzzy_set[1] = in_ze;
fl->rules[45].out_fuzzy_set = out_ze;
fl->rules[46].inp_fuzzy_set[0] = in_ze;
fl->rules[46].inp_fuzzy_set[1] = in_posl;
fl->rules[46].out_fuzzy_set = out_ps;

fl->rules[47].inp_fuzzy_set[0] = in_posm;
fl->rules[47].inp_fuzzy_set[1] = in_negm;
fl->rules[47].out_fuzzy_set = out_ps;
fl->rules[48].inp_fuzzy_set[0] = in_posm;
fl->rules[48].inp_fuzzy_set[1] = in_posm;
fl->rules[48].out_fuzzy_set = out_pm;

fl->rules[49].inp_fuzzy_set[0] = in_posl;
fl->rules[49].inp_fuzzy_set[1] = in_negl;
fl->rules[49].out_fuzzy_set = out_ps;
fl->rules[50].inp_fuzzy_set[0] = in_posl;
fl->rules[50].inp_fuzzy_set[1] = in_ze;
fl->rules[50].out_fuzzy_set = out_pm;
fl->rules[51].inp_fuzzy_set[0] = in_posl;
fl->rules[51].inp_fuzzy_set[1] = in_posl;
fl->rules[51].out_fuzzy_set = out_pl;

for (i = 0;i < no_of_x_rules;i++)
{
```

```
        fl->rules[i + 4*no_of_x_rules].inp_index[0] = in_theta_dot; //this is Y[1]=x+x_dot
        fl->rules[i + 4*no_of_x_rules].inp_index[1] = in_x_dot; //this is
        Y[3]=theta+theta_d+def+def_d
}
/* Regions for combined x_xd, theta_theta_d and d_d_d: */
fl->rules[52].inp_fuzzy_set[0] = in_negl;
fl->rules[52].inp_fuzzy_set[1] = in_negl;
fl->rules[52].out_fuzzy_set = out_nl;
fl->rules[53].inp_fuzzy_set[0] = in_negl;
fl->rules[53].inp_fuzzy_set[1] = in_ze;
fl->rules[53].out_fuzzy_set = out_nm;
fl->rules[54].inp_fuzzy_set[0] = in_negl;
fl->rules[54].inp_fuzzy_set[1] = in_posl;
fl->rules[54].out_fuzzy_set = out_ns;

fl->rules[55].inp_fuzzy_set[0] = in_negm;
fl->rules[55].inp_fuzzy_set[1] = in_negm;
fl->rules[55].out_fuzzy_set = out_nm;
fl->rules[56].inp_fuzzy_set[0] = in_negm;
fl->rules[56].inp_fuzzy_set[1] = in_posm;
fl->rules[56].out_fuzzy_set = out_ns;

fl->rules[57].inp_fuzzy_set[0] = in_ze;
fl->rules[57].inp_fuzzy_set[1] = in_negl;
fl->rules[57].out_fuzzy_set = out_ns;
fl->rules[58].inp_fuzzy_set[0] = in_ze;
fl->rules[58].inp_fuzzy_set[1] = in_ze;
fl->rules[58].out_fuzzy_set = out_ze;
fl->rules[59].inp_fuzzy_set[0] = in_ze;
fl->rules[59].inp_fuzzy_set[1] = in_posl;
fl->rules[59].out_fuzzy_set = out_ps;

fl->rules[60].inp_fuzzy_set[0] = in_posm;
fl->rules[60].inp_fuzzy_set[1] = in_negm;
fl->rules[60].out_fuzzy_set = out_ps;
fl->rules[61].inp_fuzzy_set[0] = in_posm;
fl->rules[61].inp_fuzzy_set[1] = in_posm;
fl->rules[61].out_fuzzy_set = out_pm;

fl->rules[62].inp_fuzzy_set[0] = in_posl;
fl->rules[62].inp_fuzzy_set[1] = in_negl;
fl->rules[62].out_fuzzy_set = out_ps;
fl->rules[63].inp_fuzzy_set[0] = in_posl;
fl->rules[63].inp_fuzzy_set[1] = in_ze;
fl->rules[63].out_fuzzy_set = out_pm;
fl->rules[64].inp_fuzzy_set[0] = in_posl;
fl->rules[64].inp_fuzzy_set[1] = in_posl;
fl->rules[64].out_fuzzy_set = out_pl;

return;
}
```

```
void init_flbrm_mem_fns (struct fuzzy_system_rec *fl)
{
        /* The X membership functions */
        fl->inp_mem_fns[in_x][in_negl] = init_trapz (-5.0,-0.5,0,0,left);
        fl->inp_mem_fns[in_x][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_x][in_ze] = init_trapz (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_x][in_posm] = init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_x][in_posl] = init_trapz (0.5,5.0,0.0,right);
        /* The X dot membership functions */
        fl->inp_mem_fns[in_x_dot][in_negl] = init_trapz (-5.0,-1.0,0,0,left);
        fl->inp_mem_fns[in_x_dot][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_x_dot][in_ze] = init_trapz  (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_x_dot][in_posm] = init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_x_dot][in_posl] = init_trapz (1.0,5.0,0.0,right);
        /* The theta membership functions */
        fl->inp_mem_fns[in_theta][in_negl] = init_trapz (-5.0,-1.0,-0.0,-0.0,left);
        fl->inp_mem_fns[in_theta][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_theta][in_ze] = init_trapz (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_theta][in_posm] =init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_theta][in_posl] =init_trapz (1.0,5.0,0.0,0.0,right);
        /* The theta dot membership functions */
        fl->inp_mem_fns[in_theta_dot][in_negl] = init_trapz (-5.0,-1.0,-0.0,-0.0,left);
        fl->inp_mem_fns[in_theta_dot][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_theta_dot][in_ze] = init_trapz (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_theta_dot][in_posm] = init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_theta_dot][in_posl] = init_trapz (1.0,5.0,0.0,0.0,right);
        /* The deflection membership functions */
        fl->inp_mem_fns[in_d][in_negl] = init_trapz (-3.0,-1.0,0.0,0.0,left);
        fl->inp_mem_fns[in_d][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_d][in_ze] = init_trapz (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_d][in_posm] =init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_d][in_posl] =init_trapz (1.0,3.0,0,0,right);
        /* The deflection dot membership functions */
        fl->inp_mem_fns[in_d_dot][in_negl] = init_trapz (-3.0,-1.0,0.0,0.0,left);
        fl->inp_mem_fns[in_d_dot][in_negm] = init_trapz (-2.0,-1.0,-1.0,0.0,regular);
        fl->inp_mem_fns[in_d_dot][in_ze] = init_trapz (-0.5,0.0,0.0,0.5,regular);
        fl->inp_mem_fns[in_d_dot][in_posm] = init_trapz (0.0,1.0,1.0,2.0,regular);
        fl->inp_mem_fns[in_d_dot][in_posl] = init_trapz (1.0,3.0,0,0,right);
        return;
}


void init_flbrm_fuzzy_system (struct fuzzy_system_rec *fl)
{
        fl->no_of_inputs = 2;  /* Inputs are handled 2 at a time only */
        fl->no_of_rules = 65;
        fl->no_of_inp_regions = 5;
        fl->no_of_outputs = 7;
        fl->output_values [out_nl] = -4.75;
        fl->output_values [out_nm] = -2.65;
        fl->output_values [out_ns] = -1.35;
        fl->output_values [out_ze] = 0.0;
        fl->output_values [out_ps] = 1.35;
```

```
        fl->output_values [out_pm] = 2.65;
        fl->output_values [out_pl] = 4.75;
        if((fl->rules = (struct rule *) malloc ((size_t)(fl->no_of_rules*sizeof(struct rule))))==NULL)
        {
                printf("\n\nOut of memory. Press any key to exit. ");
                getch();
                exit(1);
        }
        init_flbrm_rules(fl);
        init_flbrm_mem_fns(fl);
        return;
}

#endif
```

APPENDIX E

The Fuzzy-Genetic Algorithm Program for Flexible Pole-cart Balancing System

Important procedures in the program:

1. **init_flga_mem_fns** - Sets up the membership functions using a scheme based on the number of fuzzy sets.
2. **init_flga_setup** - Initialize the input parameters. It assigns names of external files used for the process and data storage.
3. **init_flga_rules** - Allocates memory for the storage of fuzzy rules.
4. **ga_vector_to_fuzzy_rules** - Translates the chromosomes from the genetic algorithm to FAM matrix entries.
5. **fl_evaluation_fn** - Caries out the optimization process of the genetic algorithm.
6. **fl_genetic_training** - Handles the training process of the fuzzy system. It is accomplished by running the genetic algorithm operating on the fuzzy system evaluation function.
7. **run_fuzzy_model** - Handles the details of running the trained fuzzy system.

```
/* File FL_GA_PR4 - Genetic Algorithm-Fuzzy optimization program for flexible pole-cart */
/* balancing problem By: Elmer P. Dadios */
/*Note that some of the techniques used here are based on the public-domain routines in chapter 13*/
/* of Neural Networks and Fuzzy Logic Applications in C/C++, by S. T. Welstead, 1994. */

#include <alloc.h>
#include <string.h>

#include "flginit2.c"
#include "flprcs.c"
#include "gaproc3.c"
#include "utvect.c"
#include "utprcs.c"
#include "trnprocs.c"
#include "utmatrix.c"
#include "utfiles.c"
#include "flfiles.c"
#include "flgeval.c"
#include "flgrun.c"
#include "flgtrain.c"

void print_flga_setup (tflg_setup_rec *fl)
{
  printf("\nno_of_fl_inputs = %d \n",fl->no_of_fl_inputs);
  printf("no_of_fl_inp_regions = %d\n",fl->no_of_fl_inp_regions);
  printf("fuzz_system_file = %s\n",fl->fuzzy_system_file_name);
  printf("training_file  = %s\n",fl->training_file_name);
  printf(" output data_file = %s\n",fl->out_data_file_name);
  return;
} /* end proc */

void print_ga_set_rec(tga_setup_rec *fl)
{
  printf("\npopulation_size = %d \n",fl->population_size);
  printf("vect_len = %d\n",fl->vect_len);
  printf("chrom_len = %d\n",fl->chrom_len);
  printf("max_gens = %d\n",fl->max_gens);
  printf("crossover prob = %f\n",fl->crossover_prob);
  printf("mutation prob = %f\n",fl->mutation_prob);
  return;
} /* end proc */

void print_flga_fuzzy_system(fuzzy_system_rec *fl)
{
  int i;
  printf("\nno_of_rules = %d \n",fl->no_of_rules);
  printf("no_of_inputs = %d \n",fl->no_of_inputs);
  printf("no_of_inp_regions = %d\n",fl->no_of_inp_regions);
  printf("no_of_outputs = %d\n",fl->no_of_outputs);
  for (i=0;i<fl->no_of_outputs;i++)
                printf("fuzzy->output[%d] = %f\n",i,fl->output_values[i]);
  return;
} /* end proc */
```

```
// procedure to show program menu
void menu(char *choice)
{
        clrscr();
        gotoxy(17,5);
        printf("**************************************************");
        gotoxy(17,6);
        printf("*                                                *");
        gotoxy(17,7);
        printf("* Fuzzy Logic with Genetic Algorithm Controller *");
        gotoxy(17,8);
        printf("* for the Flexible Pole-Cart Balancing Problem  *");
        gotoxy(17,9);
        printf("*    By: Elmer P. Dadios & David J. Williams    *");
        gotoxy(17,10);
        printf("*                                                *");
        gotoxy(17,11);
        printf("**************************************************");

        gotoxy(22,15);
        printf("****************************************");
        gotoxy(22,16);
        printf("* Enter 1 for running the controller  *");
        gotoxy(22,17);
        printf("* Enter 2 for training the controller *");
        gotoxy(22,18);
        printf("* Enter 0 to exit program             *");
        gotoxy(22,19);
        printf("****************************************");
        do
        {
                gotoxy(22,20);
                printf("Your choice please : ");
                *choice = getch();
        } while(*choice != '1' && *choice != '2' && *choice != '0');
}

void main(void)
{
        char choice;
        const int pop_size = 54, bit_len = 3;
        int j, gener, no_of_rules = 1;
        tflg_setup_rec g_flg_rec;
        fuzzy_system_rec the_fuzzy_system;


                init_flga_setup(&g_flg_rec);
                for (j=1; j<= g_flg_rec.no_of_fl_inputs;j++)
                        no_of_rules *= g_flg_rec.no_of_fl_inp_regions;
//              initialize_ga_setup(pop_size,bit_len,no_of_rules,&ga_setup_rec);

        do
        {
```

```
            menu(&choice);
            switch(choice)
            {
                    case '0' : break;
                    case '1' : run_fuzzy_model(g_flg_rec);
                               break;
                    case '2' :  if((fi = fopen("ga_stat.dat","w"))==NULL)
                               {
                                        printf("\n\nSorry can not open file ga_stat.dat");
                                        exit(1);
                               }
                    fprintf(fi,"time Gen Max_f  Min_f  Ave_f Sum_fit\n");
                    initialize_ga_setup(pop_size,bit_len,no_of_rules,&ga_setup_rec);
                    fl_genetic_training(&g_flg_rec,&ga_setup_rec,&ga_rec);
                    fclose(fi);
                    break;
            }
        } while (choice != '0');
}
```

This is file flgini2.c. It contains the Initialization functions for Fuzzy Logic-Genetic Modeling

```
#ifndef FLGINIT_C
#define FLGINIT_C

#include <alloc.h>
#include <string.h>

#include "flga.h"

void init_flga_setup (tflg_setup_rec *fl) {
  fl->no_of_fl_inputs = 4;
  fl->no_of_fl_inp_regions = 3;
  fl->no_of_fl_output_values = 8;
  fl->read_in_fuzzy_system = 0;
  fl->norm_in_range = 5.0;
  fl->norm_in_min = -2.5;
  fl->norm_out_range = 2.0;
  fl->norm_out_min = -1.0;
  strcpy(fl->fuzzy_system_file_name,"FL_GA.FZS");
  strcpy(fl->training_file_name,"FL_GA.TRN");
  strcpy(fl->out_data_file_name,"FL_GA.DAT");
  return;
  } /* end proc */

void init_flga_rules (fuzzy_system_rec *fz) {
  int i0,i1,i2,i3,i4,i = 0,j,n = 1,out_value = fz->no_of_outputs/2;
  for (j = 1;j<=fz->no_of_inputs;j++)
     n *= fz->no_of_inp_regions;
  fz->no_of_rules = n;
  fz->rules = (rule *) malloc ((size_t)(fz->no_of_rules*sizeof(rule)));
  for (i0 = 0;i0 < fz->no_of_inp_regions;i0++)
     for (i1 = 0;i1 < fz->no_of_inp_regions;i1++)
           for (i2 = 0;i2 < fz->no_of_inp_regions;i2++)
              for (i3 = 0;i3 < fz->no_of_inp_regions;i3++)
                    // for (i4 = 0;i4 < fz->no_of_inp_regions;i4++)
                    {
                          for (j=0;j<fz->no_of_inputs;j++)
                             fz->rules[i].inp_index[j] = j;
                             fz->rules[i].inp_fuzzy_set[0] = i0;
                             fz->rules[i].inp_fuzzy_set[1] = i1;
                             fz->rules[i].inp_fuzzy_set[2] = i2;
                             fz->rules[i].inp_fuzzy_set[3] = i3;
                             fz->rules[i].inp_fuzzy_set[4] = i4;
                             fz->rules[i].out_fuzzy_set = out_value;
                             i++;
                             } /* end loop */
  return;
  } /* end proc */

void init_flga_mem_fns (tflg_setup_rec fl,fuzzy_system_rec *fz) {
```

```
  int i,j,k;
  float *a,a_inc;
  int no_of_rgns,no_of_pts = 2 * fl.no_of_fl_inp_regions;
  no_of_rgns = no_of_pts - 1;
  a_inc = fl.norm_in_range / no_of_rgns;
  a = (float *)malloc(no_of_pts*sizeof(float));
  a[0] = fl.norm_in_min;
  for (i = 1;i<no_of_pts;i++)
     a[i] = a[i-1] + a_inc;
  /* The input membership functions */
  for (i = 0;i<fl.no_of_fl_inputs;i++) {
     fz->inp_mem_fns[i][0] = init_trapz (a[1],a[2],0,0,left);
     k = 1;
     for (j = 1;j<(fl.no_of_fl_inp_regions - 1);j++) {
           fz->inp_mem_fns[i][j] =
              init_trapz (a[k],a[k+1],a[k+2],a[k+3],regular);
           k += 2;
           } /* end j */
     fz->inp_mem_fns[i][fl.no_of_fl_inp_regions-1] =
           init_trapz (a[k],a[k+1],0,0,right);
     } /* end i */
  return;
  } /* end proc */

void init_flga_fuzzy_system (tflg_setup_rec fl,fuzzy_system_rec *fz) {
    float out_inc, out_val;
    int i;
          fz->no_of_inputs = 4;
          fz->no_of_inp_regions = 3;
          fz->no_of_outputs = 8;
          init_flga_mem_fns(fl,fz);
          init_flga_rules(fz);
          out_inc = fl.norm_out_range/(fz->no_of_outputs - 1);
          out_val = fl.norm_out_min;
          for (i = 0;i<fz->no_of_outputs;i++) {
          fz->output_values [i] = out_val;
          out_val += out_inc;
          } /* end i */
    fz->allocated = TRUE;
    return;
    } /* end proc */

#endif
```

This is file gaproc3. This contains the Functions and procedures for genetic algorithm.

```
#ifndef GAPROCS_C
#define GAPROCS_C

#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <mem.h>
#include <math.h>
#include <time.h>

#include "uttypes.h"
#include "ut.h"
#include "ga1.h"

#define TOO_SMALL 1e-6   /* Used for fitness scaling  */

tga_setup_rec ga_setup_rec;
tga_rec ga_rec;
FILE *fi;

float power_of_2 (int n);
void initialize_ga_rec (tga_rec *ga);
short initialize_genetic_alg (tga_setup_rec *setup,tga_rec *ga,
  float (*the_obj_funct)(ui_vector v));
void generation (tga_setup_rec *setup,tga_rec *ga,
  float (*the_obj_funct)(ui_vector v));
void statistics (tga_setup_rec setup,ppopulation the_population,tga_rec *ga,
  short *new_opt);
void display (int gen,tga_setup_rec setup,tga_rec ga,short *cancel,
  short *toler_flag);
void move_individual (tga_setup_rec setup,individual src_ind,
  individual *dest_ind);
void free_ga_pointers (tga_setup_rec setup,tga_rec *ga);
void initialize_ga_setup (int pop_size,int bit_len,int vect_len,
 tga_setup_rec *the_rec);

/* Implementation  */

void initialize_ga_rec (tga_rec *ga) {
    ga->allocated = 0;
    ga->populations[0] = NULL;
    ga->populations[1] = NULL;
    ga->opt_individual.chrom = NULL;
    ga->opt_individual.raw_fitness = 0.0;
    ga->opt_individual.scaled_fitness = 0.0;
    ga->opt_individual.parent1 = 0;
    ga->opt_individual.parent2 = 0;
    ga->opt_individual.xsite = 0;;
    ga->opt_vector = NULL;
```

```
    ga->g_old = 0;
    ga->g_new = 1;
    ga->opt_fitness = 0.0;
    ga->sum_fitness = 0.0;
    ga->nmutation = 0;
    ga->ncross = 0;
    ga->avg = 0.0;
    ga->max = 0.0;
    ga->min = 0.0;
    return;
    }

float power_of_2 (int n) {
  int i;
  float val;
  val = 1.0;
  if (n > 0) {
    for (i = 1;i <= n;i++)
          val *= 2.0;
    return val;
    }
  if (n < 0) {
    for (i = 1;i <= n;i++)
          val *= 0.5;
    }
  return val;
  } /*  end fn */

short flip (float prob) {
  /* prob is a probability (i.e., between 0 and 1) */
  /* frand() is defined in UT.H */
  if (frand() < prob) return 1;
  return 0;
  } /* end fn */

int rnd(int low_lim,int up_lim) {
  /* return random between low_lim and up_lim  */
  int range,result;
  range = up_lim - low_lim + 1;
  result = random (range);  /* Result is between 0 and range - 1 */
  return  (result + low_lim);
  } /* end fn */

int select (int popsize, float sumfitness,ppopulation the_population) {
  float rand,sum = 0.0;
  int j = 0;
  rand = frand() * sumfitness;
  do {
    sum += (*the_population)[j]->scaled_fitness;
    j++;
    } while ((sum < rand) && (j < popsize));
  return (j - 1);
  } /* end fn */
```

```
short mutation (short val,float pmutation,long int *nmutation) {
   short  mutate;
   mutate = flip (pmutation);
   if (mutate) {
     (*nmutation)++;
     return !val;
     }
   else
     return val;
   } /* end fn */

void crossover (chromosome parent1,chromosome parent2,
   chromosome child1,chromosome child2,
  int lchrom,long int *ncross,long int *nmutation,int *jcross,
  float pcross,float pmutation) {
  /* Cross two parent strings, place in two child strings */
  int j;
  if (flip (pcross)) {
    *jcross = rnd (0,lchrom - 2);
    (*ncross)++;
    }
  else
    *jcross = lchrom - 1;
  for (j = 0;j <= *jcross;j++) {
    child1 [j] = mutation (parent1 [j],pmutation,nmutation);
    child2 [j] = mutation (parent2 [j],pmutation,nmutation);
    } /* end j */
  if (*jcross != (lchrom - 1))
    for (j = *jcross + 1;j <= lchrom - 1;j++) {
          child1 [j] = mutation (parent2 [j],pmutation,nmutation);
          child2 [j] = mutation (parent1 [j],pmutation,nmutation);
          } /* end j */
  return;
  } /* end proc */

float scale_fitness (float x,float fmin,float fmax) {
  if (fabs (fmax - fmin) > TOO_SMALL)
    return (x - fmin)*0.95/(fmax - fmin) + 0.05;
/*  else */
  return 1.0;
  } /* end fn */

void rescale_population_fitness (tga_setup_rec setup,
  tga_rec ga,ppopulation the_population) {
  int j;
  float max_fitness,min_fitness;
  min_fitness = (*the_population)[0]->raw_fitness;
  max_fitness = (*the_population)[0]->raw_fitness;
  for (j = 0;j < setup.population_size;j++) {
    if ((*the_population)[j]->raw_fitness < min_fitness)
                min_fitness = (*the_population)[j]->raw_fitness;
    if ((*the_population)[j]->raw_fitness > max_fitness)
                max_fitness = (*the_population)[j]->raw_fitness;
    }
```

```
 if (ga.opt_individual.raw_fitness > max_fitness)
    max_fitness = ga.opt_individual.raw_fitness;
 if (max_fitness - min_fitness < TOO_SMALL)
    min_fitness = 0.9 * max_fitness;
 for (j = 0;j < setup.population_size;j++)
    (*the_population)[j]->scaled_fitness =
          scale_fitness ((*the_population)[j]->raw_fitness,
          min_fitness,max_fitness);
 return;
 } /* end proc */

void pick_fittest (tga_setup_rec setup,
  ppopulation the_old,ppopulation the_new) {
  int i,j,min_index = 0;
  float min_fitness;
  /* Replace min fitness members of new population with fitter members */
  /* from old population. */
  min_fitness = (*the_new)[0]->raw_fitness;
  for (j = 1;j<setup.population_size;j++)
     if ((*the_new)[j]->raw_fitness < min_fitness) {
           min_fitness = (*the_new)[j]->raw_fitness;
           min_index = j;
           }
  for (i = 0;i<setup.population_size;i++)
     if ((*the_old)[i]->raw_fitness > min_fitness) {
           /* Replace min individual with new individual */
           move_individual (setup,*(*the_old)[i],
              (*the_new)[min_index]);
           /* Determine min of new population */
           min_fitness = (*the_new)[0]->raw_fitness;
           min_index = 0;
           for (j = 1;j<setup.population_size;j++)
              if ((*the_new)[j]->raw_fitness < min_fitness) {
                    min_fitness = (*the_new)[j]->raw_fitness;
                    min_index = j;
                    } /* end if, j */
           } /* if,i */
   return;
   } /* end proc */

void decode (tga_setup_rec setup, chromosome chrom,ui_vector v) {
  /* Decode from chromosome to vector */
  int i,k,count = 0;
  unsigned int accum,apower_of_2;
  int start_pos = 0,end_pos = setup.bit_len;
  for (i = 0;i<setup.vect_len;i++) {
          accum = 0;
          apower_of_2 = 1;
          for (k = start_pos;k < end_pos;k++) {
             if (chrom [count])
               accum += apower_of_2;
               count += 1;
               apower_of_2 *= 2;
               } /* end k */
```

```
        v[i] = accum;
        start_pos += setup.bit_len;
        end_pos += setup.bit_len;
        } /* end i */
  return;
  } /* end proc */

void generation (tga_setup_rec *setup,tga_rec *ga,
  float (*the_obj_funct)(ui_vector v)) {
  /* Note: Population size must be even  */
  int j = 0,k,mate1,mate2,jcross;
  ui_vector v;
  v = allocate_ui_vector (0,setup->vect_len - 1);
  do {
    mate1 = select (setup->population_size,
            ga->sum_fitness,ga->populations[ga->g_old]);
    mate2 = select (setup->population_size,
            ga->sum_fitness,ga->populations[ga->g_old]);
    crossover ((*(ga->populations)[ga->g_old])[mate1]->chrom,
                  (*(ga->populations)[ga->g_old])[mate2]->chrom,
           (*(ga->populations)[ga->g_new])[j]->chrom,
           (*(ga->populations)[ga->g_new])[j+1]->chrom,
           setup->chrom_len,&(ga->ncross),&(ga->nmutation),&jcross,
           setup->crossover_prob,
           setup->mutation_prob);
    for (k = j;k <= j + 1;k++) {
          decode (*setup,(*(ga->populations)[ga->g_new])[k]->chrom,v);
          (*(ga->populations)[ga->g_new])[k]->raw_fitness =
                the_obj_funct (v);
          (*(ga->populations)[ga->g_new])[k]->parent1 = mate1;
          (*(ga->populations)[ga->g_new])[k]->parent2 = mate2;
          (*(ga->populations)[ga->g_new])[k]->xsite = jcross;
          }
    j += 2;
   } while (j < setup->population_size);
   free_ui_vector (v,0);
   if (setup->pick_fittest)
     pick_fittest (*setup,(ga->populations)[ga->g_old],
                   (ga->populations)[ga->g_new]);
   rescale_population_fitness (*setup,*ga,ga->populations[ga->g_new]);
   return;
   } /* end proc */

void statistics (tga_setup_rec setup,ppopulation the_population,tga_rec *ga,
  short *new_opt) {
  int j;
  *new_opt = FALSE;
  ga->sum_fitness = (*the_population)[0]->scaled_fitness;
  ga->min = (*the_population)[0]->raw_fitness;
  ga->max = (*the_population)[0]->raw_fitness;
  if (ga->max > ga->opt_individual.raw_fitness) {
    move_individual (setup,*(*the_population)[0],&(ga->opt_individual));
    *new_opt = TRUE;
    }
```

```
  for (j = 1;j<setup.population_size;j++) {
     ga->sum_fitness += (*the_population)[j]->scaled_fitness;
     if ((*the_population)[j]->raw_fitness > ga->max)
             ga->max = (*the_population)[j]->raw_fitness;
     if ((*the_population)[j]->raw_fitness < ga->min)
             ga->min = (*the_population)[j]->raw_fitness;
     if (ga->max > (ga->opt_individual).raw_fitness)
           move_individual (setup,*(*the_population)[j],&(ga->opt_individual));
     } /* end j */
  if ((ga->opt_individual).raw_fitness > ga->opt_fitness) {
    decode (setup,ga->opt_individual.chrom,ga->opt_vector);
    ga->opt_fitness = ga->opt_individual.raw_fitness;
    *new_opt = TRUE;
    }
  ga->avg = ga->sum_fitness/setup.population_size;
  return;
  } /* end proc */

void display_vector (ui_vector v,int vlen) {
  int i;
  int display_len = vlen;
  const max_display_len = 8;
  if (display_len > max_display_len)
    display_len = max_display_len;
  cprintf ("\r\n");
  for (i = 0;i<display_len;i++)
     cprintf("%6u ",v[i]);
  cprintf ("\r\n");
  return;
  } /* end proc */

void display (int gen,tga_setup_rec setup,tga_rec ga,short *cancel,
  short *toler_flag)
{
  time_t secsnow;
  int j;
  unsigned gen_time;
  *toler_flag = FALSE;
  clrscr();
  cprintf ("Gen: %3d\r\n",gen);
  for (j = 0;j<setup.population_size;j++) {
     cprintf ("\r\n%3d: (%3d,%3d) %3d ",
                    j,(*ga.populations[ga.g_new])[j]->parent1,
                    (*ga.populations[ga.g_new])[j]->parent2,
                    (*ga.populations[ga.g_new])[j]->xsite);
     cprintf ("Fitness: %6.4f, Scaled: %6.4f",
                  (*ga.populations[ga.g_new])[j]->raw_fitness,
                  (*ga.populations[ga.g_new])[j]->scaled_fitness);
     } /* j */
  cprintf ("\r\nGeneration %3d\r\n",gen);
  cprintf ("Max: %6.4f, Min: %6.4f, Scaled Avg: %6.4f\r\n",
    ga.max,ga.min,ga.avg);
  cprintf ("Sum: %6.4f, Nmutation: %ld, Ncross: %ld\r\n\n",
    ga.sum_fitness,ga.nmutation,ga.ncross);
```

```
  time(&secsnow);
  gen_time = secsnow;
  fprintf(fi,"%d %d %8.4f %8.4f %8.4f %8.4f\n",gen_time,gen,ga.max,ga.min,ga.avg,
                ga.sum_fitness);
  if (fabs (MAX_GA_OBJ_FN_VALUE - ga.max) < setup.tolerance) {
    *toler_flag = TRUE;
    cprintf ("\r\nThe following chromosomes are within fitness tolerance "
          "(%10.8f): \r\n",setup.tolerance);
    for (j = 0;j< setup.population_size;j++)
          if (fabs (MAX_GA_OBJ_FN_VALUE -
            (*ga.populations[ga.g_new])[j]->raw_fitness) <
                setup.tolerance)
            cprintf ("%2d: Fitness: %6.4f\r\n",j,
                (*ga.populations[ga.g_new])[j]->raw_fitness);
    }
  cprintf("\r\nOptimum Individual Fitness: %7.4f",
          ga.opt_individual.raw_fitness);
  cprintf("\r\nOptimum Vector Fitness: %7.4f",ga.opt_fitness);
  display_vector (ga.opt_vector,setup.vect_len);
  cprintf ("\r\nPress esc to cancel: ");
  if (kbhit() && (getch () == ESC_KEY)) *cancel = TRUE;
  else *cancel = FALSE;
  return;
  } /* end proc */

void display_stats (int gen, tga_setup_rec setup,tga_rec ga) {
  clrscr();
  cprintf ("\r\n");
  cprintf ("Generation %2d\r\n",gen);
  cprintf ("Max: %6.4f, Min: %6.4f, Avg: %6.4f\r\n",
    ga.max,ga.min,ga.avg);
  cprintf ("Sum: %6.4f, Nmutation: %d, Ncross: %d\r\n\n",
    ga.sum_fitness,ga.nmutation,ga.ncross);
  if (fabs (MAX_GA_OBJ_FN_VALUE - ga.opt_individual.raw_fitness)
          < setup.tolerance) {
    cprintf ("\r\nOptimum individual is within fitness tolerance  "
          "(%10.8f): \r\n:",setup.tolerance);
    cprintf ("Opt Individual Fitness: %6.4f\r\n",
        ga.opt_individual.raw_fitness);
    } /* end if */
   return;
   } /* end proc */

void move_individual (tga_setup_rec setup,individual src_ind,
  individual *dest_ind) {
  /* Move content of src_ind to dest_ind */
  /* Need to move contents of chrom, not the pointer value chrom ! */
  memmove(dest_ind->chrom,src_ind.chrom,
    (sizeof(src_ind.chrom[0]))*setup.chrom_len);
  dest_ind->raw_fitness = src_ind.raw_fitness;
  dest_ind->scaled_fitness = src_ind.scaled_fitness;
  dest_ind->parent1 = src_ind.parent1;
  dest_ind->parent2 = src_ind.parent2;
  dest_ind->xsite = src_ind.xsite;
```

```
  return;
  } /* end proc */

short allocate_population (tga_setup_rec setup,
  ppopulation the_population) {
  int j;
  *the_population =
    (population)malloc(setup.population_size*sizeof(pindividual));
  if (!(*the_population)) return 0;
  for (j = 0;j<setup.population_size;j++) {
     (*the_population)[j] = (pindividual) malloc(sizeof(individual));
     if (!(*the_population)[j]) return 0;
     (*the_population)[j]->chrom =
                    (short *)malloc(setup.chrom_len*sizeof(short));
     if (!((*the_population)[j]->chrom)) return 0;
     }
  return 1;
  } /* end proc */

void free_population (tga_setup_rec setup,
  ppopulation the_population) {
  int j;
  if (the_population)
  for (j = 0;j<setup.population_size;j++) {
     if ((*the_population)[j]->chrom) free ((*the_population)[j]->chrom);
     if ((*the_population)[j]) free ((*the_population)[j]);
     }
  return;
  } /* end proc */

void initialize_population (tga_setup_rec setup,tga_rec *ga,
  ppopulation the_population, pindividual the_opt,
  float (*the_obj_funct)(ui_vector v)) {
  int i,j,k;
  float max_fitness;
  int opt_index;
  ui_vector v;
  v = allocate_ui_vector (0,setup.vect_len - 1);
  /* Initialize ga.opt_vector */
  ga->opt_vector = allocate_ui_vector (0,setup.vect_len - 1);
  for (j = 0;j<setup.population_size;j++) {
     for (k = 0;k < setup.chrom_len;k++)
           (*the_population)[j]->chrom [k] = flip (0.5);
     decode (setup,(*the_population)[j]->chrom,v);
     (*the_population)[j]->raw_fitness = the_obj_funct (v);
     (*the_population)[j]->scaled_fitness = 0.0;
     (*the_population)[j]->parent1 = 0;
     (*the_population)[j]->parent2 = 0;
     (*the_population)[j]->xsite = 0;
     cprintf ("\r\nIndividual %2d initialized.",j);
     } /* end j */
  max_fitness = (*the_population)[0]->raw_fitness;
  opt_index = 0;
  for (j = 1;j < setup.population_size;j++)
```

```
    if (max_fitness < (*the_population)[j]->raw_fitness) {
          max_fitness = (*the_population)[j]->raw_fitness;
          opt_index = j;
          }
  move_individual(setup,*((*the_population)[opt_index]),the_opt);
  decode (setup,the_opt->chrom,ga->opt_vector);
  ga->opt_fitness = max_fitness;
  rescale_population_fitness (setup,*ga,the_population);
  free_ui_vector (v,0);
  return;
  } /* end proc */

void initial_display (tga_setup_rec setup,tga_rec ga,short *cancel)
{
  clrscr();
  textcolor(YELLOW);
  cprintf ("Population size:       %d\r\n",setup.population_size);
  cprintf ("Chromosome length:     %d\r\n",setup.chrom_len);
  cprintf ("Max No. of Generations: %d\r\n",setup.max_gens);
  cprintf ("Crossover Probability:  %6.4f\r\n",
    setup.crossover_prob);
  cprintf ("Mutation Probability:   %6.4f\r\n",
    setup.mutation_prob);
  cprintf ("\r\nInitial Population Statistics: \r\n\n");
  cprintf ("Max Fitness:     %10.5f\r\n",ga.max);
  cprintf ("Avg Fitness:     %10.5f\r\n",ga.avg);
  cprintf ("Min Fitness:     %10.5f\r\n",ga.min);
  cprintf ("Sum of Fitness:  %10.5f\r\n",ga.sum_fitness);
  cprintf ("\r\n\n\n");
fprintf(fi,"%05d %d %8.4f %8.4f %8.4f %8.4f\n",0,0,ga.max,ga.min,ga.avg,
                    ga.sum_fitness);
  cprintf ("Press key to continue (esc to cancel): ");
  if (getch() == ESC_KEY) *cancel = TRUE;
  else
     *cancel = FALSE;
  return;
  } /* end proc */

short initialize_genetic_alg (tga_setup_rec *setup,tga_rec *ga,
  float (*the_obj_funct)(ui_vector v)) {
  short opt_flag,cancel;
  int i;
  initialize_ga_rec (ga);
  ga->populations[ga->g_old] = (ppopulation)malloc (sizeof(population));
  if (!(ga->populations)[ga->g_old])
  {
          printf("\n Error allocating ga->populations[ga->g_old]\n");
          getch();
          return 0;
  }
  ga->populations[ga->g_new] = (ppopulation)malloc (sizeof(population));
  if (!(ga->populations)[ga->g_new])
  {
          printf("\n Error allocating ga->populations[ga->g_new]\n");
```

```
        getch();
        return 0;
  }
//printf("\n done ... allocating ga->populations[ga->new]\n");
  if (!allocate_population (*setup,ga->populations[ga->g_old]))
  {
        printf("\n Error allocate_population *setup,ga->populations[ga->g_old]\n");
        getch();
        return 0;
  }
//printf("\n done ... allocate_population ga->g_old\n");
  if (!allocate_population (*setup,ga->populations[ga->g_new]))
  {
        printf("\n Error allocate_population *setup,ga->populations[ga->g_new]\n");
        getch();
        return 0;
  }
//printf("\n done ... allocate_population ga->g_new\n");
  ga->opt_individual.chrom =
                (short *)malloc(setup->chrom_len*sizeof(short));
  if (!(ga->opt_individual.chrom))
  {
        printf("\n Error allocating ga->opt_individual.chrom\n");
        getch();
        return 0;
  }
//printf("\n done ... allocating ga->opt_individual.chrom\n");
  ga->allocated = 1;
  initialize_population (*setup,ga,ga->populations[ga->g_old],
    &(ga->opt_individual),the_obj_funct);
  cprintf ("\r\n\nMemory Available: %lu\r\n",(unsigned long)coreleft());
  cprintf ("\r\nDoing statistics for initial population...");
  statistics (*setup,ga->populations[ga->g_old],ga,&opt_flag);
  initial_display (*setup,*ga,&cancel);
  if (cancel) return 0;
  return 1;
  } /* end proc */

void free_ga_pointers (tga_setup_rec setup,tga_rec *ga) {
  if (ga->allocated) {
    free_population (setup,ga->populations[0]);
    free_population (setup,ga->populations[1]);
    free_ui_vector (ga->opt_vector,0);
    }
  ga->allocated = 0;
  cprintf ("\r\nfree_ga_pointers. heapcheck = %d",heapcheck());
  getch();
  return;
  } /* end proc */

void initialize_ga_setup (int pop_size,int bit_len,int vect_len,
 tga_setup_rec *the_rec)
 {
 int i,gener;
```

```
printf("\n\n              Please enter total number of generations : ");
scanf("%d",&gener);
the_rec->population_size = pop_size;
the_rec->bit_len = bit_len;
the_rec->vect_len = vect_len;
the_rec->chrom_len = bit_len*vect_len;
the_rec->max_gens = gener-1;
the_rec->pick_fittest = TRUE;
the_rec->crossover_prob = 0.6;
the_rec->mutation_prob = 1.0/pop_size;
the_rec->tolerance = 0.01;
return;
} /* end proc */

#endif
```

This is file trnprocs.c. This contains Procedures and functions for processing training file.

```
/* Note: All arrays for input and output have starting index 1 (not 0) !*/

#ifndef TRNPROCS_C
#define TRNPROCS_C

#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <math.h>

#include "uttypes.h"
#include "ut.h"
#include "trn.h"

training_rec g_tr_rec;

const float zero_check = 1e-8;

void free_training_pointers (training_rec *tr_rec);
short process_training_file (path_str the_file_name,int no_of_inputs,
  int no_of_outputs,training_rec *tr_rec);
void normalize_data (f_vector x,f_vector norm_x,int no_of_pts,
  f_vector x_max,f_vector x_min,float norm_range,float norm_min);
void normalize_training_set (int no_of_inputs,int no_of_outputs,
  float norm_in_range,float norm_in_min,float norm_out_range,
  float norm_out_min,training_rec tr_rec);
void denormalize_output (f_vector y,f_vector denorm_out,int no_of_outputs,
  float norm_out_range,float norm_out_min,training_rec tr_rec);

/* Implementation */

void free_training_pointers (training_rec *tr_rec) {
   if (tr_rec->allocated) {
        free_matrix2d (tr_rec->training_array,1,1);
        free_matrix2d (tr_rec->correct_answers,1,1);
        free_matrix2d (tr_rec->norm_answers,1,1);
        }
   if (tr_rec->max_min_allocated) {
        free_f_vector (tr_rec->max_in,1);
        free_f_vector (tr_rec->min_in,1);
        free_f_vector (tr_rec->max_out,1);
        free_f_vector (tr_rec->min_out,1);
        }
   tr_rec->allocated = FALSE;
   tr_rec->max_min_allocated = FALSE;
   return;
   } /* end proc */

short check_network_size (int set_in,int set_out,FILE *infile,int line) {
  int n_in,n_out;
```

```
  fscanf (infile,"%d  %d\n",&n_in,&n_out);
  gotoxy (1,line);
  cprintf ("No. of input nodes: %d",n_in);
  gotoxy (1,line + 2);
  cprintf ("No. of output nodes: %d",n_out);
  cprintf ("\r\nSetup inputs: %d  outputs:  %d  \r\n",
    set_in,set_out);
  if ((n_in != set_in) || (n_out != set_out)) {
     gotoxy (1,line + 6);
     cputs ("These numbers do not agree with setup.\r\n"
               "Press any key to cancel: ");
     getch();
     return 0;
     }
  return 1;
  } /* end func */

short process_training_file (path_str the_file_name,int no_of_inputs,
  int no_of_outputs,training_rec *tr_rec) {
#define FIELD_LEN 255
  float value;  /* used with sscanf to avoid potential addressing problems */
  int i,j,k,rec_len;
  const display_line = 7;
  long file_size_in_bytes,remaining_bytes,current_pos;
  char char_buffer[FIELD_LEN],*the_str;
  FILE *the_file;
  tr_rec->allocated = FALSE;
  tr_rec->max_min_allocated = FALSE;
  tr_rec->no_of_training_items = 0;
  tr_rec->no_of_training_items_inv = 1;
  tr_rec->norm_error_scaling = 1;
  tr_rec->training_array = NULL;
  tr_rec->correct_answers = NULL;
  tr_rec->norm_answers = NULL;
  tr_rec->max_in = NULL;
  tr_rec->min_in = NULL;
  tr_rec->max_out = NULL;
  tr_rec->min_out = NULL;
  clrscr();
  gotoxy (1,5);
  file_size_in_bytes = open_input_text_file (&the_file,the_file_name);
  if (file_size_in_bytes == 0) return 0;  /* File name not found */
  if (!check_network_size (no_of_inputs,no_of_outputs,
        the_file,display_line)) return 0;
  cputs ("\r\nReading training/test items...\r\n");
  /* Allocate space max and min arrays */
  tr_rec->max_in = allocate_f_vector (1,no_of_inputs);
  tr_rec->min_in = allocate_f_vector (1,no_of_inputs);
  tr_rec->max_out = allocate_f_vector (1,no_of_outputs);
  tr_rec->min_out = allocate_f_vector (1,no_of_outputs);
  tr_rec->max_min_allocated = TRUE;
  /* Read past "minimum"  */
  the_str = fgets(char_buffer,FIELD_LEN,the_file);
  if (!the_str) return 0;
```

```
/* Read min input data */
for (i=1;i<=no_of_inputs;i++) {
        if (!fgets(char_buffer,FIELD_LEN,the_file))
        {
                        cprintf("\nReading first line error ... press any key. ");
                        getch();
                        return 0;
        }
        sscanf(char_buffer,"%f",&value);
        tr_rec->min_in[i] = value;
        } /* end i */
/* Read min output data */
for (i=1;i<=no_of_outputs;i++) {
        if (!fgets(char_buffer,FIELD_LEN,the_file))
        {
                        cprintf("\nReading line %d error ... press any key. ",i);
                        getch();
                        return 0;
        }
        sscanf(char_buffer,"%f",&value);
        tr_rec->min_out[i] = value;
        }
/* Read past "maximum" */
if (!fgets(char_buffer,FIELD_LEN,the_file)) return 0;
/* Read max input data */
for (i=1;i<=no_of_inputs;i++) {
   if (!fgets(char_buffer,FIELD_LEN,the_file)) return 0;
   sscanf(char_buffer,"%f",&value);
   tr_rec->max_in[i] = value;
   }
/* Read max output data */
for (i=1;i<=no_of_outputs;i++) {
   if (!fgets(char_buffer,FIELD_LEN,the_file)) return 0;
   sscanf(char_buffer,"%f",&value);
   tr_rec->max_out[i] = value;
        }

/* Read past "Training Data" */
if (!fgets(char_buffer,FIELD_LEN,the_file)) return 0;
/* Determine remaining size of file so we can calculate number of */
/*   training items (for memory allocation). */
current_pos = ftell(the_file);
remaining_bytes = file_size_in_bytes - current_pos;
if (remaining_bytes > 0) {
/* First, we need the length of one record */
/* When creating training files, be sure that all data records */
/* in the file have the same length, otherwise this read method */
/* won't work. */
if (!fgets(char_buffer,FIELD_LEN,the_file)) return 0;
rec_len = ftell(the_file) - current_pos;
/* Restore file to start of data */
fseek (the_file,current_pos,SEEK_SET);
tr_rec->no_of_training_items =
        remaining_bytes/((no_of_inputs + no_of_outputs) * rec_len)-1;
```

```
tr_rec->no_of_training_items_inv =
  1.0/(tr_rec->no_of_training_items);
/* Now do dynamic allocation for training items */
tr_rec->training_array =
  matrix2d (1,tr_rec->no_of_training_items,1,no_of_inputs);
tr_rec->correct_answers =
  matrix2d (1,tr_rec->no_of_training_items,1,no_of_outputs);
tr_rec->norm_answers =
  matrix2d (1,tr_rec->no_of_training_items,1,no_of_outputs);
tr_rec->allocated = TRUE;
for (i=1;i<=tr_rec->no_of_training_items;i++)
{
        for (j=1;j<=no_of_inputs;j++)
        {
                if (!fgets(char_buffer,FIELD_LEN,the_file))
                {
                        cprintf("\n Reading training data_in %d,%d error ... press any key. ",i,j);
                        getch();
                        return 0;
                }
                sscanf(char_buffer," %f",&value);
                tr_rec->training_array[i][j] = value;
        } /* end j */
        for (j=1;j<=no_of_outputs;j++)
        {
                if (!fgets(char_buffer,FIELD_LEN,the_file))
                {
                        cprintf("\n Reading training data_out %d,%d error ... press any key. ",i,j);
                        getch();
                        return 0;
                }
                sscanf(char_buffer," %f",&value);
                tr_rec->correct_answers[i][j] = value;
        } /* end j */
} /* end i */
} /* if */

fclose (the_file);
gotoxy (1,wherey() + 2);
cprintf ("Done.");
gotoxy (1,wherey() + 2);
cprintf ("%d training items processed.",tr_rec->no_of_training_items);
gotoxy (1,wherey() + 2);
cprintf ("Press key to continue (esc to cancel): ");
if (getch() == ESC_KEY) return 0;
return 1;
} /* end func */

void normalize_data (f_vector x,f_vector norm_x,int no_of_pts,
  f_vector x_max,f_vector x_min,float norm_range,float norm_min) {
  /* Same normalization routine used for both inputs and outputs */
  /* Assumes vector index range is 1..no_of_pts for all vectors */
  int i;
  for (i=1;i<=no_of_pts;i++) {
```

```
    if (fabs (x_max[i] - x_min[i]) > zero_check) {
          if (x[i] >= x_max[i]) /* Greater than max, set equal to max  */
            norm_x[i] = norm_range + norm_min;
          if (x[i] <= x_min[i]) /* Less than min, set equal to min  */
            norm_x[i] = norm_min;
          if ((x[i] > x_min[i]) && (x[i] < x_max[i])) {
            norm_x[i] = (x[i] - x_min[i])/
                    (x_max[i] - x_min[i]);
            norm_x[i] = norm_range * norm_x[i] + norm_min;
            }
          }
    else
    /* Max = min, so data is constant: */
    norm_x[i] = 0.5 * norm_range + norm_min;
    } /* end i */
   return;
   } /* end func */

void normalize_training_set (int no_of_inputs,int no_of_outputs,
  float norm_in_range,float norm_in_min,float norm_out_range,
  float norm_out_min,training_rec tr_rec) {
  int i,j;
  /* Normalize training set input and output */
  for (i=1;i<=tr_rec.no_of_training_items;i++) {
    normalize_data (tr_rec.training_array[i],
            tr_rec.training_array[i],
            no_of_inputs,tr_rec.max_in,tr_rec.min_in,norm_in_range,
            norm_in_min);
    normalize_data (tr_rec.correct_answers[i],
            tr_rec.norm_answers[i],
            no_of_outputs,tr_rec.max_out,tr_rec.min_out,norm_out_range,
            norm_out_min);
    } /*  end i */
  if (fabs(norm_out_range) > 1e-6)
    tr_rec.norm_error_scaling = tr_rec.no_of_training_items_inv/norm_out_range;
  return;
  } /*  end proc  */

void denormalize_output (f_vector y,f_vector denorm_out,int no_of_outputs,
  float norm_out_range,float norm_out_min,training_rec tr_rec) {
  /* Translate normalized output values to regular output value range. */
  /* Assumes index range for vectors is 1..no_of_pts.  */
  int i;
  for (i=1;i<=no_of_outputs;i++) {
    denorm_out[i] = (y[i] - norm_out_min)/norm_out_range;
    denorm_out[i] = denorm_out[i] *
          (tr_rec.max_out[i] - tr_rec.min_out[i]) +
            tr_rec.min_out[i];
    } /* i */
  return;
  } /* end procedure */

#endif
```

This is file flgrun.c. It contains procedures for running GA-Fuzzy logic model in forward mode

```
#ifndef FLGRUN_C
#define FLGRUN_C

#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <alloc.h>

#include "ut.h"
#include "trn.h"
#include "flga.h"

void run_fuzzy_model (tflg_setup_rec fl) {
  fuzzy_system_rec the_fuzzy_system;
  float fl_out = 0,*pfl_out,denorm_fl_out = 0,*pdenorm_fl_out,
        norm_ans = 0,*pnorm_ans;
  const int one_fl_output = 1;
  training_rec tr_rec = {0.0,1.0,1.0,NULL,NULL,NULL,0,
        NULL,NULL,NULL,NULL};
  int i;
  FILE *outfile;
  /* We need to treat output values as arrays (of size 1) that start   */
  /* at index 1 (so we can interface with the normalization procedures). */
  pfl_out = &fl_out - 1;
  pdenorm_fl_out = &denorm_fl_out - 1;
  pnorm_ans = &norm_ans - 1;
  the_fuzzy_system.allocated = FALSE;
  clrscr ();
  textcolor (YELLOW);
  gotoxy (1,5);
   if (fl.fuzzy_system_file_name[0] == '\0') {
        cprintf ("\r\nFuzzy System File Name is blank.\r\n");
        cputs ("Press key to cancel: ");
        getch();
        return;
        }
   if (fl.training_file_name[0] == '\0') {
        cprintf ("\r\nTraining File Name is blank.\r\n");
        cputs ("Press key to cancel: ");
        getch();
        return;
        }
   if (fl.out_data_file_name[0] == '\0') {
        cprintf ("\r\nOutput Data File Name is blank.\r\n");
        cputs ("Press key to cancel: ");
        getch();
        return;
        }
```

```
  cprintf ("Run Fuzzy Model.");
  gotoxy (1,7);
  cprintf ("Fuzzy System File Name: %s",fl.fuzzy_system_file_name);
  gotoxy (1,9);
  cprintf ("Training File Name: %s",fl.training_file_name);
  gotoxy (1,11);
  cprintf ("Output Data File Name: %s",fl.out_data_file_name);
  gotoxy (1,13);
  cprintf ("Press key to continue (esc to cancel): ");
  if (getch() == ESC_KEY) goto exit_proc;
  clrscr();
  init_flga_fuzzy_system (fl,&the_fuzzy_system);
  if (!process_training_file (fl.training_file_name,
     fl.no_of_fl_inputs,one_fl_output,&tr_rec)) goto exit_proc;
  clrscr();
  gotoxy (1,5);
  cprintf ("Initializing system...");
  normalize_training_set (fl.no_of_fl_inputs,one_fl_output,
     fl.norm_in_range,fl.norm_in_min,fl.norm_out_range,fl.norm_out_min,
     tr_rec);
  if (!read_fuzzy_system (fl.fuzzy_system_file_name,
     fl.no_of_fl_inputs,fl.no_of_fl_inp_regions,
     the_fuzzy_system.no_of_rules,fl.no_of_fl_output_values,
     &the_fuzzy_system)) goto exit_proc;
  if (!open_output_text_file (&outfile,fl.out_data_file_name)) goto exit_proc;
  clrscr();
  for (i = 1;i<=tr_rec.no_of_training_items;i++) {
     fl_out = fuzzy_system (tr_rec.training_array[i],the_fuzzy_system);
     denormalize_output (pfl_out,pdenorm_fl_out,one_fl_output,
           fl.norm_out_range,fl.norm_out_min,tr_rec);
     normalize_data (tr_rec.correct_answers[i],pnorm_ans,one_fl_output,
           tr_rec.max_out,tr_rec.min_out,fl.norm_out_range,fl.norm_out_min);
     cprintf ("\r\n%2d: Fz: %8.5f, Act: %8.5f, Err: %8.5f, NErr: %8.5f",
           i,denorm_fl_out,tr_rec.correct_answers[i][1],
           (denorm_fl_out - tr_rec.correct_answers[i][1]),
           fabs(fl_out - norm_ans));
     fprintf(outfile,"%2d %8.5f %8.5f %8.5f %8.5f\n",
           i,denorm_fl_out,tr_rec.correct_answers[i][1],
           (denorm_fl_out - tr_rec.correct_answers[i][1]),
           fabs(fl_out - norm_ans));
     } /* end i */
     fclose (outfile);
     cprintf ("\r\nDone. Data saved to file: %s",fl.out_data_file_name);
     cprintf ("\r\n\nHit key to continue: ");
     getch();
exit_proc:
     free_fuzzy_rules (&the_fuzzy_system);
     free_training_pointers(&tr_rec);
     textmode (LASTMODE);
     return;
     } /* end proc */

#endif
```

This is file flgtrain.c. It contains procedures for training GA-Fuzzy system output rules using genetic optimization.

```
#ifndef FLGTRAIN_C
#define FLGTRAIN_C

#include <conio.h>
#include <alloc.h>

#include "ut.h"
#include "trn.h"
#include "flga.h"

void fl_genetic_training (tflg_setup_rec *fl,tga_setup_rec *ga_setup,
        tga_rec *ga);

/* Implementation */

void fl_genetic_training (tflg_setup_rec *fl,tga_setup_rec *ga_setup,
        tga_rec *ga) {
  int gen = 0;
  const int one_fl_output = 1;
  short cancel_flag = 0,new_optimum = 0,within_tolerance = 0;
  int i;
  clrscr();
  textcolor (YELLOW);
  gotoxy (1,5);
   if (fl->fuzzy_system_file_name[0] == '\0') {
        cprintf ("\r\nFuzzy System File Name is blank.\r\n");
        cputs ("Press key to cancel: ");
        getch();
        return;
        }
   if (fl->training_file_name[0] == '\0') {
        cprintf ("\r\nTraining File Name is blank.\r\n");
        cputs ("Press key to cancel: ");
        getch();
        return;
        }
  cprintf ("Fuzzy - Genetic Modeling");
  cprintf ("\r\n\nNumber of Inputs: %d",fl->no_of_fl_inputs);
  cprintf ("\r\n\nNumber of Input Fuzzy Sets: %d",fl->no_of_fl_inp_regions);
  cprintf ("\r\n\nPopulation Size: %d",ga_setup->population_size);
  cprintf ("\r\n\nBit Length: %d      Chromosome Length: %d",
    ga_setup->bit_len,ga_setup->chrom_len);
  cprintf ("\r\n\nFuzzy System File Name: %s",fl->fuzzy_system_file_name);
  cprintf ("\r\n\nTraining File Name: %s",fl->training_file_name);
  cprintf ("\r\n\nPress key to continue (esc to cancel): ");
  if (getch() == ESC_KEY) return;
  init_flga_fuzzy_system (*fl,&g_fuzzy_system);
  /* fl_evaluation_fn needs to use global g_tr_rec: */
```

```
  if (!process_training_file (fl->training_file_name,
          fl->no_of_fl_inputs,one_fl_output,&g_tr_rec)) goto exit_proc;
  g_tr_rec.norm_error_scaling =
    1.0/(g_tr_rec.no_of_training_items * (fl->norm_out_range));
  clrscr();
  gotoxy (1,5);
  cprintf ("Initializing system...");
  normalize_training_set (fl->no_of_fl_inputs,one_fl_output,
    fl->norm_in_range,fl->norm_in_min,fl->norm_out_range,fl->norm_out_min,
    g_tr_rec);
  if (!initialize_genetic_alg (ga_setup,ga,&fl_evaluation_fn))
    goto exit_proc;
  cprintf ("\r\n\nStarting first generation...");
  do {
    gen += 1;
    generation (ga_setup,ga,&fl_evaluation_fn);
    statistics (*ga_setup,ga->populations[ga->g_new],ga,&new_optimum);
    if (new_optimum) {
          ga_vector_to_fuzzy_rules (ga->opt_vector,&g_fuzzy_system);
          write_fuzzy_system (fl->fuzzy_system_file_name,g_fuzzy_system);
          }
    display (gen,*ga_setup,*ga,&cancel_flag,&within_tolerance);
    ga->g_old = (ga->g_old + 1)%2;
    ga->g_new = (ga->g_new + 1)%2;
  } while ((gen <= ga_setup->max_gens) && (!within_tolerance)
            && (!cancel_flag));
exit_proc:
  free_fuzzy_rules (&g_fuzzy_system);
  free_ga_pointers (*ga_setup,ga);
  free_training_pointers(&g_tr_rec);
  cprintf ("\r\nFLGTRAIN:  heapcheck = %d",heapcheck());
  getch();
  return;
  } /* end proc */

#endif
```

This is file flgeval.c. It contains procedures for Evaluation (objective) function for fuzzy logic-genetic algorithm system

```
#ifndef FLGEVAL_C
#define FLGEVAL_C

#include <conio.h>
#include <stdlib.h>
#include <math.h>

#include "trn.h"
#include "flga.h"

float fl_evaluation_fn (ui_vector v);
void ga_vector_to_fuzzy_rules (ui_vector v,fuzzy_system_rec *fz);

/* Implementation */

void ga_vector_to_fuzzy_rules (ui_vector v,fuzzy_system_rec *fz) {
  int i;
  for (i=0;i<fz->no_of_rules;i++)
    fz->rules[i].out_fuzzy_set = v[i];
  } /* end func */

float fl_evaluation_fn (ui_vector v) {
  int i;
  float cum_error = 0.0,fl_out;
  ga_vector_to_fuzzy_rules (v,&g_fuzzy_system);
  /* Compute cumulative absolute error over training set */
  for (i=1;i<=g_tr_rec.no_of_training_items;i++) {
    fl_out = fuzzy_system (g_tr_rec.training_array[i],g_fuzzy_system);
    cum_error += fabs(fl_out - g_tr_rec.norm_answers[i][1]);
    } /* end i */
  cum_error *= g_tr_rec.norm_error_scaling;
  return MAX_GA_OBJ_FN_VALUE - cum_error;
        /* values should be between 1 and 2 */
  } /* end func */

#endif
```