A RADIAL BASIS FUNCION NEURO CONTROLLER FOR PERMENENT MAGNET STEPPER MOTOR

SAIKIRAN GUMMA

Bachelor of Engineering in Electronics and Telecommunication Engineering

J.N.T.University, India

July, 2001

Submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

AUGUST, 2004

ACKNOWLEDGEMENT

I would like to express my sincere indebt ness and gratitude to my thesis advisor Dr. Dan Simon, for the ingenious commitment, encouragement and highly valuable advice he provided me over the entire course of this thesis.

I would also like to thank Dr. Xiong Gao and Dr. Yongjian Fu for their constant support and advice throughout the work.

I wish thank my lab mates at the Embedded Controls and research systems laboratory for their encouragement and intellectual input during the entire course of this thesis without which this work wouldn't have been possible.

Finally, I wish to thank my roommates and my brother Dr. Sasidhar Gumma has always been my role model, and has been a constant source of inspiration to me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Changes in the environment, unmeasurable disturbances, changes in the system parameters, and component failures are some the characteristics of complex dynamic systems that necessitate *intelligent control* techniques. Traditionally plant dynamics are first modeled and verified through experiments, and then controllers are designed. However, such controllers are limited by the accuracy of the identified model and cannot accommodate large variations in parameters. While *adaptive control* is a natural choice to overcome parametric uncertainties, other major issues remain unsolved at this level. Although the region of operation is considerably increased compared to classical control systems as adaptive controllers tune themselves, they don't possess long term memory. Thus, adaptation must be repeated every time the system is confronted with changing operating conditions. To tackle such problems *intelligent control* techniques have been developed, neuro-control being one of those. In this work we use a *specialized learning architecture* with a *radial basis neural network* to develop an inverse dynamic model for a nonlinear permanent magnet stepper motor. This *neuro-controller* is initially trained offline using the *bold driver gradient descent* algorithm and is later used in feedback as a controller. Its performance is then compared with traditional PD controllers tuned for various trajectories and external disturbances. The effect of the number of neurons and initialization of the neural network weights on the performance of the controller is also studied.

CHAPTER I
INTRODUCTION

Real-time control of non-linear plants with unknown dynamics remains a very challenging area of research [22]. Traditionally plant dynamics were first modeled and verified through experiments, and then controllers were designed. Such controllers are limited in performance by the accuracy of the identified model and cannot accommodate large variations in plant parameters, even though they guarantee good tracking performance and robustness to external disturbances.

1.1 Intelligent Control

In the last two decades in the areas of robotics, aircraft control, process control and estimation there have been successful applications of *adaptive control theory* boosted by the availability of powerful microprocessors. “Adaptive Control” is used to denote a class of control techniques where the parameters of the controller are changed (adapted) during control, utilizing the observations on the plant to compensate for parameter changes, other disturbances and unknown factors of the plant. However, most adaptive controllers

are designed for systems that are expressed as linear functions of unknown parameters, and their performance degrades considerably due to disturbances in the regression matrix, and external disturbances. Also it has to be noted that, as the parameters of the plant vary, the adaptive controllers tune themselves but don't possess any kind of memory.

Use of control methodologies in standard practice has opened the doors to a wide spectrum of complex applications. Such complex systems typically characterized by poor models, high dimensionalities and high noise levels can be classified into three categories [Narendra, 1990].

a. Computational complexity
b. Presence of non-nonlinear systems with many degrees of freedom
c. Uncertainties

The third category includes modeling uncertainties, parametric uncertainties, disturbances and noise. The greater the ability to deal with above mentioned difficulties, the more intelligent the control system. *"Qualitatively , a system which includes the ability to sense its environment, process the information to reduce the uncertainty , plan , generate and execute control action constitutes and intelligent control system"*.

It can be inferred that if a human in the control loop can properly control a plant, then that system would be a good candidate for intelligent control [15]. Unlike conventional control techniques, intelligent control has the capability to deal with incomplete plant information, its environment and unexpected or unfamiliar disturbances. Thus,

“Intelligent adaptive control” may be viewed as a class of control techniques that ensures proper operation of a plant, particularly in the presence of parameter changes and unknown disturbances [15]. Over the years, computational procedures such as fuzzy logic, neural networks and genetic algorithms collectively known as “soft computing” techniques, were successfully used either directly or synergistically, for control of various complex systems.

1.2 Literature survey

In recent years learning based control such as neural network and fuzzy logic based controllers has emerged as an alternative to adaptive control. Notably Narendra et al. [21] emphasize the use of dynamic backpropagation for tuning neural networks. Sadegh [29] employs approximate gradients to perform stability analysis, while Polycarpou and Ioannou [24], and Chen and Khallil [6] offer rigorous proofs of performance in terms of tracking error stability and bounded NN weights.

The rationale for using neural control or any other soft computing methods is related to the difficulties faced by control engineers in real-world applications. It is generally difficult to represent a complex process by a mathematical model or by a simple computer model. Even if the model itself is tractable, control of the process using “hard” (non-soft or crisp) control might not provide satisfactory performance. Furthermore, it is commonly known that the performance of industrial processes can be considerably

improved through high-level control actions made by an experienced or skilled operator, which cannot (in most cases) be formulated as crisp control algorithms [11].

Some important properties of neural networks later presented in Chapter II are summarized below in the context of neuro-controllers [4].

- Massive parallelism: Neural networks are highly parallel and can be easily implemented in parallel hardware.
- Inherent nonlinearity: Neural networks have the ability to model any piecewise continuous nonlinear mapping to an arbitrary degree of accuracy by properly selecting the size and parameters of the networks.
- Learning capability: They have the exceptional capability of learning from example data sets.
- Capability of generalization: They exhibit structural capability for generalization, thus will cover many more situations than the examples used to train them. Therefore, they have the ability to deal with difficulties arising from uncertainty, imprecision, and noise in a wide range of problems.
- Guaranteed stability: Theoretical results have been presented to prove that certain neural network architectures (radial basis functions) are guaranteed to be stable for certain non linear control problems.

As can be seen clearly these characteristics are essential in dealing with increasingly complex systems with less precise prior knowledge of a plant and its environment. Because of these capabilities of ANNs mentioned above like learning capability,

generalization, inherent non-linearity, and robustness to unknown dynamics, it has been argued by Werbos [39] that if a control task can be done equally well using a conventional method and a neural network, then there are several advantages of using the latter.

1.3 Thesis Organization

The current research work focuses on the development of an “intelligent adaptive control” method using a class of artificial neural networks for a non-linear stepper motor plant model with unknown disturbance parameters with an upper bound.

Chapter II introduces the fundamental concepts of Artificial Neural Networks, comparison with their Biological counter parts, different architectures of ANN’s and their training procedures and the wide range of possible applications.

Chapter III presents an overview of various kinds of stepper motors, their advantages and various control techniques.

Chapter IV establishes the need for intelligent control of stepper motors and possible application of radial basis function neural networks for control. It gives an overview of training procedures for RBF networks and finally the design of a direct inverse controller.

Chapter V presents the results for various settings of direct inverse controller and its comparison with a typical PD controller. It finally states the various advantages of this type of controller and proposes enhancements to this first step towards *direct inverse control* of PM steppers.

Chapter II
Artificial Neural Networks

Digital computers in use today are mostly based on the principle of using a single powerful processor through which all computations are channeled. This is termed as the *von Neumann architecture*, named after John von Neumann. The power of such a processor can be measured in terms of its speed and complexity. Such computers have been traditionally utilized by writing a precise sequence of steps (a computer program or an algorithm) to be executed by the computer. This is the *algorithmic approach* [5].

On the other hand researchers in artificial intelligence (AI) follow the algorithmic approach and try to capture the knowledge of an expert in some specific domain as a set of rules to create so called *expert systems*. This is based on the hypothesis that the expert's thought process can be modeled by using a set of symbols and a set of logical rules which manipulate such symbols. This is the *symbolic approach* [5].

A deep scientific concern has been that it still requires someone to understand the process (the expert) and someone to program the computer. The algorithmic and symbolic approaches can be very useful for certain problems where it is possible to find a precise sequence of mathematical operations or a precise sequence of rules. However, such approaches have the weaknesses in the sense that "*learning*" is difficult and they fail to tackle problems with increasing dimensionality (like in regression problems) [7]. If we define computational "learning" as the construction or modification of some computational representation or model [5], it is difficult to simulate "learning" using the algorithmic and symbolic approaches.

Artificial Neural Networks (ANN), also referred to as connectionist models, parallel distributed processors and self-organizing systems, provide an alternative approach to be applied to problems where the algorithmic and symbolic approaches are not well suited. ANNs are computational models of the human brain. The brain is composed of approximately 10^{11} nerve cells termed neurons. Although each of these elements is relatively simple in design it is believed the brain's computational power is derived from the interconnection, hierarchical organization, firing characteristics, and the sheer number of these elements. The actions and interconnections of biological neurons have given the spirit and scope for the fascinating field of artificial neural networks [16].

2.1 Biological Neural Networks

The human information processing system consists of the 'central nervous system' (CNS) and the 'peripheral nervous system' (PNS). The CNS is composed of the brain and the spinal cord. The PNS is composed of the nervous system outside the brain and spinal cord [5]. The CNS of the human body consists of three stages: receptors, a neural network, and effectors. The nervous system can be seen as a vast electrical switching network to which the inputs are provided by sensory receptors. Such receptors act as transducers and generate signals from within the body or from sense organs that observe the external environment. The information is then conveyed by the PNS to the CNS, where it is then analyzed and processed. If necessary, the CNS sends signals to the effectors and the related motor organs that will execute the desired actions. From the above description we can see that the human nervous systems can be described as a closed-loop control system as shown in *Fig. 2.1*, with feedback from within and from outside the body to regulate some bodily functions [42].

The basic building block of the CNS is the biological neuron, the cell that communicates information to and from the various parts of the body [16]. The human brain contains approximately 10^{11} neurons and each of these neurons is connected to around 10^{3} to 10^{4} other neurons, and therefore the human brain is estimated to have 10^{14} to 10^{15} connections.

The neuron consists of a cell body called soma, and several spine-like extensions off the cell body called dendrites. A single nerve fiber called the axon branches out from the

soma and connects many other neurons. The junctions by which these connections between neurons occur are called synapses which are either on the cell body or on the dendrites, as shown in *Fig. 2.2*. Nerve impulses originate at the dendrite tree or the cell body, propagate through the axon and communicate with the neighboring neurons through the synaptic junctions. The inter-neuronal signal at the synapse is usually chemical diffusion but sometimes electrical impulses

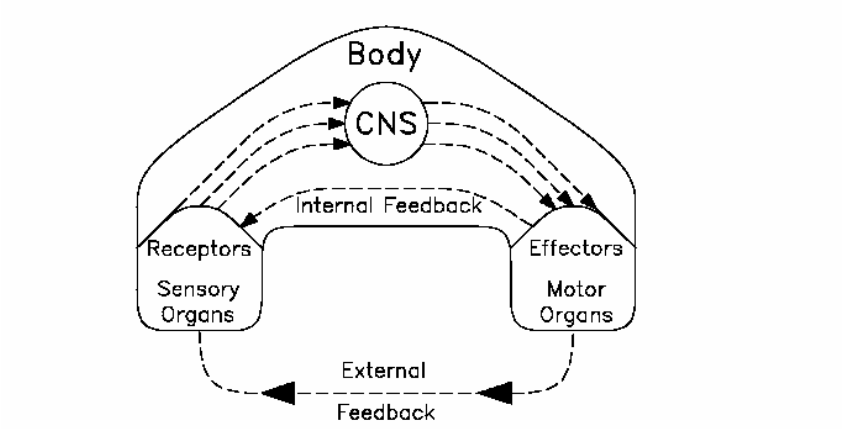
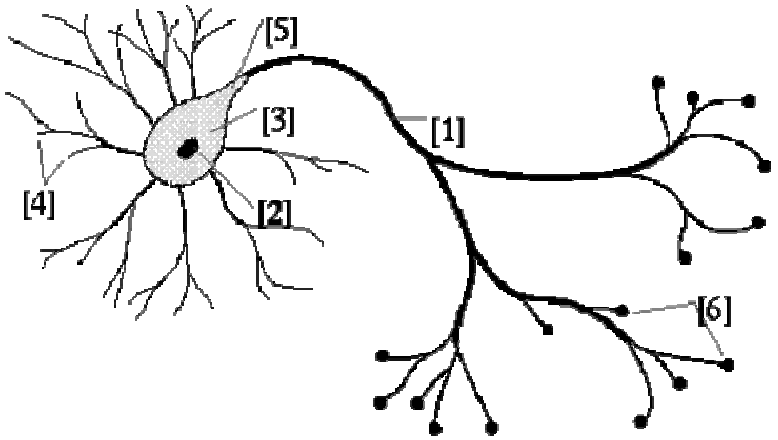


Fig. 2.1 Control system view of human body

The incoming impulse signal from each synapse to the neuron is either excitatory or inhibitory, which means helping or hindering firing. The condition of causing firing is that the excitatory signal should exceed the inhibitory signal by a certain amount in a short period of time, called the period of latent summation [20]. With a weight assigned to each incoming impulse signal, the excitatory signal has positive weight and the inhibitory signal has negative weight. This way, we can say, "A neuron fires only if the total weight of the synapses that receive impulses in the period of latent summation exceeds the threshold [1]."



1.Axon 2. Nucleus 3.Soma (Body) 4. Dendrite 5. Axon Hillock 6. Terminals (Synapses)

Fig. 2.2 Biological neuron

In essence, all that a neuron does is to sum up the values of various inputs applying a weighting factor to each and give an output when this sum of weighted inputs exceeds a certain threshold.

2.2 Artificial Neural Networks

An excellent ANN repository on a Usenet newsgroup [31] says,

"There is no universally accepted definition of an ANN. But perhaps most people in the field would agree that an ANN is a network of many simple processors ("units"), each possibly having a small amount of local memory."

According to the DARPA Neural Network Study [8] ,

"A neural network is a system composed of many simple processing elements operating in parallel whose function is determined by network structure, connection strengths, and the processing performed at computing elements or nodes."

These “units” are connected by communication channels (“connections”) which usually carry encoded numeric (as opposed to symbolic) data. The units operate only on their local data and on the inputs they receive via the connections, as shown in *Fig. 2.3*. The restriction to local operations is often relaxed during training.

Of the wide variety of ANN models, some resemble biological neural networks and some do not. But historically much of the inspiration for the field of ANNs came from the desire to produce artificial systems capable of sophisticated, perhaps “intelligent”, computations similar to those that the human brain routinely performs. The fundamental element, an artificial neuron, is a model based on known behavior of biological neurons that exhibit most of the characteristics of human brains. It is generally believed that knowledge about *real* biological neural networks can help by providing insights about how to improve the *artificial* neural network models and clarifying their limitations and weaknesses [20]. A comparison of artificial neural networks with biological neural networks is presented in *Table 2.1*.

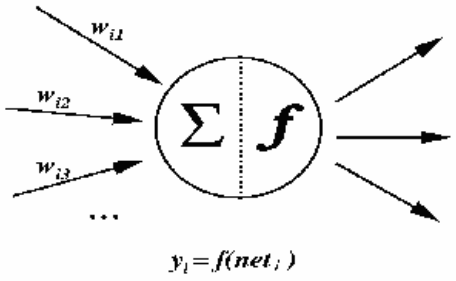


Fig. 2.3 An ANN ‘processing unit’/ ‘artificial neuron’

Element	Brain	ANN
1.Organization	Network of neurons.	Network of processing elements.
2.Component	Dendrites, axons, summer, threshold.	Inputs outputs, weights, summation and threshold function.
3.Processing	Analog	Digital.
4.Architecture	10-100 billion neurons.	1-1,000,000 processing elements.
5.Hardware	Neurons.	Switching devices.
6.Switching speed	1 millisecond.	1-nano second to 1-millisecond.
7.Technology	Biological.	Silicon, optical, molecular.
8.Speed	Slow in processing information.	Fast.
9.Control machines	No central control.	One central unit.

Table 2.1 Comparison between ANN and BNN

Most ANNs have some sort of "training" rule whereby the weights of connections are adjusted on the basis of data. ANNs "learn" from examples, and if trained carefully, may exhibit some capability for generalization beyond the training data. That is they tend to produce approximately correct results for new cases that were not used for training.

ANNs normally have great potential for parallelism, since the computations of the components are largely independent of each other.

It should be noted that massive parallelism and high connectivity cannot be the defining characteristics of ANNs, as such requirements rule out various simple models, such as simple linear regression, which are usefully regarded as special cases of ANNs [20].

2.3 Types of Artificial Neural Networks

The artificial neuron, the most fundamental computational unit, is modeled on the basis of a biological neuron. Such a neuron basically consists of a number of inputs each associated with a memory (weight). However, as the name indicates, the true computing power of biological neural networks lies in the fact that the neurons are highly interconnected units, giving exceptional parallel processing abilities. Thus, connecting multiple neurons is a key aspect of ANN design. In view of this importance, a brief overview of neural network topologies is presented in the next section.

2.3.1 Topology Characteristics

Organizing artificial neurons into fields (also called slabs or layers) and linking them with weighted interconnections forms ANN topologies. The main characteristics of these topologies include connection types, connection schemes and field configurations.

a) Connection types: There are two primary connection types, excitatory and inhibitory. Excitatory connections increase a neuron's activation and are typically represented by a positive signal. On the other hand inhibitory connections decrease a neuron's activation and are represented by a negative signal [16].

b) Interconnection Schemes: The three primary neuron interconnection schemes are intra-field, inter field and recurrent connections. Intra-field connections or intra-layer connections or lateral connections are the connections between neurons in the same layer. Inter-field or inter-layer connections are the connections between different layers. Recurrent connections are connections that loop and connect back to the same neuron.

As can be understood from above classifications, the inter-field connection signals, propagate in one of two ways:

i. Feed forward

ii. Feed back.

Feed forward signals only allow information to flow among neurons in one direction. In case of the feedback signals, information flow is in either direction and/or recursive.

c) Field Configurations: Field configurations combine fields of neurons, information flow and connection schemes into a coherent architecture. Field configurations include lateral feedback, field feed forward and field feedback. A field that receives input signals from the environment is called an input field and a field that emits signals to the environment

is called an output field (output layer). Any field that lies in between the input and output fields are called hidden layers and have no direct contact with the environment (i.e. input and output neurons).

2.3.2 The McCulloch-Pitts Model of Neuron

An early artificial neuron model introduced by Warren McCulloch and Walter Pitts in 1943 is also known as the *threshold logic gate* (TLG). McCulloch-Pitts view of neuron model depicted in *Fig. 2.4* has a set of inputs [$\boldsymbol{I_1, I_2, I_3 \ldots I_N}$] and one output $\boldsymbol{y}$. This early version of neuron, simply classifies the input vector into two different classes, that is to say, the output $\boldsymbol{y}$ is binary [7].

Such a function can be mathematically described as follows:

$$sum = \sum_{i=1}^{N} I_i W_i \qquad 2.1$$

$$y = f(sum) \qquad 2.2$$

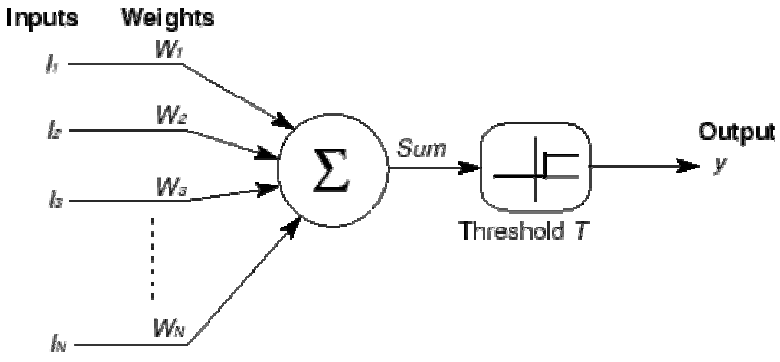


Fig. 2.4 McCulloch-Pitts Model of Neuron

[W_1, W_2, W_3,…,W_N] are weight values normalized in the range of either [0,1] or [-1,1] and associated with each input line, ***sum*** is the weighted sum, and ***'T'*** is a threshold constant. The function ***f(.)*** is a linear step function at threshold ***'T'*** as shown in *Fig. 2.5*.

$$f(k) = \begin{cases} 0 & \forall\ k \le T \\ 1 & \forall\ k > T \end{cases} \qquad 2.3$$

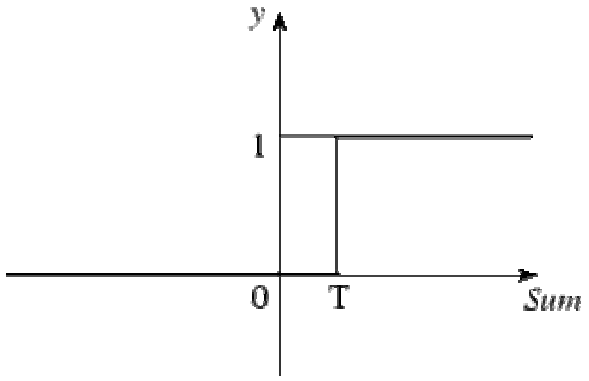


Fig. 2.5 Linear step function (Threshold = T)

The McCulloch-Pitts model of a neuron with a precise mathematical definition has proven to have substantial computing potential. However, this model is so simplistic that it only generates a binary output, and also the weight and threshold values are fixed.

2.3.3 The Perceptron

In early 1960's Rosenblatt studied single layered networks for classification problems [7]. His model of the neuron, the *perceptron*, was a merge between two concepts proposed in the 1940s, the McCulloch-Pitts model of an artificial neuron and the Hebbian learning rule of adjusting weights [2]. Apart from the variable weight values of the TLG, the perceptron model added an extra input θ, which represents *bias*. In the first stage of

processing, a linear combination of inputs is calculated where each input is associated with its weight value. The summation function often takes an extra input value θ, with a weight value of 1 to represent the threshold or *bias* of a neuron.

$$sum = \sum_{i=1}^{N} A_i W_i + \theta \qquad 2.4$$

$$= \sum_{i=1}^{N} A_i W_i + W_0 \qquad [W_0 = \theta = 1, \text{ is the bias}]$$

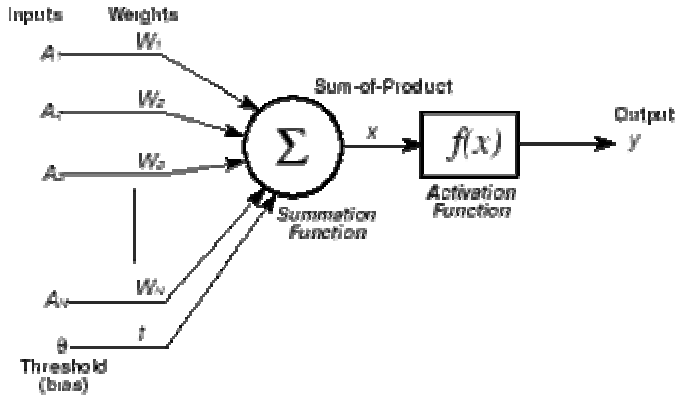


Fig. 2.6 Perceptron

Further, the sum-of-product value is passed into the second stage to perform the activation function which generates the output from the neuron. The activation function 'squashes' the amplitude the output in the range of [0, 1] or [-1, 1]. The behavior of the activation function will describe the characteristics of an artificial neuron model. As most real world signals are continuous in nature, activation functions that are continuous with bounded range were also introduced.

One such convenient function is the *logistic sigmoid* function as shown in *Fig. 2.7*. Here as the input ***x*** tends to a large positive value, the output value ***y*** approaches to 1. This function can be expressed mathematically as below:

$$y = \frac{1}{1 + \exp(-x)} \qquad 2.5$$

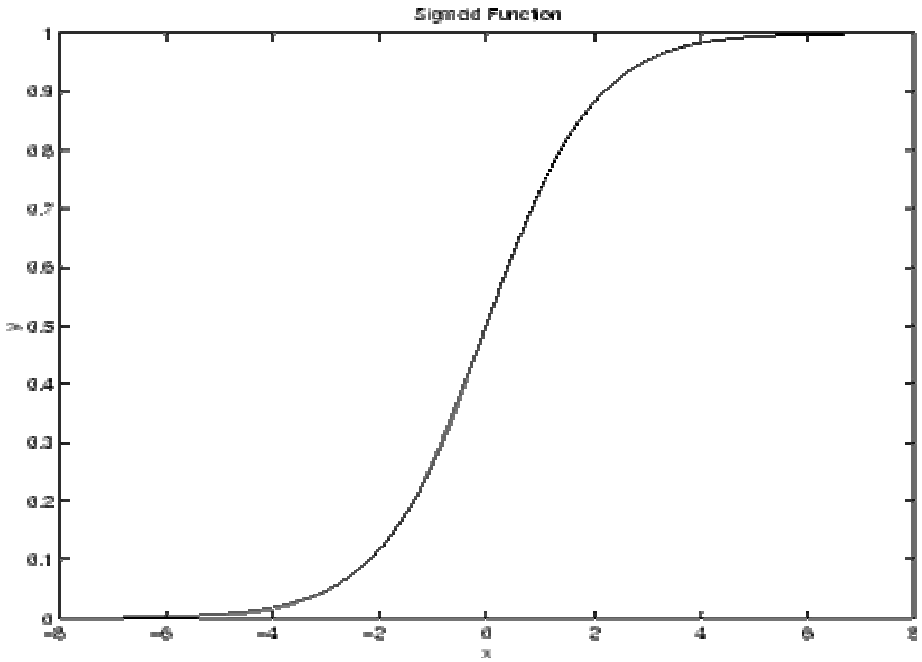


Fig. 2.7 Sigmoid activation function

2.3.4 Single Layered Perceptron

Individual neurons described previously can perform a substantial level of computation. However, as mentioned earlier, the true computing power of neural networks comes by connecting multiple neurons. A common topology of connecting neurons into a network is by forming layers of neurons. The simplest form of layered network is shown in *Fig. 2.8* [20], which is a feed forward topology. Only two layers of neurons are involved, viz.,

the *input layer* (shaded nodes on the left) and *output layer*. Here the input layer neurons only pass and distribute the inputs and perform no computation.

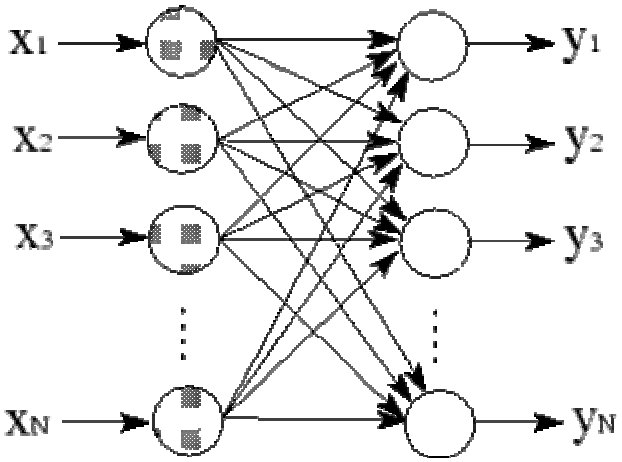


Fig. 2.8 Single Layered Network

This perceptron network has only one group of *weights* and also has no *hidden layers* hence is referred to as a single layer network. This net can be used with both continuous valued and binary inputs, in which inputs are given at one layer and the output is obtained at another layer. Note that all processing is done at the output layer, and the topology doesn't involve any type of recurrent (intra layer) connections. Each of the inputs [$x_1, x_2, x_3,\ldots,x_N$] is connected to every artificial neuronin the output layer, through corresponding connection weight. Since all output values [$Y_1, Y_2, Y_3,\ldots,Y_N$] are calculated from the same set of input values, each output is dependent on the connection weights.

$$Y_j = f\left(sum_j\right)$$
$$sum_j = \sum_{j=1}^{N} W_{ji}X_i + W_{j0} \qquad 2.6$$

The *learning* or *training* process of such a network typically involves adjusting the weight matrix so as to mimic the response for a known input-output mapping. Although *Fig. 2.8* shows a fully connected form of the network, the true neural network may sometimes not have all possible connections, meaning a weight value of zero which represents 'no connection'.

2.3.5 Multi Layered Perceptron (MLP)

In problems with complicated input-output relationships often a more complex structure of neural network is required, to achieve a higher level of computation. The multi-layer perceptron is multi-layered feed forward neural network architecture with one *input layer*, one *output layer* and a number of hidden layers, as shown in *Fig. 2.9*.

$$sum_j^M = \overline{f}\left(\sum_{i=1}^{N} \left(sum_i^{M-1} \right) + sum_{j0}^{M-1} \right) \qquad \forall\ j \in (1, M) \qquad 2.7$$

$$Xi = sum_i^0 \qquad Yi = sum_i^M \qquad \forall\ i \in (1, N)$$

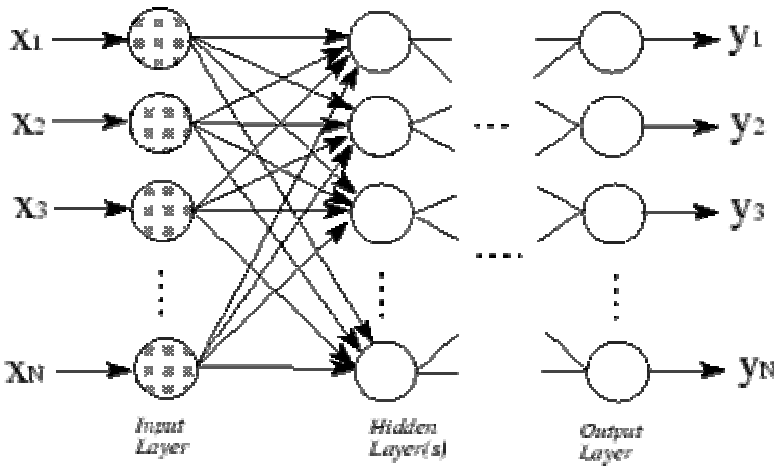


Fig 2.9 Multi Layered Network

A *multilayer neural network* basically distinguishes itself from the single-layer network by having one or more *hidden layers*. In such a network, inputs given at the input layer are processed at the hidden layers. There may be one or more than one hidden layer, where the number of hidden layers is generally a tradeoff between the complexity of the problem and computational effort. From a designer's point of view the multilayer networks can also be trained similar to a single layer network; however, the weight adjustments have to be propagated back through the layers of the neural network.

2.3.6 Radial Basis Function (RBF)

The Radial Basis Function (RBF), a relatively new type of neural network architecture introduced in the 1980's, belongs to the class of feed forward topologies. It stands out from the traditional MLP class of architectures by its *hidden unit* (middle layer) activation where the activation of the hidden unit is determined by the *Euclidian distance* between the input vector and the prototype vector. RBF networks have only *one* hidden layer, in which each neuron (*radial unit*), each modeling a responsive surface (generally a gaussian), is defined by its center point (in *N* dimensional space) and a radius that makes a prototype vector.

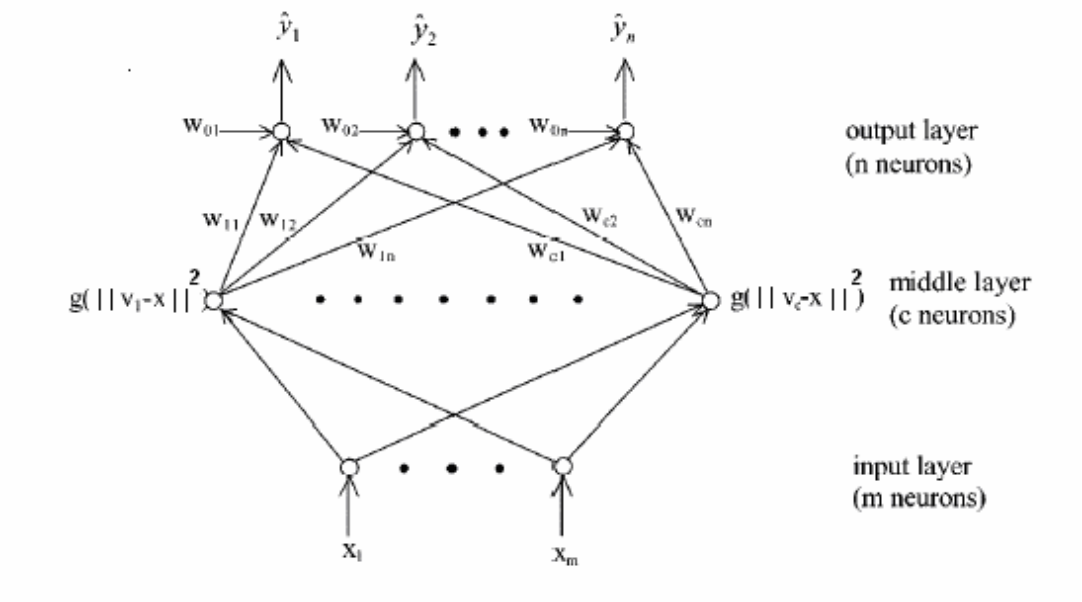


Fig 2.10 Radial Basis Function NN

Consider the network in *Fig. 2.10* with ***m*** input neurons, ***c*** hidden neurons and ***n*** output layer neurons. Each of the ***c*** neurons in the hidden layer applies an activation function g(.) which is a function of the *Euclidean distance* between the input and an ***m***-dimensional prototype vector [35] . Each hidden neuron with its own prototype vector as a parameter gives an output that is then weighted and passed to the output layer. The outputs of the network consist of sums of the weighted hidden layer neurons.

$$\hat{y}_i = \sum_{i=1}^{n} W_{ji} C_j + W_{j0} \qquad 2.8$$
$$where \quad , \quad C_j = g(\| v_j - x \|)$$

Since the activation functions are nonlinear, it is not actually necessary to have more than one hidden layer to model any shape of function: sufficient radial units will always be enough to model any function [7]. It turns out to be quite sufficient to use a linear

combination of these outputs (i.e., a weighted sum of the Gaussians) to model any nonlinear function. The RBF ANNs *training* is generally a two stage process.

(a) assignment of prototype vectors and parameters (getting radial centers and widths)

(b) adjustment of output layer weights.

Different training schemes for RBF networks will be discussed in detail in Chapter IV.

2.3.7 Other forms of ANNs

Since the advent of neural networks during middle of this last century, various models and topologies have been proposed, each with its own significance. A few of them are Hopfield networks, Hamming nets, Boltzman machines, Carpenter/Grossberg classifiers, Kohonen's Self-Organizing feature maps, etc. However, it is not in the scope of this thesis to introduce or discuss all of them in detail.

2.4 Learning Methods (Training Algorithms)

Learning is a process in which samples containing a pattern are presented to the network several times before the information pattern is captured by the weights (memory) of the network. An interesting feature of learning is that the network from the training samples slowly acquires the pattern information and the training samples themselves are never stored in the network. The learning methods can be broadly classified into three major groups.

i. Supervised learning.

ii. Reinforcement learning.

iii. Unsupervised learning.

In *supervised learning* it is assumed that the correct target output values (***y***) are known for each pattern. The learning (training) process involves some kind of feedback for adjustment of weights of the ANN optimally so as to generate the desired output pattern.

In the extreme case there is only a single bit of feedback information indicating whether the output is right or wrong. Learning based on this kind of critic information is called *reinforcement learning* and the feedback information is called the reinforcement signal. Reinforcement learning is a form of supervised learning because the network still receives some feedback from its environment. But the feedback is only evaluative (critic) rather than instructive.

In *unsupervised learning* there is no teacher to provide any feedback information. There is no feedback from the environment to say what the outputs should be or whether they are correct. The network must discover for itself patterns features, regularities, correlation's or categories in the input data and code for them in the output while discovering these features the network undergoes changes in its parameters; this process is called self-organizing.

2.4.1 Basic Learning laws

Hebb's law: This first and the best known unsupervised learning rule was introduced by Donald Hebb in 1949. This basic rule is: *If a neuron receives an input from another neuron, and if both are highly active (mathematically have the same sign), the weight between the neurons should be strengthened*. This law represents unsupervised learning. Consider two neurons in consecutive layers of a NN. The connection strength between i^{th} neuron $\boldsymbol{a_i}$ (signal A_i), and j^{th} neuron $\boldsymbol{s_j}$ (signal S_j) in the next layer is given by W_{ij}.

Now, Hebb's lay states that the change in weight vector is given by,

$$\Delta W_{ij} = \eta . S_j . A_i$$

Where, η is the *learning rate*. Here, the weights are strengthened if units connected are activated (with same output sign). Weights are normalized to prevent infinite increase.

Delta learning: This learning law is valid only for a differentiable output function as it depends on the derivative of the output function wherein the change in weights is proportional to the mean squared error. This is also viewed as a continuous perceptron learning law where the initial weights are taken randomly and are modified by this supervised learning rule.

Change in the weight matrix associated jth neuron be given by $\boldsymbol{W}$

Let, $S_j = f(\boldsymbol{W}.\mathbf{A})$ be the output of the j^{th} neuron, and T_j be the desired target value.

Now according to the delta rule,

$W_{new} = W_{old} + \Delta \boldsymbol{W}$, where $\Delta \boldsymbol{W} = \eta . \ \partial(\boldsymbol{T}-\boldsymbol{S})/ \partial \boldsymbol{W}$

η is called the learning rate.

There have been number of training schemes proposed for different types of neural networks that are direct implementations of the learning laws introduced above and their derivatives. *Back propagation algorithm* is one such training scheme that is directly based on delta learning. A detailed insight into the learning schemes and their use to train RBF neural networks is given in Chapter IV.

2.4.2 Salient features of learning laws

1. The learning law should lead to convergence of weights.
2. The learning or training time for capturing the pattern information from samples should be as small as possible.
3. An online learning is preferable to an off-line learning. That is, the weights should be adjusted on presentation of each sample containing the pattern information.
4. Learning should use only local information as far as possible. That is, the change in the weight on a connecting link between two units should depend on the state of these two units only. In such a case, it is possible to implement the learning law in parallel for all the weights, thus speeding up the learning process.
5. Learning should be able to capture non-linear mappings between input-output pattern pairs as well as between adjacent patterns in a temporal sequence of patterns.

6. Learning should be able to capture as many patterns as possible into the network. That is, the pattern information storage capacity should be as large as possible for a given network.

2.5 Comparison between RBFs and MLPs

RBF networks have a number of advantages over MLPs. First, as previously stated, they can model any nonlinear function using a single hidden layer, thus reducing some design decisions about numbers of layers. Secondly, the simple linear transformation in the output layer can be optimized fully using traditional linear modeling techniques. Hence, RBF networks can be trained extremely quickly, orders of magnitude faster than MLPs.

On the other hand, before linear optimization can be applied to the output layer of an RBF network, the number of *radial units* (hidden neurons) must be decided, and then their centers and standard deviations must be determined. Although faster than MLP training, the algorithms to do this are prone to discover sub-optimal combinations [Bishop]. RBFs more eccentric response surface requires a *lot* more units to adequately model most functions. Consequently, an RBF solution will tend to be slower to execute and more space consuming than the corresponding MLP [7].

Also, RBFs are not inclined to extrapolate beyond known data: the response drops off rapidly towards zero if data points far from the training data are used. In contrast, an MLP is more certain in its response when far-flung data is input. Whether this is an

advantage or disadvantage depends largely on the application, but on the whole the MLPs uncritical extrapolation far from training data is usually dangerous and unjustified [16].

2.6 Applications

The motivation of studies in neural networks lies in the flexibility and power of information processing that conventional computing machines do not have. Although most computers can process faster and more precisely than human brains, people have ability to obtain experience then make more sensible decisions [42]. Similar to the way that the human brain generalizes, the neural network system can "learn by examples and experience" and perform a variety of nonlinear functions that are difficult to describe mathematically [20].

Attractive features of ANNs are their robustness and fault tolerance, flexibility, ability to deal with a variety of data situations and collective computation. Thus, ANNs are used by a wide variety of people as mentioned below.

- Computer scientists want to find out about the properties of non-symbolic information processing with neural nets and about learning systems in general.
- Statisticians use neural nets as flexible, nonlinear regression and classification models.
- Engineers of many kinds exploit the capabilities of neural networks in many areas, such as signal processing and automatic control.

- Cognitive scientists view neural networks as a possible apparatus to describe models of thinking and consciousness (high-level brain function).
- Neuro-physiologists use neural networks to describe and explore medium-level brain function (e.g. memory, sensory system).
- Physicists use neural networks to model phenomena in statistical mechanics.
- Biologists use neural networks to interpret nucleotide sequences.
- Philosophers and Economists are interested in ANNs for modeling and prediction in systems that don't have well defined mathematical models.

Chapter III
STEPPER MOTORS

3.1 Introduction

The essential property of a *stepper motor* is to translate switching excitation changes into precisely defined increments of rotor position [23]. Stepping motors can be viewed as electric motors without commutators [13]. They are used in a wide variety of applications, ranging from simple applications like machine tools, typewriters, and watches, to high end space applications such as positioning mechanisms for antennas, mirrors, telescopes and complete payloads. Their use has skyrocketed with the popularity of embedded systems in printers, disk drives, toys, windshield wipers, vibrating pagers, robotic arms, and video cameras [34].

Due to various disadvantages of DC motors (typically involving a potentiometer to provide feedback) positioning systems are increasingly being implemented by using induction motors and stepper motors [25]. However, when making a choice between

steppers and servos, a number of issues that are application specific must be considered. From a control engineer's prospective, the development of open loop or closed loop control involves modeling, and simulation of any system (a stepper motor in this case) requires a thorough understanding of the system dynamics. Thus, this chapter attempts to provide a brief overview of physics and electromechanical behavior of a stepper motor and its principles of operation.

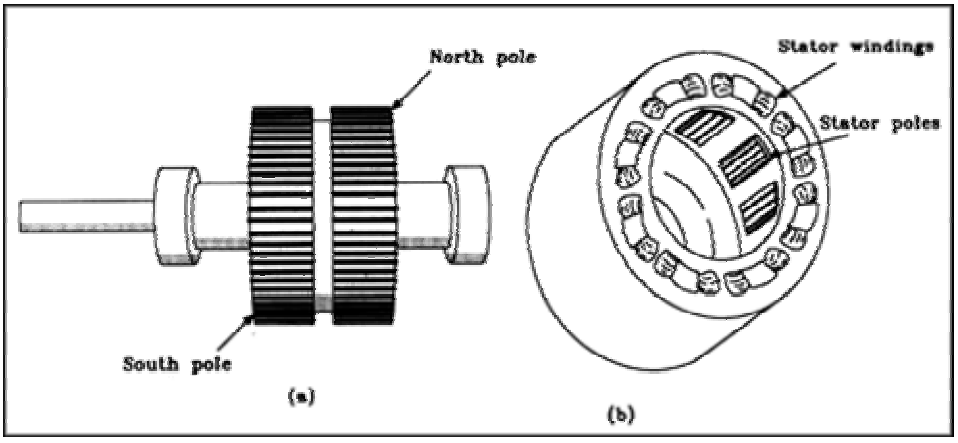


Fig. 3.1 Components of a PM stepper motor: (a) Rotor; (b) stator

3.2 Types of Stepper Motors

3.2.1 Permanent Magnet (PM) Stepper Motor

A PM stepper motor operates on the reaction between a permanent-magnet rotor and an electromagnetic field. A basic two-pole PM stepper motor is shown in *Fig 3.1*. The *rotor*, the freely rotating cylindrical part of the motor, has a permanent magnet mounted with one pole at each end as shown in *Fig. 3.1(a).* The *stator*, the stationary part of a motor, is illustrated in *Fig. 3.1(b)* has current carrying conductors that are wound around its teeth. The wire that is wound around the teeth is called a *winding, coil,* or *phase*. The current

flowing in the phase induces a magnetic field in the stator poles, given by Ampere's Law and the right hand rule (see *Fig. 3.2*). These winding currents produce magnetic fields which add together vectorially to produce an overall stator flux.

The stator flux interacts with the permanent magnet rotor flux to produce a torque in the rotor that is free to move about its axis. When the stator and rotor fluxes are aligned with each other, the motor is in a stable equilibrium and zero torque is produced. When the stator and rotor fluxes are opposite each other, the rotor is in an unstable equilibrium position. Any other relative orientation of the stator and rotor fluxes produces torque in the rotor [33]. This forms the basic principle for operation of the stepper motor. Generally teeth on the rotor surface and the stator pole faces are offset so that there will be only a limited number of rotor teeth aligning themselves with an energized stator pole [23].

As is obvious from intuition, the number of teeth on the rotor and number of stator phases determine the step angle. The greater the number of teeth, the smaller will be the step angle. For a PM stepper motor *holding torque* is defined as the amount of torque required for moving the rotor one full step, with the stator energized [13]. An important characteristic of the PM stepper motor is that it can maintain the holding torque indefinitely when the rotor is stopped. That is even if no power is applied to the windings a small amount of magnetic force is developed between the permanent magnet and the stator. This magnetic force is called a *residual* or *detent torque*. The detent torque can be noticed by turning a stepper motor by hand and is generally about one-tenth of the

holding torque. The PM stepper motor has to overcome the detent torque to line up with the stator field when a steady DC signal is applied to the stator winding.

3.2.2 Principle of Operation of a PM Stepper Motor

To give an understanding of the working principle of the PM stepper motor, a description of one full revolution of a simple *two-phase, two-pole* PM motor shown in *Fig. 3.2* [34], in *half-step mode*, is described below. For other modes of excitation possible refer under Section 3.3 of this chapter. As per Ampere's Law and the right hand rule, the current flowing in the direction shown in *Fig. 3.2(a)* in the stator phase induces a magnetic field with the north pole of the field pointing upwards.

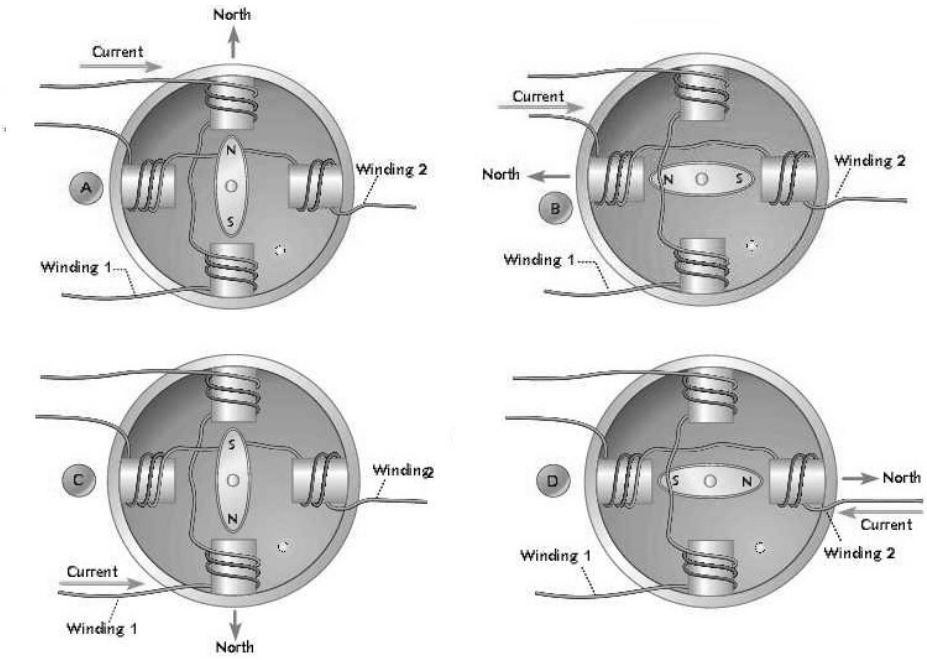


Fig. 3.2 One full revolution of two-phase two-pole PMS motor

With a current through winding 1 in the direction shown in *Fig. 3.2(a)*, and no current through winding 2, the rotor will align itself in the direction shown, with its north pole pointing in the north direction of the stator's magnetic field. Suppose current from winding 1 is removed and applied to winding 2 in the direction shown in *Fig. 3.2(b)*. The stator's magnetic field will point to the left, and the rotor will rotate to the equilibrium position where it is aligned with the stator's magnetic field, yielding a zero sum of rotor and stator flux.

Similarly, removing current from winding 2 and applying current to winding 1 in opposite direction to that of *Fig. 3.2(a)*, as shown in *Fig. 3.2(c)* will result in the stator field pointing down. Exciting only winding 2 in the direction shown in *Fig. 3.2(d)* will result in the stator field pointing to the right. These excitations simultaneously force the rotor to positions where the rotor aligns itself with the stator flux. As a final step, removing current from winding 2 and apply current to winding 1 in the direction shown in *Fig. 3.2(a)*, returns the rotor to its original position.

At this point one full cycle of *electrical excitation* of the motor windings is said to be completed, while the rotor has rotated one complete revolution. In this case, the *electrical frequency* (f_e) of the motor is equal to the *mechanical frequency* (f_m) of the motor. Other kinds of PM steppers such as unipolar, bifilar with different constructions of pole winding structure and other variations of this basic configuration are also available.

3.2.3 Variable Reluctance Stepper Motors

The variable-reluctance (VR) stepper motor at its core basically differs from the PM stepper in that it has no permanent-magnet rotor and thus no residual torque to hold the rotor at one position when turned off. The stator of a variable-reluctance stepper motor has a magnetic core constructed with a stack of steel laminations. The rotor is made of unmagnetized soft steel with teeth and slots, or any other such magnetically permeable substance, unlike PM stepper motors [23]. When the stator coils are energized, the rotor teeth will align with the energized stator poles. This type of motor operates on the principle of minimizing the reluctance along the path of the applied magnetic field. By alternating the windings that are energized in the stator, the stator field changes, and the rotor moves to a new position [13].

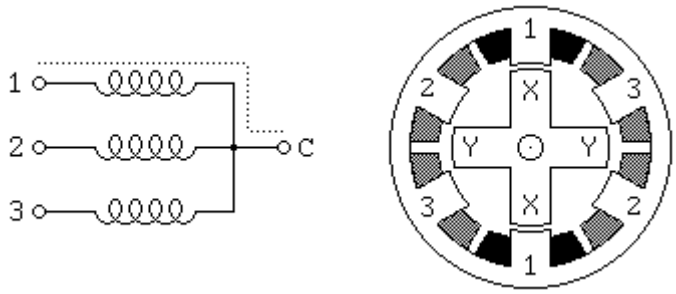


Fig. 3.3 Cross section of a VR stepper motor

As a example to understand the working principle consider *Fig. 3.3* that shows a basic variable-reluctance stepper motor that has six stator teeth. There are fewer rotor teeth than those on the stator, which ensures that only one set of stator and rotor teeth will align at any given instant. This often proves to be limitation in this kind of motor [23].

As long as a single phase (say only phase 1) is energized, the rotor will be held stationary (X-X aligned along vertical axis). When phase 1 is switched off and phase 2 is energized, the rotor will turn 30° until the remaining two poles of the rotor (Y-Y) are aligned under the north and south poles established by phase 2. Similarly another change in excitation causes the rotor to move another 30° and X-X will then be aligned under the north and south poles created by phase 3. By repeating this pattern, the motor can be rotated in a clockwise direction. Reversing the direction of current in each phase can change the direction of the motor.

The VR stepper motors mentioned up to this point are all single-stack motors. That is, all the phases are arranged in a single stack, or plane. The disadvantage of this design for a stepper motor is that the steps are generally quite large (above 15°) [23]. A variation to this scheme is the *multistack stepper motor* that can produce smaller step sizes because the motor is divided along its axial length into magnetically isolated sections, or stacks. A separate winding, or phase, excites each of these sections. In this type of motor, each stack corresponds to a phase, and the stator and rotor have the same tooth pitch.

3.3 Comparison between VR and PM Stepper Motors

In general Hybrid/PM steppers have great step resolution (typically 1.8^{o}) which is advantageous when high angular position resolution is needed. On the other hand variable reluctance steppers are useful in applications where a load is to be moved a considerable distance, due to their large step size (typically 15^{o}), with fewer number of

excitations. PM motors produce a small amount of detent torque that help in preserving the position even after current excitations in the windings are removed. This also proves to be a disadvantage as they have a large mechanical inertia compared to VR motors. In summary the choice of the type of step motor is influenced by the application and it is not possible to categorically state which type is 'better' [23].

With its wide range of applications and simple understandable physics PM stepper motors are of great interest to control engineers. In this work we focus on investigating a new control methodology for PM steppers. The next section presents a brief overview of the modeling of the nonlinear dynamics of the motor.

3.4 Modes of Excitation

To enable rotation of the rotor the magnetic field generated by the stator windings have to interact and drive the rotor flux, which is achieved by switching the direction of current flow through each winding. Basic stepper motor 'step modes' include full-step, half-step, and micro-step. The type of step mode output of any motor is dependent on the design of the driver circuit.

Full-step: A full step mode is achieved by energizing both phases (as in case of a two phase motor) of the motor, while reversing the current alternately. In this method windings are energized producing a 'north-south' pole pair in a cyclic fashion. The flux vectors are out of phase which attracts the rotor's respective poles and holds the rotor in position at each step. The length of each step depends on the number of rotor teeth (Nr).

This way the torque produced by the motor is increased but the power supply to the motor is also increased.

Half-step: Exciting in a half-step mode essentially doubles the resolution (steps per rotation) of the stepper. Even though the switching sequence is similar to that of the full-step mode, instead of just reversing the flow of current through a phase, one phase is completely switched off in between. Thus there is another stage in the electrical switching cycle where in only one winding is excited while the other is completely switched off. This method allows the rotor to follow and take up even more positions.

Micro-step: The full step length of a stepping motor can be divided in to smaller increments of rotor motion, known as “micro-step” by partially exciting several phase windings. Micro stepping is a relatively new stepper motor technology that controls the current in the motor winding. Micro stepping is typically used in applications that require accurate positioning and a fine resolution over a wide range of speeds. The major disadvantage of the micro-step drive is the cost of implementation due to the need for partial excitation of the motor windings at different current levels.

3.5 Modeling of a Permanent Magnet Stepper (PMS) Motor

In order to investigate the dynamics of mechanisms driven by stepper motors a model had to be created. A number of references are available on the generation of a model [41,25]. With a minimum background of basic laws of electromagnetism and motor physics, this

section provides a brief derivation of a nonlinear model of the 2-phase PM stepper motor shown in Figure 3.3.

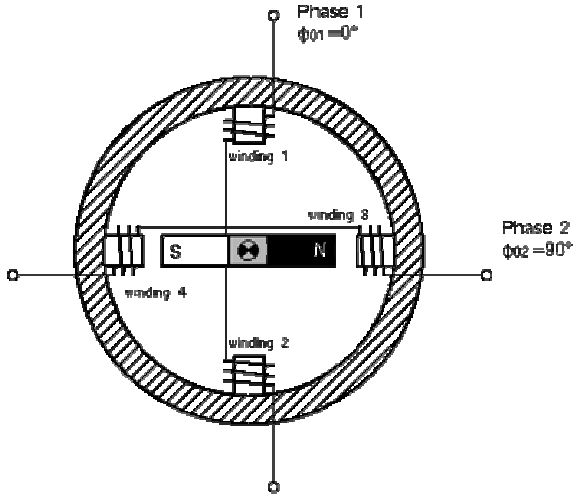


Fig. 3.4 Single pole, 2 phase - PMS

As explained earlier, when the windings of a phase are energized, a magnetic dipole is generated on the stator side. If for example phase 2 is active (phase 1 is switched off), winding 3 produces an electrical north pole and winding 4 a south pole. *Fig. 3.3* shows the rotor in a stable position with phase 2 only powered. Alternatively powering the windings of the stator commands the rotor flux so as to follow the stator field.

The number of steps per revolution of the rotor is given by,

$$S = Nr * P \qquad 3.1$$

Where,
Nr = number of rotor pole
P = number of stator phases

And the *stepping angle* in radian per each step is given by,

$$\theta_0 = \frac{2\pi}{S} = \frac{2\pi}{Nr.P} \qquad 3.2$$

If a sinusoidal characteristic of the magnetic field in the air gap is assumed, the contribution of each phase *j* on the motor torque T_{Mj} can be written as,

$$T_{Mj} = k_m.\ \sin(\phi_j + Nr\theta(t)).\ I_j(t)$$

Where, k_m = motor constant
$\theta(t)$ = actual rotor position
$I(t)$ = current in the coil as function of time
φ_j = location of coil j in the stator 3.3

However the current $I_j(t)$ in the coil is a function of the supplied voltage $V_j(t)$ and the coil properties. A general equation between $V_j(t)$ and $I_j(t)$ is given by,

$$V_j(t) = emf_j + R.I_j(t) + L.\frac{dI_j(t)}{dt}$$

Where, emf_j = electromotive force induced in the phase j
R = resistance of the coils
L = inductance of the coils 3.4

However, the EMF in each coil can be expressed as,

$$emf_j = k_m.\ \sin(\phi_j + Nr\theta(t)).\ \omega(t) \qquad 3.5$$

Where, ω = rotational velocity of the rotor

The total torque produced by the stepper is given as,

$$T_M = \sum_{j=1}^{P} T_{Mj} \qquad 3.6$$

Using Equation 3.6, and considering the equation of motion of a stepper motor,

$$T_M = J\frac{d\omega}{dt} + B\omega + T_l$$

Where, J = inertia of the rotor and the load 3.7
D = viscous damping constant
Tl = frictional load torque / load torque

The angular velocity is given by,

$$\frac{d\theta}{dt} = \omega \qquad 3.8$$

The above three equations (3.6, 3.7, 3.8) form the basis for a general state space description model of a PM stepper motor. Hence for a 2 phase PM motor with Nr rotor teeth and the two phases (ϕ_j) at 0 and (π/2) the following state space equations can be derived,

$$\begin{aligned}
\frac{d\theta}{dt} &= \omega && \%\ ang.vel \\
\frac{d\omega}{dt} &= \frac{(-k_m.I_a.\sin(Nr\theta) + k_m.I_b.\cos(Nr\theta) - B\omega - Tl)}{J} && \%\ load\ acceleration \\
\frac{dI_a}{dt} &= \frac{(V_a - R.I_a + k_m.\omega.\sin(Nr\theta))}{L} && \%\ current\ through\ winding\ a \\
\frac{dI_b}{dt} &= \frac{(V_b - R.I_b + k_m.\omega.\sin(Nr\theta))}{L} && \%\ current\ through\ winding\ b
\end{aligned} \qquad 3.9$$

Where the emf_j and V_j are given by Equations 3.4, 3.5..

Based on these basic equations a simple model of stepper motor can be developed in simulation software like *Matlab* or *Simulink* for simulation and analysis.

3.6 Control in PMS Motors

Originally stepper motors were designed for operation in open-loop configuration, to provide precise position control with an integer number of steps, without any sensors for feedback [41]. These are generally adequate for systems that operate at low accelerations with static loads, but closed loop control may be essential for high accelerations, particularly if they involve variable loads. Unlike servo motors, if a stepper in an open-loop control system is over torqued or is influenced by external loads and unmodelled disturbances all knowledge of rotor position is lost and the system must be reinitialized [13].

Furthermore at higher stepping rates the oscillatory nature of the motor adds to the loss of synchronism. In particular PM stepper motors have notoriously significant overshoot for step response which is often overcome by the use of dampers or operating at lower speeds [25]. Due to these problems, one is generally motivated to go ahead and consider feedback for stepper motors.

3.6.1 Field-oriented control

"Step motors, as typically driven in industrial applications, can exhibit undesirable behavior such as stepping resonances and skipped steps. However, this is because of the drive method that is used and is not due to the motor itself [33]".

In the case of a PM stepper motor, the current flow in each winding of the stator produces a magnetic field vector, which adds up vectorially to produce a net stator magnetic field

in arbitrary direction. The torque produced in the rotor is a result of the net stator field and the magnetic field of the PM rotor. The basic idea behind field-oriented control is that for any position of the rotor, there is an optimal direction of the net stator field which maximizes torque and there is also a direction which will produce no torque. If the stator field is orthogonal to the field produced by the rotor, then magnetic forces work to turn the rotor and torque is maximized. Thus by maintaining the stator magnetic field vector 90° (electrical) ahead of the magnetic field vector of the rotor, then the motor is field-oriented, and torque will be maximum (for a given power supply voltage) [33].

If the phase currents are sinusoids phased 90° with respect to each other the resulting stator magnetic field vector will rotate at the sinusoidal frequency. The field-orientated control method involves having sinusoidal voltage applied to phases such that they meet the 90° phase difference requirement of the currents, and position the stator magnetic field vector 90° ahead of the rotor flux vector.

This method of control that derives the maximum theoretical performance from the PM stepper motor was applied in conjunction with traditional control methods (P, PD) and RBF neural networks in this thesis work. The next chapter presents the procedure involved.

Chapter IV

RBF-NEURO CONTROLLER FOR STEP MOTORS

4.1 Introduction

Artificial intelligence computational procedures such as fuzzy logic, artificial neural networks and genetic algorithms, collectively known as “soft computing” techniques, were successfully used in the past decade, either directly or synergistically, for control of various complex systems. Learning based control methodologies such as neural networks and fuzzy logic based controllers has emerged as an alternative to adaptive control. The rationale for using neural controls or any other soft computing methods as such is directly related to the difficulties faced by control engineers in real-world applications.

Generally, it is quiet difficult to exactly represent (with minimum discrepancies) a complex process by a mathematical model or by a simple computer model. As seen from the control theory point of view, if a process (plant) itself is poorly modeled (or if the

parameter values are partially known, ambiguous or vague) appropriate estimates have to be made for the design of a controller. In such scenarios “crisp control algorithms” based on incomplete information may not give satisfactory results. A primary purpose of classical feedback is thus to increase the robustness of the system; i.e., to increase the performance of the system when there is uncertainty such as modeling errors, unknown disturbances and noise [40]. Furthermore, as stated earlier, it is a commonly known fact that the performance of industrial processes can be considerably improved through high-level control actions made by an experienced or skilled operator, which cannot (in most cases) be formulated as crisp control algorithms [11].

Robust and adaptive control (both parametric and nonparametric) techniques have been extensively developed for a variety of control problems to cope with uncertainties due to large parameter variations and thus achieve required levels of performance. Although the region of operability is considerably increased compared to non-robust classical control systems, these techniques lack the feature of learning [10]. That is to say, the control scheme cannot use the knowledge it has acquired in the past to tackle similar situations in the present or the future. As a result the same adaptation operation must be repeated every time the system is confronted with similar operating conditions. To tackle such problems “intelligent control” techniques have been developed, neuro-control being one of those.

The successful operation of an autonomous machine depends on its ability to cope with variety if unexpected and possibly unfamiliar events arise in the operating environment,

perhaps relying on incomplete information [12]. It is evident that such an autonomous machine would have to be presented with a goal which it would try to achieve through continuous interaction with external ambience and automatic feedback of its response. By enabling machines to posses such a level of autonomy, they would be able to learn higher-level cognitive tasks that cannot be easily learned by existing machines [9]. This in fact is the essential part of learning found in the majority of biological control as described in Chapter II. As presented earlier in this work, neural networks have a great potential in the realm of nonlinear control problems and they have been successfully used for system identification and control.

4.2 Adaptive control using ANNs

A neuro-controller (neural network based controller) in general performs a specific task of adaptive control, with the controller taking the form of a multi-layered neural network and the adaptable parameters being defined as adjustable weights [9]. This approach defines the problem of control as the mapping of measured signals of system "change" into calculated "control actions" as shown in *Fig. 4.1* below.

Neural control has existed for many years and there is a large body of empirical evidence of its viability in non-linear control applications [40]. Also, theoretical findings have guaranteed the stability of neural controllers for problems that can be composed as a nonlinear auto regressive moving average model [21].

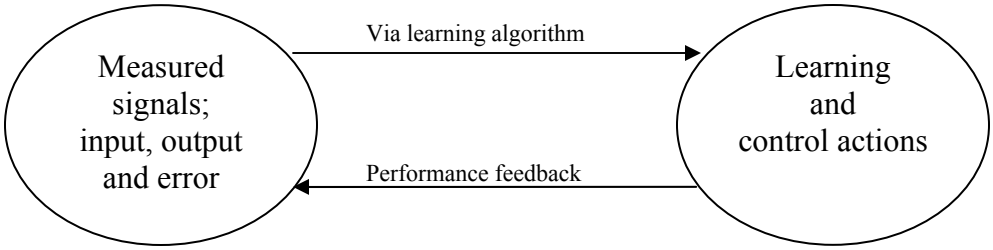
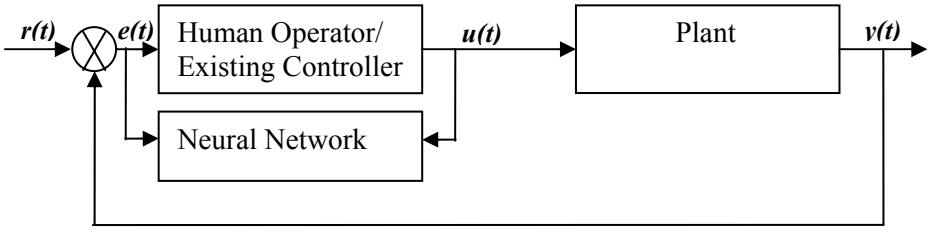
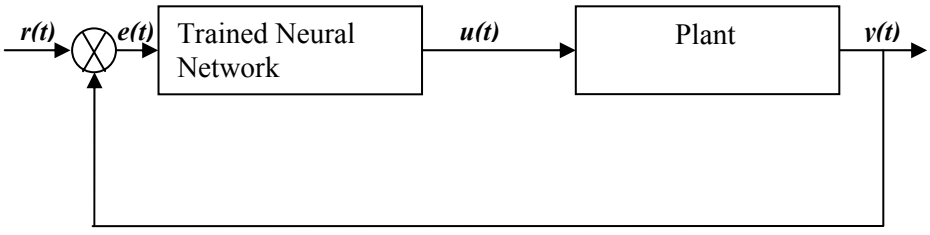


Fig. 4.1 Representation of learning and control actions in an ANN approach. Mapping measured signals onto learning and control space [90].

A number of neural-controllers have been proposed over the years, which can be classified in three major categories [17]; supervised control, neural adaptive control and direct inverse control.



(a) Network training phase



(b) On-line control phase

Fig. 4.2 Supervised Control

Supervised Control: In this method, a neural network is trained to perform a control task similar to that of a human operator. Thus, training data is collected in advance (on-line) from a existing physical system. *Fig. 4.2(a)* shows a network in the on-line training phase while *Fig. 4.2(b)* shows a neural network in on-line control phase. This type of control has been employed in many control applications like aircraft landing control [27] and neural net robot controllers.

This type of neural network controller doesn't require any explicitly stated control objective. Typically the objective of the neural network is to just find a mapping which will map sensor inputs to desired actions as accurately as possible. However, the training data has to be carefully examined for inconsistencies and contradictions, since such data may cause difficulties in training neural nets and may even cause instability and erratic behavior (in the case of robots) [40].

Neural Adaptive Control: This is a much more sophisticated on-line control scheme for the control of non-linear plants. In this type of neuro-control a neural network is first used to identify the system parameters and then the controller is tuned as in a conventional, adaptive control structure like model referenced adaptive control (MRAC) or self tuning regulator (STR) [22]. This type of control scheme is also called the indirect learning architecture. It is more complicated in the sense that it involves not one but two dynamic ANNs and thus its training is considerably more difficult.

In this type of control scheme one ANN is trained to model the plant dynamics while the other ANN performs the controller's task using a feed-forward neural network, where both the ANNs are trained on-line. Figure 4.3 shows the flow diagram of such a neuro-controller.

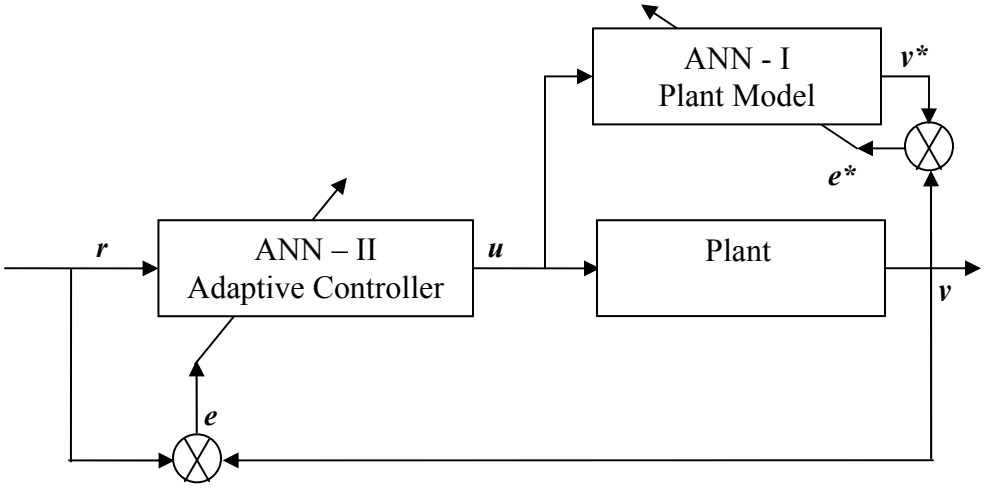


Fig. 4.3 Indirect learning architecture

Direct Inverse Control: In this type of control a neural network is trained to learn the inverse dynamics of a system. Thus, the inverse dynamics model will provide the input that will generate a particular output. This approach has been successfully used in robotic manipulators [26], where the manipulators are moved around and system inputs and manipulator positions are recorded. A neural network is then trained so that given trajectory positions the NN-controller will generate desired inputs for the motor joints. *Fig. 4.4(a)* and *4.4(b)* illustrate two inverse function learning architectures. The first one, referred to as generalized learning architecture, provides a method for training the neural controller that minimizes the overall error. The training procedure is given as below.

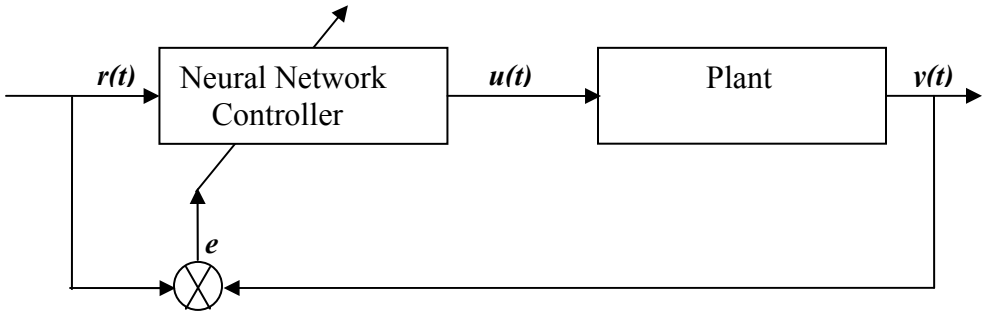
a) A plant input u(t) is selected and applied to the plant

b) The corresponding output from the plant y(t) is obtained

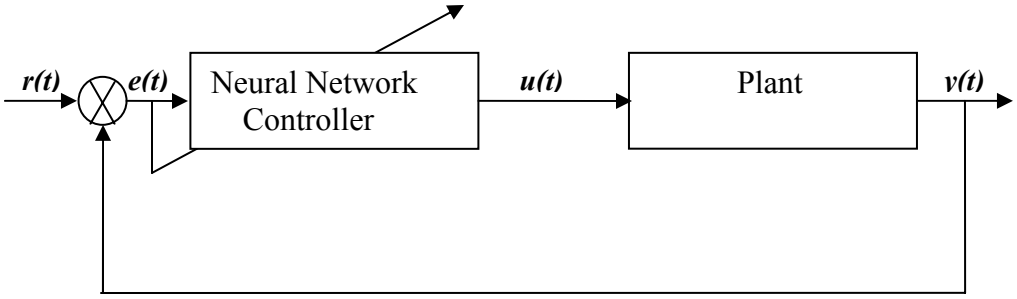
c) The ANN is trained to reproduce u(t) when y(t) is given as the input to the network.

After training such a neural network, during on-line operation will reproduce input ***u*** to the plant for a desired response ***r***. However, it has been argued that this architecture by itself would be difficult to use in real applications, since it is difficult to know in advance the region of interest in which the plant may operate. To overcome this problem a specialized learning architecture (Fig. 4.4b) has been used in the literature [37]. This architecture uses the difference between the actual and desired outputs to modify the weights of the inverse model. Thus, this approach takes into consideration the operating regions of the plant.

Successful implementations of direct inverse control schemes have been widely reported in the literature. CMAC networks have been used to learn the inverse dynamics of plants such as robot manipulators in [18] and the manufacturing of complex thermo plastic structures [36].



(a) Generalized learning architecture



(b) Specialized learning architecture

Fig. 4.4 Direct Inverse Control

As indicated in [19] learning the inverse model is among the more viable options for applications of neural networks in control. It must be noted that when the plant inverse is not causal or well-defined as indicated in [30] or if plants exhibit non-linearity and variations in parameters due to noise and other environmental factors, much effort is needed to apply this approach to real-time plants.

4.3 Problem Formulation

This thesis work considers speed control for permanent magnet stepper motor. A new intelligent control scheme based on a kind of *direct inverse controller* using a radial basis function neural network is proposed.

Originally, stepper motors (Chapter III) were designed to provide precise positioning control within an integer number of steps without the use of any sensors [25] in open-loop operation. The basic equations of a PM stepper, Equations 3.9, derived in Chapter III, are restated below. A stepper motor model has been developed in MATLAB® for simulation and analysis of standard open loop operation.

For a 2 phase PM motor with Nr rotor teeth and the two phases (ϕ_j) at 0 and (π /2) the following state space equations can be derived,

$$\frac{d\theta}{dt} = \omega \qquad \text{\% ang.vel}$$

$$\frac{d\omega}{dt} = \frac{(-k_m.I_a.\sin(Nr\theta) + k_m.I_b.\cos(Nr\theta) - B\omega - Tl)}{J} \qquad \text{\% load acceleration}$$

$$\frac{dI_a}{dt} = \frac{(V_a - R.I_a + k_m.\omega.\sin(Nr\theta))}{L} \qquad \text{\% current through winding a}$$

$$\frac{dI_b}{dt} = \frac{(V_b - R.I_b + k_m.\omega.\sin(Nr\theta))}{L} \qquad \text{\% current through winding b}$$

Where the emf_j and V_j are given by the equations 3.4, 3.5..

$$V_j(t) = emf_j + R.I_j(t) + L.\frac{dI_j(t)}{dt}$$

Where, emf_j = electromotive force induced in the phase j

R = resistance of the coils

L = inductance of the coils

However, the EMF in each coil can be expressed as,

$$emf_j = k_m . \sin(\phi_j + Nr\theta(t)) . \omega(t)$$

Where, ω = rotational velocity of the rotor

Now, if $\boldsymbol{x}$ denotes the state vector of the nonlinear system such that,

$\boldsymbol{A}(\boldsymbol{x}(k),\boldsymbol{u}(k)) = \boldsymbol{x}(k+1) = [x_0, x_1, x_2, x_3]^T$ given by Equation 3.9,

If $\boldsymbol{y}$ denotes the output vector so that,

$\boldsymbol{y}(k) = \boldsymbol{C}(\boldsymbol{x}(k))$

and given a desired trajectory $\boldsymbol{r}$ in terms of output vector. The problem is to find a suitable control input $\boldsymbol{u}(k+1)$, so that the system tracks the desired trajectory with an acceptable bounded error in presences of disturbances while all states and controls remain bounded.

The following characteristics of the PM stepper model (given in Table I) were chosen for simulation and maintained constant throughout this work, except for external torque disturbances and measurement noise that are simulated and added to the response externally.

It is evident from *Fig. 4.5* that using stepper motors in open loop configuration results in poor performance. In particular, one can notice that PM stepper motors have a step response with significant overshoot and long settling times which is often overcome by

the use of dampers or operating at lower speeds [25]. Due to these problems, one is generally motivated to go ahead and consider feedback for stepper motors.

Motor parameter	symbol	Value	Units
Rotor load Inertia	J	3.6 * 10^-6	N.m.s^2/rad
Viscous friction	B	1 * 10^-4	N.M.s^2/rad
External Load torque	Tl	0 (NONE)	
Self inductance of windings	L	0.001	H
Resistance in phase windings	R	8.4	Ohms
Number of rotor teeth	Nr	50	
Motor torque constant	Km	0.05	V.s/rad

Table I. PMS Motor simulation specs

With the increasing popularity of PM steppers over direct current drives, feedback controls have been proposed for stepper motor positioning systems. One such control idea is the exact feedback linearization technique. The basic design idea here is that the controller is a function of all plant parameters and external disturbances such as load torque [14]. In practice some of these parameters are subjected to variations and it is understandable that such a control technique is capable of providing excellent positioning results only if complete system dynamics are known. It actually results in no better performance than that of a conventional fixed gain controller (PD controller with gain scheduling if necessary) due to the fact that it is very hard to obtain a perfect dynamic model for the stepper motors in practice [25].

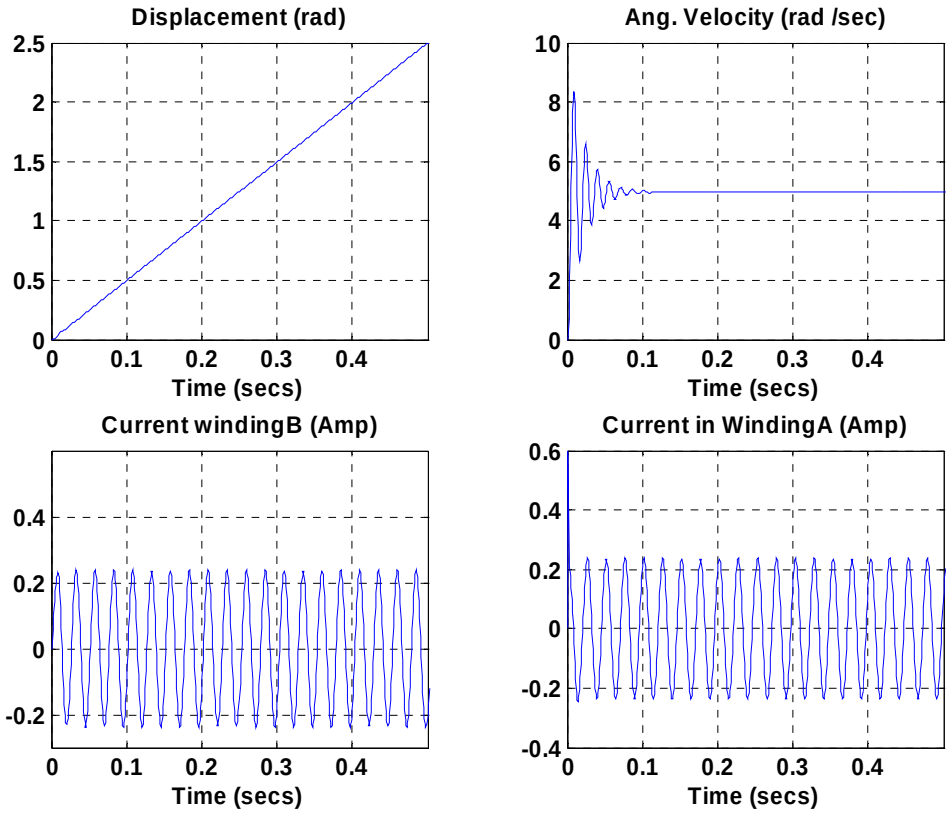


Fig. 4.5 Open loop response of a permanent magnet stepper motor.

On the other hand, there has been a strong interest in applying nonlinear control methodology to electric motors. The use of dynamic ANNs which allows efficient modeling of dynamic systems has been increasing to successfully model PM stepper motors [14], and brushless DC motors [38], or to provide robust compensating control as suggested by Gang Feng [25], for example.

Our design goal is to develop a discrete time, direct (inverse) adaptive controller for permanent magnet stepper motors. The structure of the proposed adaptive controller is "direct"; that is, there is no explicit attempt to determine the plant dynamics. The controller directly tunes its parameters in response to the measured deviations of the

process dynamics from its desired behavior. Thus, the difference between actual and desired outputs is used to tune the weights (parameters) of the neuro-controller (or the inverse model to be specific).

Before proceeding to the actual controller design a brief overview of radial basis functions and supervised learning algorithms is presented in the next section.

4.4 The Radial Basis Function Neural Network

4.4.1 Overview of RBFs

A general overview of neural networks and different architectures has been presented in Chapter II. A class of its own, the radial basis function neural networks, have only one hidden layer, in which each neuron (radial unit) each modeling a responsive surface (generally a Gaussian), defined by its center point and its radius (in $\boldsymbol{m}$ dimensional space). The following section describes the usage of a radial basis function in the context of interpolation and the development of RBF neural networks.

Consider an $\boldsymbol{m}$ dimensional input space $\boldsymbol{x}$ with a one-dimensional ouput space $\boldsymbol{t}$. The data set contains $\boldsymbol{N}$ input vectors $\boldsymbol{x}^n$ together with corresponding target vectors $\boldsymbol{t}^n$. Suppose the goal is to find a function h($\boldsymbol{x}$) such that,

$$h(\boldsymbol{x}^n) = \boldsymbol{t}^n \qquad n = 1,2 \ldots\ldots, \boldsymbol{N}. \qquad 4.2$$

In the radial basis function interpolation approach (Powell, 1987) introduces a set of ***N*** *basis functions,* one for each data point, which take the form g(||***x*** **-** ***x***^{n}||), such that the output mapping is a linear combination of the basis functions given as,

$$h(\boldsymbol{x}) = \sum w_n \, g(\|\boldsymbol{x} - \boldsymbol{x}^{n}\|) \qquad 4.3$$

According to Powell's exact interpolation technique, the interpolation conditions in Equation 4.2 can be written in matrix form as,

$$G = \begin{pmatrix} \| x^1 - x_1 \| & \cdots & \| x^1 - x_n \| \\ \vdots & \ddots & \vdots \\ \| x^{n'} - x_1 \| & \cdots & \| x^{n'} - x_n \| \end{pmatrix}$$

$$W = \begin{bmatrix} w_1 \\ \vdots \\ w_{n'} \end{bmatrix}; T = \begin{bmatrix} t_1 \\ \vdots \\ t_n \end{bmatrix}$$

$$G'.W = T \qquad 4.4$$

Here, n = n' = N and n' is the number of basis centers

Provided the inverse matrix $\mathbf{G}^{-1}$ exists we can solve (4.4) directly to give,

$$W = G^{-1}.T \qquad 4.5$$

Also, it has been shown (Michille, 1986) that, for a large class of functions g(.), the matrix **G** is indeed non-singular provided the data points are distinct. Thus, when the weights of the RBFs are set to as those given by Equation (4.5) the function h(*x*) represents a continuous differentiable surface which passes exactly through each data

point. Several forms of *basis functions* g(.) have been considered such as Gaussians, thin-plate splines, multi-quadratic, cubic etc. However, Gaussians have been most generally used due to their localized property; i.e., $g(x) \rightarrow 0$ as $x \rightarrow \infty$, which is given as,

$$g(x) = \exp(-x^2 / 2\sigma^2) \qquad 4.6$$

Here, σ is the variance parameter for a given dimension of $\boldsymbol{x}$ and controls the smoothness properties of the basis function. Since the activation functions are nonlinear, it is not actually necessary to have more than one hidden layer; sufficient radial units will always be enough to model any function [7]. It turns out to be quite sufficient to use a linear combination of these outputs (i.e., a weighted sum of the Gaussians) to model any nonlinear function.

Radial basis function mappings discussed above provide an interpolating function which passes exactly through every data point. However such an exact interpolation for noisy data is highly oscillatory in nature and not desirable. Another serious limitation of the exact interpolation technique is that the number of basis functions has to be equal to number of patterns in the data set, which becomes highly costly to evaluate in case of large data sets [7].

In seminal papers published by Broomhead and Lowe [3], a number of possible modifications to the exact interpolation techniques were suggested, thus giving rise to the Radial Basis Function Neural Network model (*Fig. 4.6*). Unlike the exact interpolation

described above, this provides a smooth interpolating function, in which the number of basis functions (hidden layer neurons) is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modifications required are summarized below [7];

1. The number ***m*** of basis functions need not be equal the number ***N*** of data points, and is typically much less than ***N***.
2. The centers of the basis functions are no longer constrained to be given by input data vectors. Instead, the determination of suitable centers becomes a part of the training process.
3. Instead of having a common width parameter 'σ', each of the basis functions is given its own width σ_j whose value is also determined during training.
4. Bias parameters are included in the linear sum. They compensate for the difference between the average value over the data set of the basis function activations and the corresponding average value of the targets.

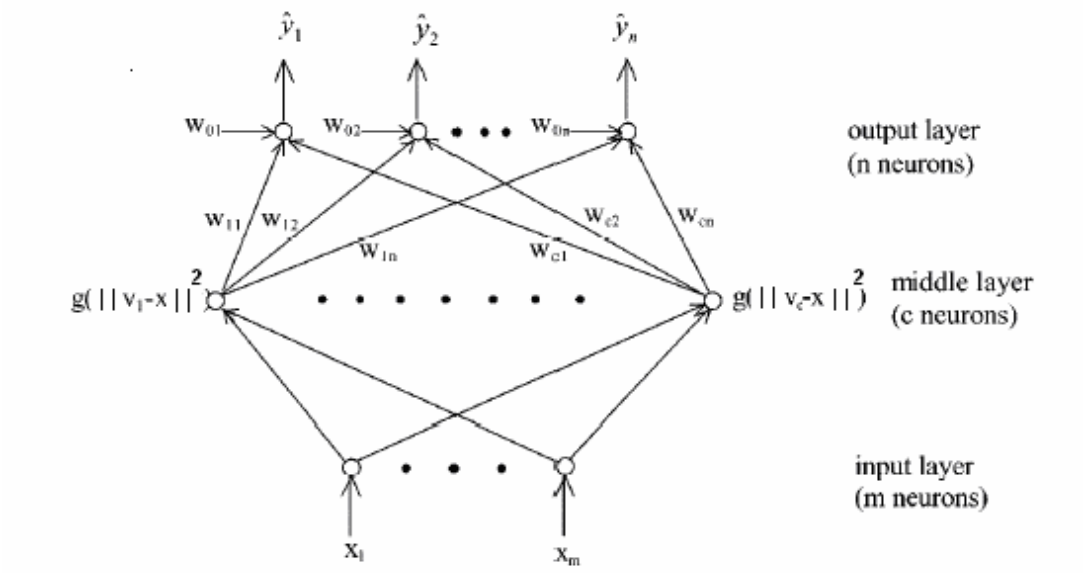


Fig. 4.6 Radial Basis Function NN

Consider the above network (*Fig. 4.6*) with ***m*** input neurons, ***c*** hidden neurons and n output layer neurons. Each of the ***c*** neurons in the hidden layer applies an activation function g(.) which is a function of the Euclidean distance between the input and an m-dimensional prototype vector. Each hidden neuron with its own prototype vector as a parameter gives an output that is then weighted and passed to the output layer. The outputs of the network consist of sums of the weighted hidden layer neurons given by:

$$\hat{y}_i = \sum_{j=1}^{c} W_{ji} C_j + W_{j0} \qquad 4.7$$
$$where, \quad C_j = g(||v_j - x||)$$

The Gaussian radial basis functions can be generalized to allow for on arbitrary covariance matrix $\sum$ such that,

$$g(\boldsymbol{x}) = \exp\{ - 0.5 * (\boldsymbol{x} - \boldsymbol{v_j})^T * \sum_j^{-1} * (\boldsymbol{x} - \boldsymbol{v_j})\}$$

4.8

In practice a trade-off is to be considered between using a smaller number of bases with adjustable parameters and a larger number of less flexible functions.

4.4.2 Optimizing the RBF-NN

A pronounced difference in Radial Basis Function architecture as compared to Multi Layered Perceptron (MLP) neural networks is the role of first and second layer weights, and leads to a two-stage training procedure of the RBF-NNs. In the first stage, the input data alone is used to determine the first layer weights (parameters of the basis functions $\boldsymbol{v_j}$, σ_j) by unsupervised training. The first layer weights are then kept fixed while the second layer weights are then *optimized* in second phase of training. However, note that after the modifications suggested in previous section, since fewer basis functions are used than data point it will no longer be possible to find a set of weight values for which the mapping function will exactly fit all the data points.

As given by Equation 4.7 the output of RBF-NN with bias values absorbed into main weight vector is given as,

$$\boldsymbol{y}(\boldsymbol{x}) = \mathbf{W} * \mathbf{G} \qquad 4.8$$

For a given training data set the target output data is ignored and basis function parameters prototypes (centers $\boldsymbol{v_j}$) with suitable widths are chosen such that they cover

the entire ***m***-dimensional input space covered by the training inputs $\boldsymbol{x^n}$. This can be simply done by closely looking at the range and density of the input patterns.

However it is worth noting here that, if such basis function centers are used to fill out the sub-space as stated above, then the number of basis function centers will be an exponential function of ***m*** [7]. This notorious problem with RBF-NNs, known as '*the curse of dimensionality*' is more pronounced in the case of input nodes that are largely uncorrelated. This also increases the computation time and number of training patterns required. These compelling reasons often lead to choice of unsupervised algorithms, such as '*K-means clustering*', to choose optimal first layer parameters depending on the density of the data points.

For the RBF-NN, given the target mapping values $\boldsymbol{t^n}$ for an input sequence $\boldsymbol{x^n}$ and after fixing the first layer weights (prototype vectors with centers $\boldsymbol{v_j}$, and basis widths σ_j), a suitable error function (cost function) ***J(ξ)*** where, $\xi = y^n - \hat{y}^n$ can be defined to optimize (tune) the second layer *weights*. The most commonly used cost function is the sum-of-squares error given below.

$$J(e) = \frac{1}{2} \sum_n \sum_k \{y_k(x^n) - t_k^n\}^2 \qquad 4.9$$

As is evident one of the principle advantages of the RBF-NN over its counterparts the MLP is the possibility to choose suitable parameters for the hidden layer neurons (prototype basis functions) and thus avoid the need to perform full non-linear

optimization of the network. However, it should be noted that this non-linear optimization problem is computationally intensive and can be prone to finding local-minima.

It is a well known fact that optimization algorithms which proceed by a steady monotonic reduction in the error function can become stuck in local minima. A suitable value of initial weights is therefore essential in allowing the training algorithm to produce a good set of weights, and in addition may lead to improvement in training speed. Majority of the initialization procedures in the current state of art involve setting weights to *randomly choosen small values* [7], to avoid problems that arise due to symmetry in the network and so that the activation functions are not driven into saturation regions.

Once the weights are properly initialized optimization (weight adjustment or training) can done iteratively using several algorithms. One of the simplest network training algorithms one is *gradient descent optimization,* also known as *steepest descent*. In *batch* version of a gradient descent an initial weight vector guess is made and a weight update is made at each step 'i' iteratively such that a move is made in direction of greatest rate of decrease of the error function,

$$\Delta w_i = - \eta \, (\partial J / \partial w_i) \qquad 4.10$$

Note that here in the *batch version* the gradient is reevaluated at each step. However, the the *sequential version* of this algorithm the error function gradient is evaluated for one pattern (*n* inputs) at a time and the weights are updated using,

$$\Delta w_i = - \eta \, (\partial J^n / \partial w_i)$$

4.11

The parameter η is called the *learning rate,* and provided this value is sufficiently small, as expected the value of J is bound to decrease at each successive step, eventually leading to a weight vector at which the condition, gradient (J) = 0 is satisfied.

In practice a constant value of η is often chosen. However, one serious limitation of this procedure is that if η is too large the algorithm may overshoot leading to an increase in J and possibly divergent oscillations, which lead to breakdown of the algorithm [7]. Conversely, if η is too small the search procedure can proceed extremely slowly leading to large computational times.

There have been several modifications made to the standard gradient descent algorithms to overcome the above mentioned limitations. One such is adding a *momentum term* μ to the gradient formula given in Equation 4.10

$$\Delta w_i = - \eta \, (\partial J / \partial w_i) + \mu \, \Delta w_{i-1} \qquad 4.12$$

The effect of momentum is to increase the *learning rate* from η to $\eta/(1- \mu)$.

While notable research has been done in this area of optimization, even with a momentum term included gradient descent is not a particularly effective algorithm for

error function minimization. Various adhoc modifications have been suggested. In the current thesis work a *bold driver technique* [Vogal et al, 1988] which has an automatic procedure to set the *learning rate* is used. The basic idea behind the algorithm is to check if error function has actually decreased after each step of gradient descent. If it has increased then an overshoot is recognized, the weight change is undone, and the learning rate is decreased. Also, if an error decrease is seen, then the new weight values are accepted and the learning rate, probably too small, is increased. The following is update law for the learning rate **η**,

$$\begin{aligned} \eta_{\text{new}} &= \text{k1} \;.\; \eta_{old} \quad \text{if} \;\; \nabla \text{J} < 0 \\ &= \text{k2} \;.\; \eta_{old} \quad \text{if} \;\; \nabla \text{J} > 0 \end{aligned} \qquad 4.13$$

The parameters ***k1*** and ***k2*** are chosen such that they are slightly greater and less than unity respectively, typically chosen as *k1 = 1.1, k2 = 0.5* [7]. The choice of k1 and k2 has to be done carefully to avoid changes in the *learning rate* and at the same time boost the speed of convergence.

4.5 Controller Design

Adaptation in the direct adaptive controllers discussed above involves online or sequential adjustment (training/tuning) of the parameters of the neural network in the feedback control loop so as to force tracking error (ξ) to tend to zero or at least remain bounded. The Radial Basis Function Neural Networks described above have been found more suitable than MLPs for online or sequential adaptation as they are insensitive to the order of presentation of training data.

Let **R** denote real numbers and $\mathbf{R}^n$ denote real n- real vectors. Let **S** be the compact simply connected set of $\mathbf{R}^n$ if F(.): **S** → $\mathbf{R}^k$ define a space $\mathbf{C}^k$(**S**) such that F(.) is continuous. By application of multivariable Fourier analysis and Whittaker/Shannon sampling theory, it has been shown [7] that Gaussian Radial Basis Functions arranged on a regular lattice on $\mathbf{S}^n$ are capable of universally approximating a smooth function to a chosen degree of tolerance everywhere on a specified subset. It must be noted that RBF–NN controllers are linear in the sense of tunable weights, which is a far milder assumption compared than the adaptive control requirement of linearity in parameters (LIP) . While, the latter holds only for a specific function F(x) the former holds good for all functions of F(x) $\in \mathbf{C}^m$(**S**). In the ANN property, the same set of basis functions g(x(k)) suffices for all F(x) $\in \mathbf{C}^m$(**S**) while, in the LIP assumption a regression matrix must be computed for each F(x). Also, in comparison with other ANN architectures the use of Gaussian activation functions, the RBF forms a local representation (unlike multi layered perceptrons) where each basis function responds only to the inputs in the neighborhood of a reference vector [32].

In this direct controller design the centers of the basis functions were placed on regular points of a square mesh covering a relevant region of space where the input space ($\xi = \boldsymbol{r} - \boldsymbol{y}$) of the RBF NN is known to be contained, denoted by a compact set $\xi_n \in \mathbb{S}^n$. It is assumed that desired trajectory vector with its delayed values is valuable for measurement. The weights of the network are then initialized randomly and the weights

are tuned with a supervised learning algorithm such as *bold driver gradient descent optimization*, with a suitable cost function ***J(ξ)***.

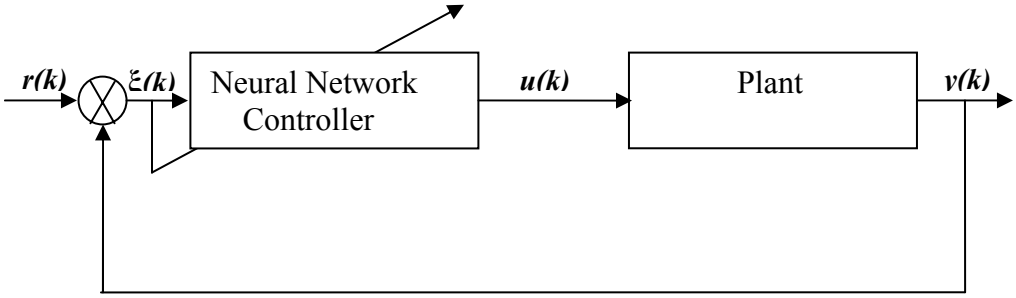


Fig. 4.7 Plant with RBF in feedback loop, representing training/control phases

Such a neuro-controller, once trained, during an on-line operation phase will reproduce input ***u*** to the plant for a desired response ***r***. Here the tracking error (ξ) becomes the driving force as the ANN is placed in series with the plant. This results in increased robustness of the system coupled with advantages of conventional feedback, since the training is based on "some measure" of closed system error (***J(ξ) = J(r – y)***). However, one has to accept the fact that training is more difficult with such a structure, due to feedback action through the ANN. This approach allows the ANN training to take place within the operating region of the plant and is hence more accurate than the generalized learning method described earlier in Section 4.3.1. Given below is the algorithm for implementation of the controller.

1. Decide the number of feedback samples of the measurement ***m*** to be used as the control input to the RBF and the number of control outputs (***n***) from the

neuro-controller (In this case ***n*** = 1, RBF output = M, the magnitude of control inputs *ua*, *ub*).

2. Define RBF network with ***c*** hidden layer neurons, ***m*** input and ***n*** output neurons.
3. Simulate the stepper motor system from time 0 to tf, with initial states given by $\boldsymbol{x}_0$, using a nominal controller (open loop), to get an ***m*** dimensional input space $\mathbb{S}^m$ for the RBF controller.
4. Define the activation functions (Gaussians) with centers $\boldsymbol{v}_j$ and their widths σ_j, both with ***m*** dimensions, where j varies from 1 through ***c***.
5. Initialize the network weights randomly.
6. Choose a suitable learning rate **η**.
7. Now, place the RBF-NN in feedback to deliver control magnitude (M) where the out of phase control signals are *ua* = M cos (Nr.ω_d); *ub* = M sin (Nr.ω_d).
8. Simulate the system for a small amount of time (0:tf) , acquire the measurement vector ***y*** and compute tracking error **ξ** (= ***r*** – ***y***).
9. Evaluate a cost function $J(\xi) = \lambda 1 . \| \xi \| + \lambda 2 . \| \Delta \xi \|$ .Where, $\lambda 1$ and $\lambda 2$ are chosen arbitrarily as weighting factors.
10. If this is not the first iteration, compare current J (**ξ**) with previous one, and perform learning rate update using bold driver technique given in 4.10 as follows, where k1,k2 are typically chosen as ***k1*** *= 1.1,* ***k2*** *= 0.5*

$$\begin{aligned} \eta_{new} &= k1 \; . \; \eta_{old} \quad \text{if } \nabla J < 0 \\ &= k2 \; . \; \eta_{old} \quad \text{if } \nabla J > 0 \end{aligned}$$

11. If $\nabla J < \varepsilon$, weights converged , GoTo END

12. For each weight w_i perturb it by a small amount Δw_i, and simulate the system for entire time length again to get $\hat{J}_i(\xi)$.Compute numerical partial differential $\frac{\Delta J_i}{\Delta w_i}$ and thus the weight updates using *gradient descent* described earlier in this chapter given by, $W_{inew} = W_{iold} + \eta . \frac{\Delta J_i}{\Delta W_{iold}}$.
13. Go To 8.
14. END

As, it can be noticed from the algorithm except for the learning rate parameter settings and a choice of suitable subspace $\mathbb{S}_m$ for deciding the input vectors , which can be easily done by using any nominal control there are no other tuning parameters required in this procedure. Such a controller once trained can be used in on-line mode with final converged weights which exhibits robustness for external disturbances, so long as the errors remain bounded with in $\mathbb{S}_m$. Also, the Gaussian nature of the activation functions ($g(\boldsymbol{x}) \rightarrow 0$ as $\boldsymbol{x} \rightarrow \infty$) ensures that the RBF control outputs remain bounded even if there are large errors.

The next chapter presents results for various settings of the RBF, the effect of choice of ***m, c*** and various weight initializations.

Chapter V
RESULTS AND CONCLUSIONS

5.1 Introduction

In this work a direct adaptive controller based on a radial basis function neural network has been developed for control of a PMS motor. Such a controller, once trained offline, learns the inverse plant dynamics and can be used to control the non-linear plant (PMS), better than traditional PD controllers used in the industry. This controller never considers any modeling parameters as design criteria, and therefore offers an edge over conventional controllers in that it doesn't require repeated tuning of the controller for different tracking trajectories.

Initially in this chapter the performance of the RBF-Neuro controller trained with various configuration changes to learn the inverse plant dynamics of the PMS motor is analyzed. All configurations are compared with a standard open loop controller. An analysis of the

weight and control surface adaptation in all the configurations is also presented. Later, the best configuration setting is chosen and the performance is compared with a classic PD controller tuned for two different trajectories. Finally, the robustness of the neuro-controller is demonstrated by adding external disturbances other than those for which the controller has been trained.

The following criteria are used as performance index for analysis purpose,

1. Maximum peak error
2. Steady state error
3. RMS error

Additionally, for the comparison of different settings of neuro-controllers the *cost function* $J(\xi) = \lambda 1. \| \xi \| + \lambda 2. \| \triangle \xi \|$ is used as the performance index.

5.2 Performance Analysis of RBF-Neuro Controllers

In this section a neuro-controller placed in the feedback loop is trained initially as described in Chapter IV. The RBF parameters, such as number of centers, and their distribution and weight initializations, are varied and their effect on the response of the controller is studied. This study is limited to single dimensional input and hidden layered neurons of the RBF. That is, the RBF has only one input neuron providing *tracking error* delayed by one time step ($\xi_{(k-1)}$).

The following simulation parameters are used consistently throughout the study,

Simulation parameter	Value	Units
Initial time (t0)	0	Sec
Final time (tf)	1.9	Sec
Simulation time step (dt)	0.0001	Sec
Control time step (dtCntrl)	0.001	Sec
Integration method used	Rectangular	N/A

Table II. Simulation specs

A PMS motor with the following characteristics is used consistently for all simulations presented in this chapter. The initial condition is chosen to be $\boldsymbol{x_0}$ = [-0.183, 5, 0, 0.119] in all cases, where $\boldsymbol{x}$ signifies the state vector i.e., the displacement, angular velocity and the currents in windings B, A respectively.

Motor parameter	symbol	Value	Units
Rotor load Inertia	J	3.6 * 10^-6	N.m.s^2/rad
Viscous friction	B	1 * 10^-4	N.M.s^2/rad
External Load torque (Square wave)	Tl	5 * 10^-4	V.s/rad
Self inductance of windings	L	0.001	H
Resistance in phase windings	R	8.4	Ohms
Number of rotor teeth	Nr	50	

Table III. PMS motor simulation specs

As explained in Chapter III, field oriented control derives the maximum theoretical performance from the PM stepper motor. The field-oriented control method involves having sinusoidal voltage applied to phases such that they meet the 90° phase difference

requirement of the currents, and position the stator magnetic field vector 90° ahead of the rotor flux vector. Thus, to provide out of phase winding currents to both phases, the control inputs are chosen to be sinusoid in nature, where the control input (M) will be magnitude of these winding currents. The control inputs are given by Equation 5.1 below.

$$ua = M * \cos (Nr.\omega_t)$$
$$ub = M * \sin (Nr.\omega_t) \qquad 5.1$$

5.2.1 Neuro-Control vs. Open Loop Control

Originally stepper motors were designed for operation in open-loop configuration, to provide precise position control with an integer number of steps, without any sensors for feedback [41]. Thus, for a given constant velocity trajectory a simple open loop controller can be easily obtained as per Equation 5.1. *Fig. 5.1* shows the open loop response of the PMS system for a constant velocity trajectory of ω_d = 5 rad/sec. A square wave external load torque with a magnitude of 5*10^–4 was chosen as the external disturbance input [refer to Table III] . The control inputs are chosen as follows,

$$ua = 1 * \cos (Nr.\omega_d)$$
$$ub = 1 * \sin (Nr.\omega_d)$$

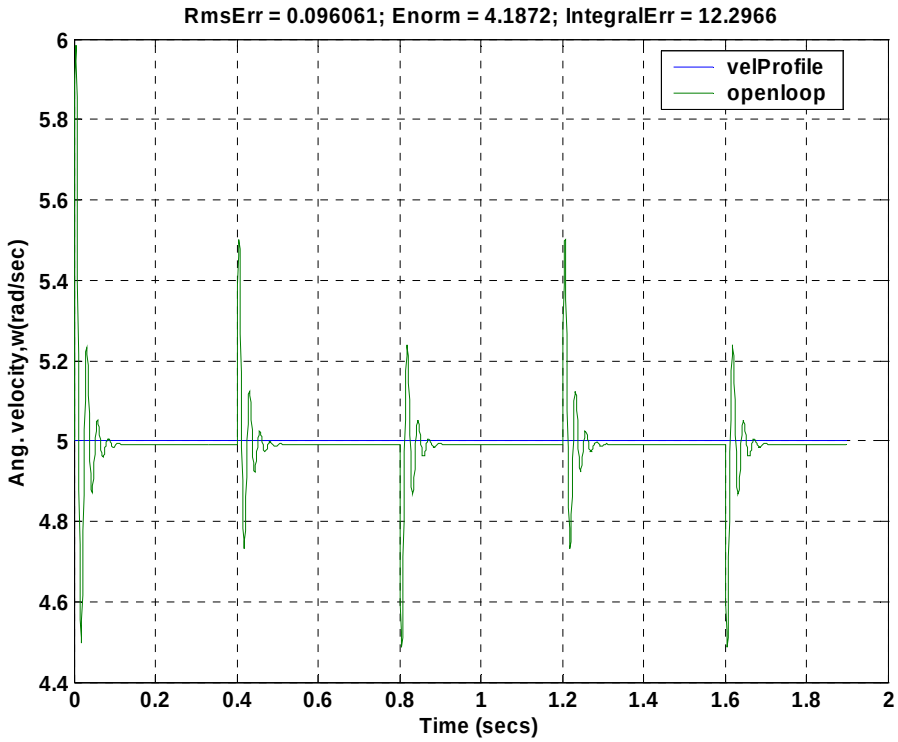


Fig. 5.1(a) Open loop response of PMS plant

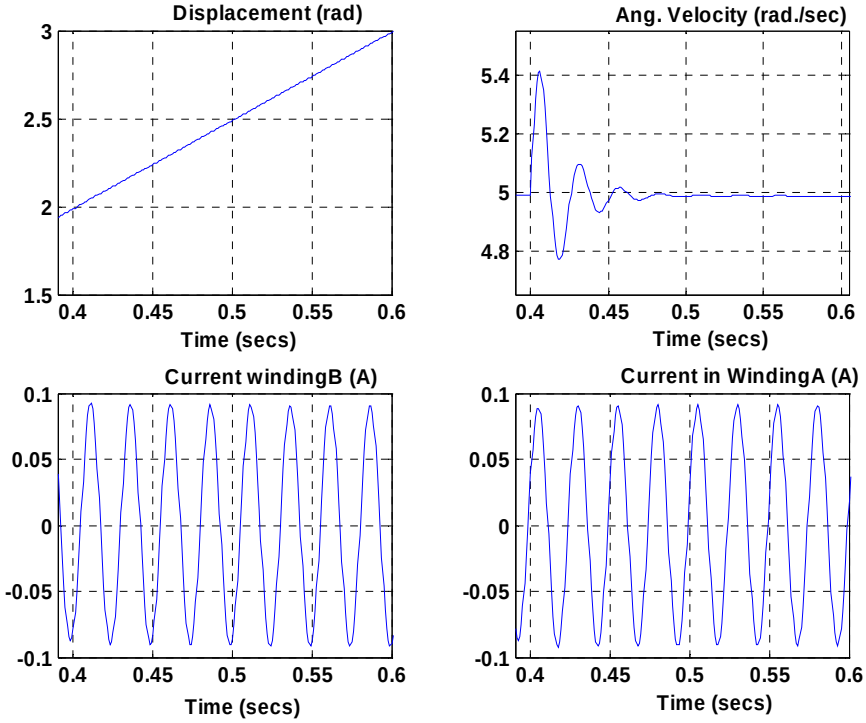


Fig. 5.1(b) Open loop response of PMS plant

It can be clearly noticed that such an open loop control such a response is not desirable in presence of external disturbances. This necessitates the need for a closed loop controller. Although it is simple to implement a classic PD controller, to overcome the burden of tuning each time an adaptive RBF Neuro Controller that learns the inverse plant dynamics is trained. The neuro controller is then trained as described in Chapter IV for the same velocity trajectory (ω_d = 5 rad/sec) with the parameters given in Table IV.

RBF parameters	Value
Input neurons	1
Output neurons	1
Hidden neurons	18
Input space	±2
Basis centers	Distributed equidistantly, overlapping each other between -2 to 2 (including both boundaries)
Basis Radius	0.5*(4/17) ≈ 0.1177
Bias	1

Table IV. RBF-Neuro Controller – [Config. 16, Table V]

As mentioned, in this control method the input to the RBF is the tracking error. So, based on open loop control and knowledge of plant dynamics an input space is assumed a priori and the RBF-NN activation function centers are distributed uniformly in this space (refer to Table IV). The hidden layer neuron prototype vectors are distributed evenly in the input space, and weights are initialized with random numbers. Weighting in the cost function is chosen as $\lambda_1 = 0.2, \lambda_2 = 0.8$ such that $\Delta\xi$ has more weight in the error function. Also, the perturbation value for computing the partial differential is chosen consistently to be 0.1% of the original value.

Fig. 5.2 shows the reduction in the cost function after optimization performed by using the *bold driver gradient descent* algorithm. *Fig. 5.3* shows the adaptation of weights. It can be seen that after a considerable number of iterations the weight values start to converge to the best possible solution using the gradient descent algorithm.

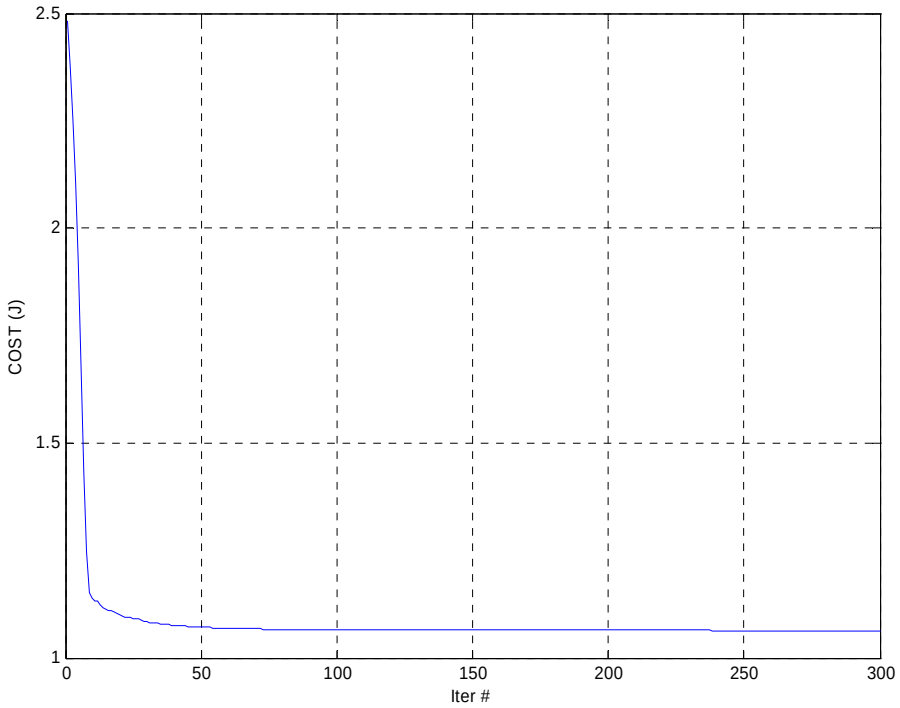


Fig. 5.2 Reduction of cost function

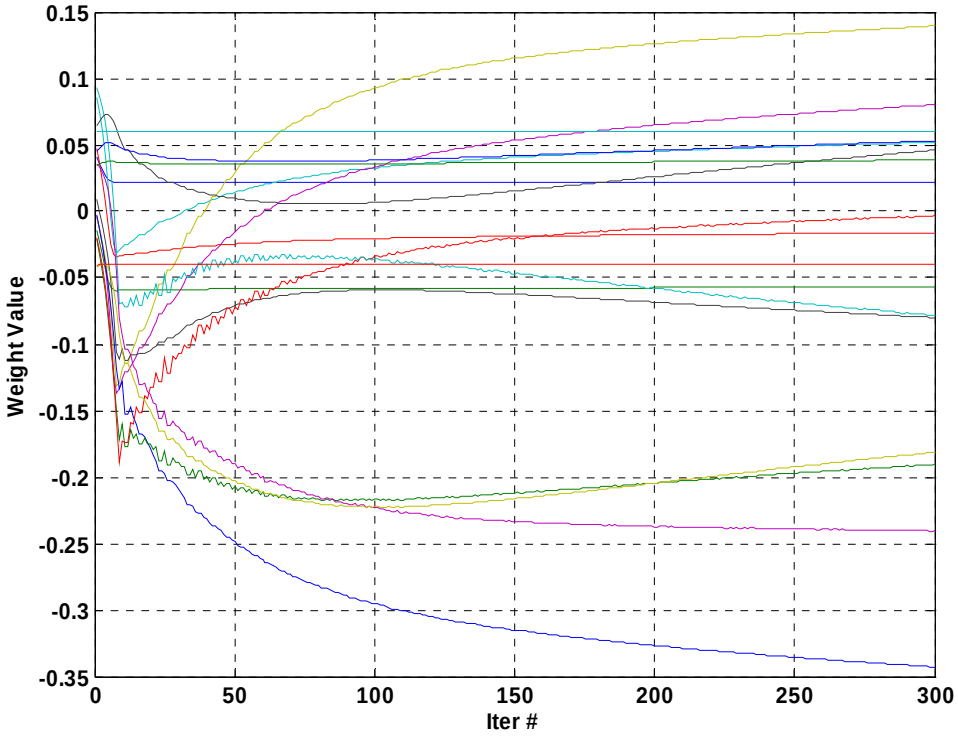


Fig. 5.3 Adaptation in weights

The learning of the controller during the training process can be best judged by looking at the change in the RMS value of the tracking error shown in *Fig. 5.4*. *Fig. 5.5* shows the response of RBF controller for the input space it has been defined for, both before and after training. It is worth noting that for values of tracking error outside the input space (±2 in this configuration of the RBF, refer Table IV) the output of the RBF controller is M = 1, thus acting as a nominal open loop controller.

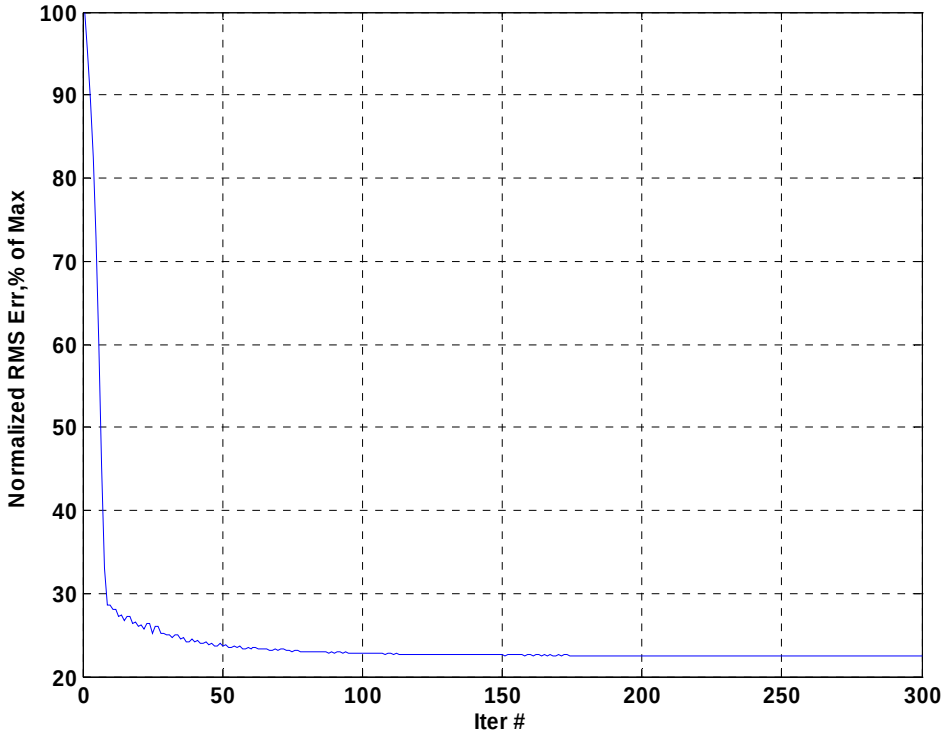


Fig. 5.4(a) Change in RMS error (% of max. error)

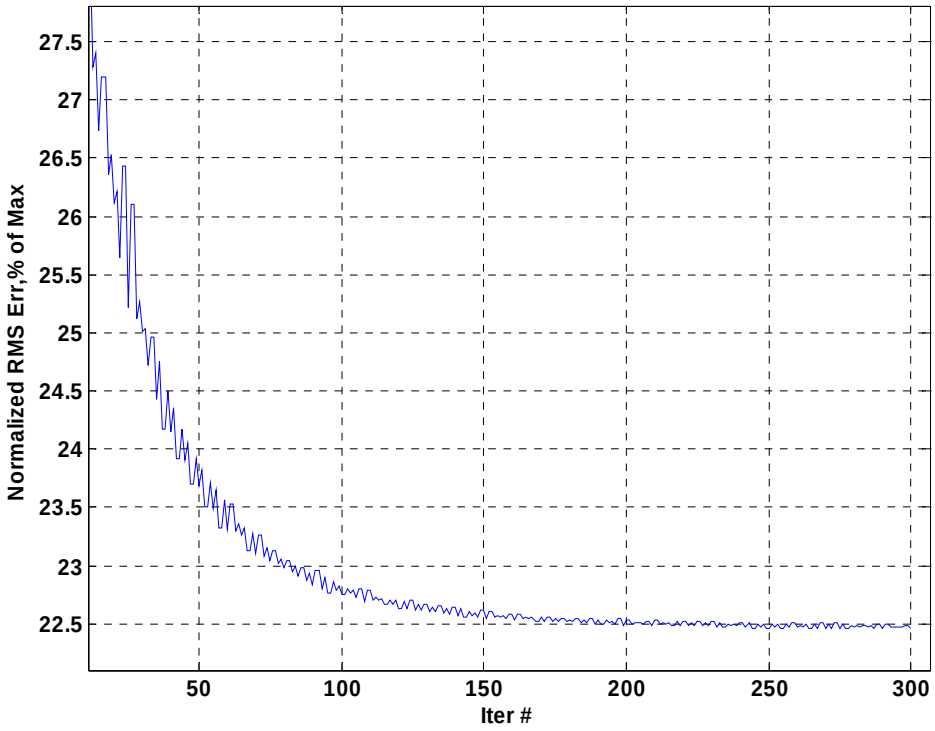


Fig. 5.4(b) Change in RMS error, [Fig. 5.4(a)]50 to 100 iterations

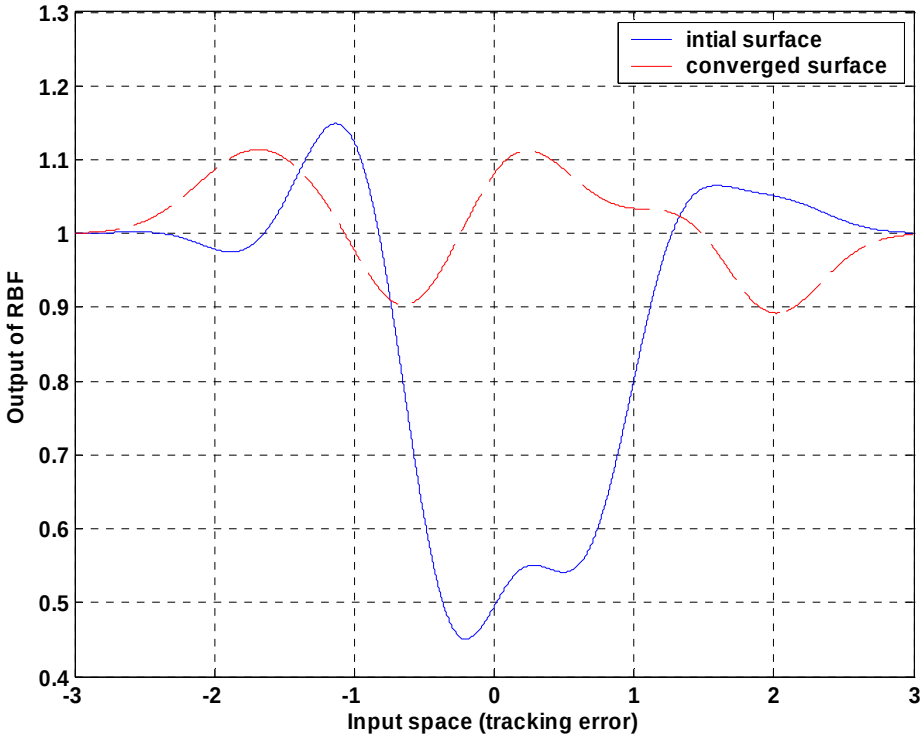


Fig. 5.5 Adaptation of control surface

Finally, as a comparison between the open loop controller and the RBF controller, *Fig. 5.6(a)* shows a plot of neuro-controller response, while *Fig. 5.6(b)* shows the response of both open loop [refer to *Fig. 5.1(a)*] and RBF-neuro control methods. It can be seen that steady state, RMS and peak errors are all reduced in case of the later, thus justifying the use of the neuro-controller.

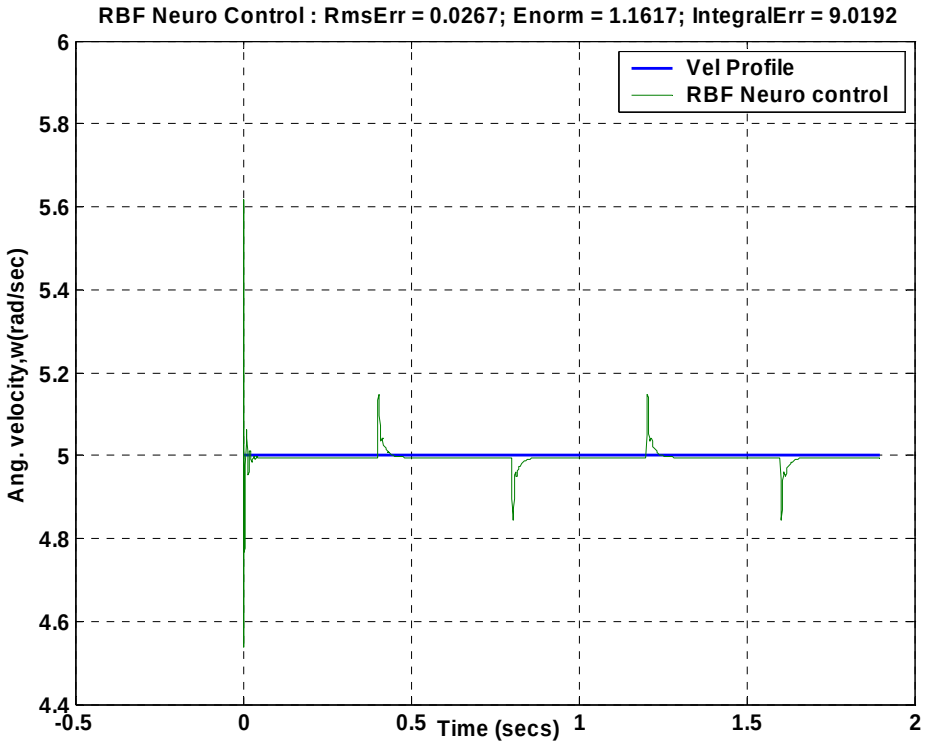


Fig. 5.6(a) Neuro Controller response of PMS plant

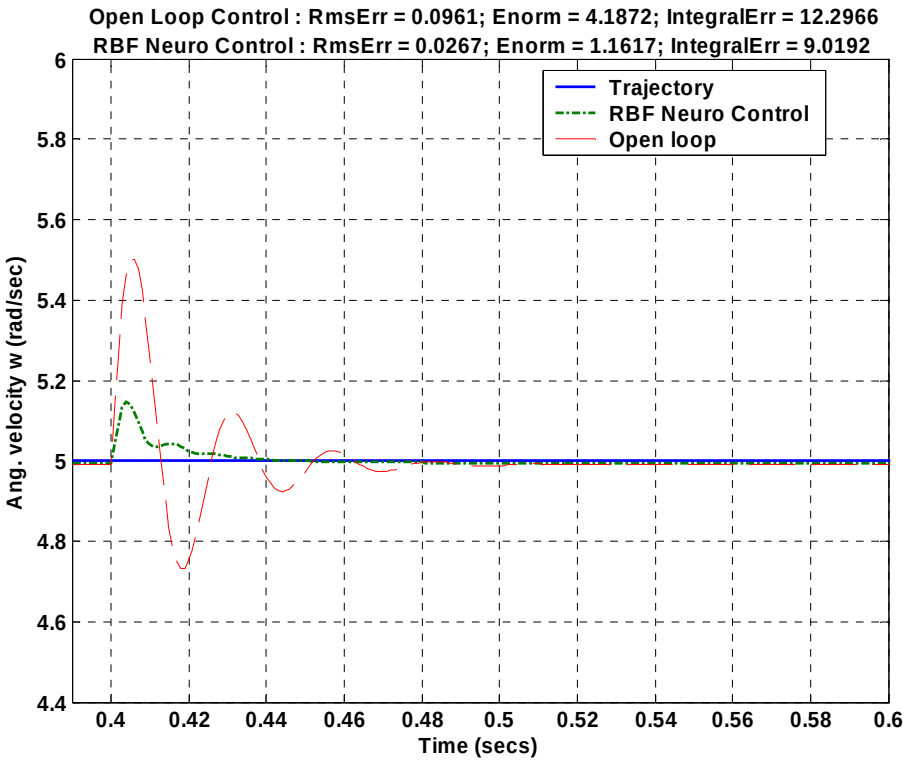


Fig. 5.6(b) Open loop vs. RBF Neuro Control
[Time scale 0.4 to 0.48 seconds]

Now, to evaluate the effect of RBF parameters such as weight initializations and the number of hidden layer neurons, each of the parameters is varied as shown in Table V thus giving rise to a variety of configurations.

RBF Configuration	#Hidden neurons	Matlab random number seed for weight initialization
1	5	-99
2	5	777
3	5	-1
4	5	0
5	10	-99
6	10	777
7	10	-1
8	10	0
9	15	-99
10	15	777
11	15	-1
12	15	0
13	18	-99
14	18	777
15	18	-1
16	18	0

Table V. RBF Neuro Controller Configurations

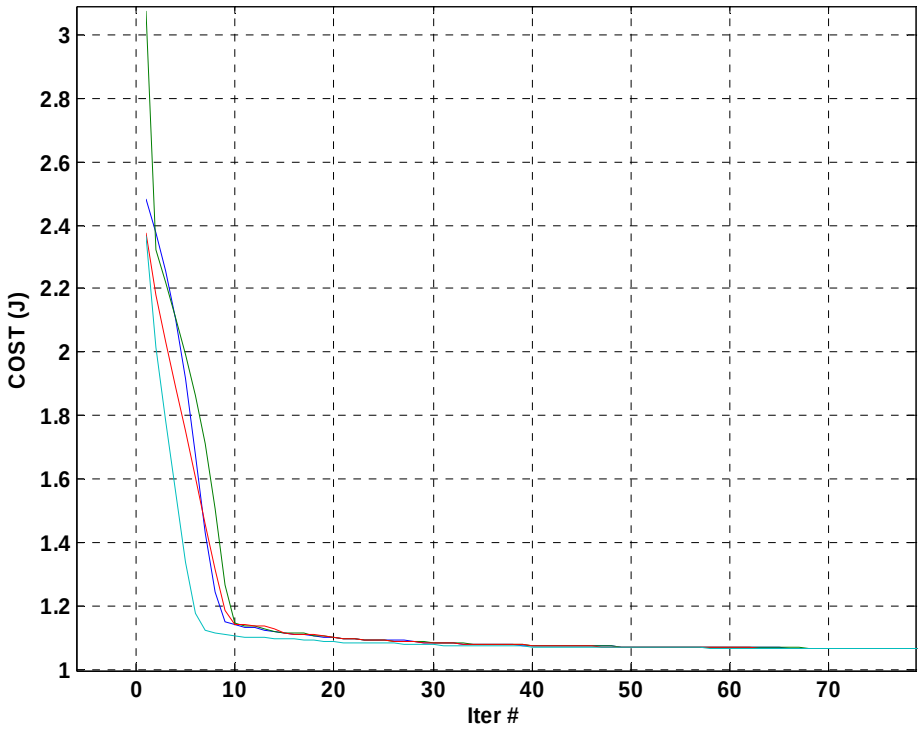


Fig. 5.7(a) Effect of random initialization on cost (J) of the controller

To notice the effect of different weight initialization on the controller performance, the configurations with same number of neurons are grouped together and RMS error and Cost functions are shown in *Fig. 5.7 (a), Fig. 5.7 (b)* for configurations 13 through 16. It can be clearly seen that except for difference in the path the controller takes to converge, so long as the number of neurons is same the final RMS errors in each of these cases is same. Similar analysis can be done with other number of hidden neurons. This shows the robustness of the RBF training procedure. In spite of the highly nonlinear error surface, the training procedure converges to the same configuration regardless of the initialization values.

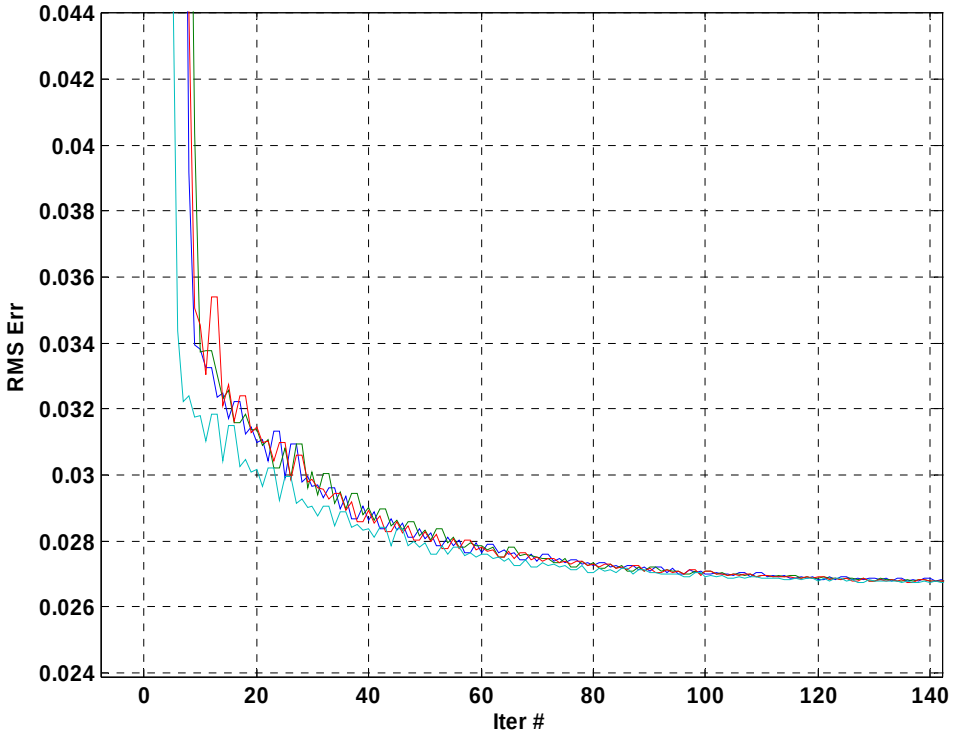


Fig. 5.7(b) Effect of random initialization on RMS error (J) of the controller

To notice the effect of number of neurons on the controller performance, the configurations with the same random initializations (configurations 4, 8, 12, and 16 from Table 5.4) are grouped together and cost functions optimizations are shown in *Fig. 5.8(a), Fig. 5.8(b)*. It can be seen that as the number of neurons increases, the cost convergence and hence performance of the controller improved. Also, Fig.5.8(c) shows the control surface for all the configurations which shows that a robust control surface as the number of neurons increases both within and outside the input boundaries of the RBF-NN. Thus, the controller with maximum number of neurons (18) is chosen for comparison with a classical PD controller in the next section.

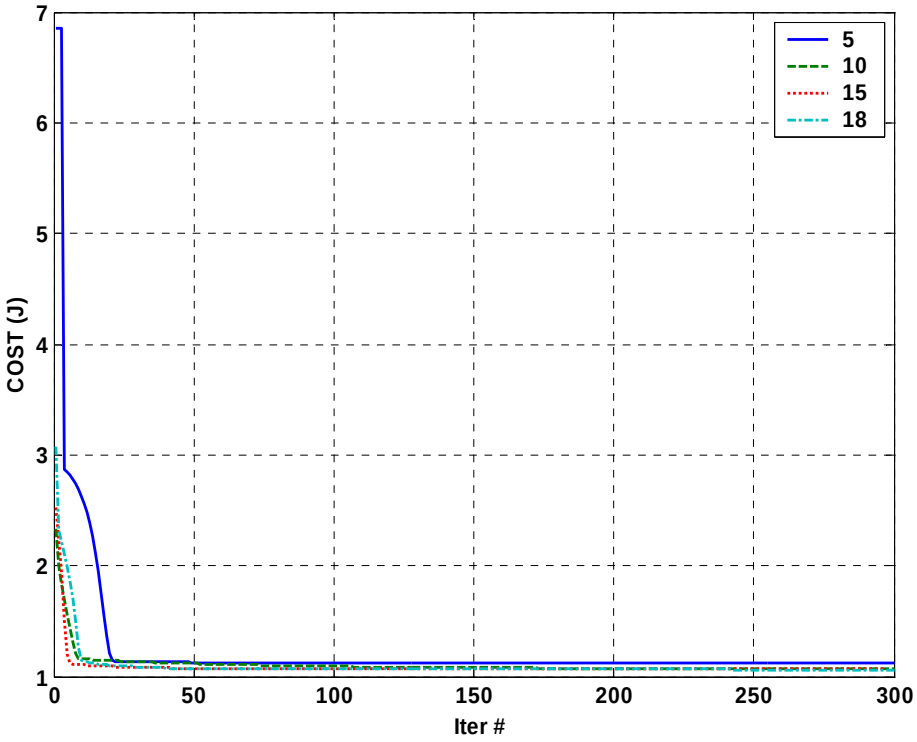


Fig. 5.8(a) Effect of number of neurons on performance of the controller

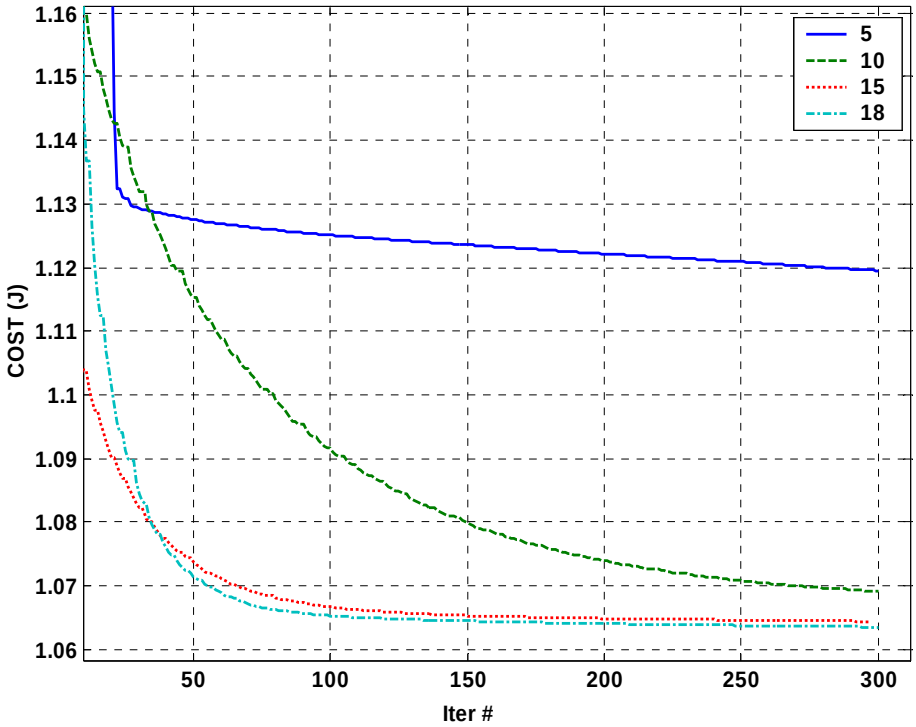


Fig. 5.8(b) [Fig. 5.8.(a)] Iterations 50 to 300

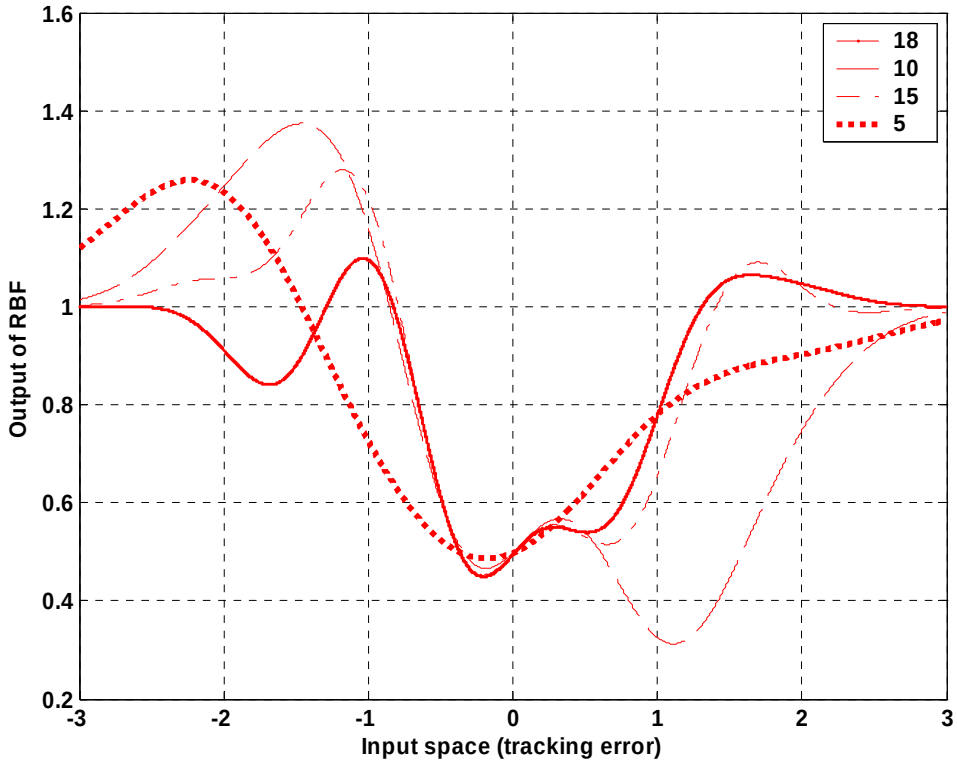


Fig. 5.8(c) RBF NN control surface
[1-d for various number of Hidden layer neurons]

5.2.2 Neuro-Controller vs. PD-Controller

Initially the operating conditions are chosen to be $\boldsymbol{x}_0$ = [-0.183, 5, 0, 0.119] as stated earlier for open loop control, where ***x*** signifies the state vector i.e., the displacement, angular velocity and the currents in windings B, A respectively. All other plant parameters as specified in Table 5.1. A neuro-controller with 18 hidden layer neurons is trained to learn the inverse plant dynamics in Configuration 16 given by Table 5.4. A PD controller is then tuned by trial and error to obtain fixed gains. An additional tuning parameter ***Ks,*** a constant additive gain, was needed to amplify the gain and drive the stepper and required tracking velocity. The tuned fixed gains for a desired tracking velocity of ω_d = 5 rad/sec are,

Ks = 0.8, ***Kp*** = 0.63, ***Kd*** = 1.8*10^–4

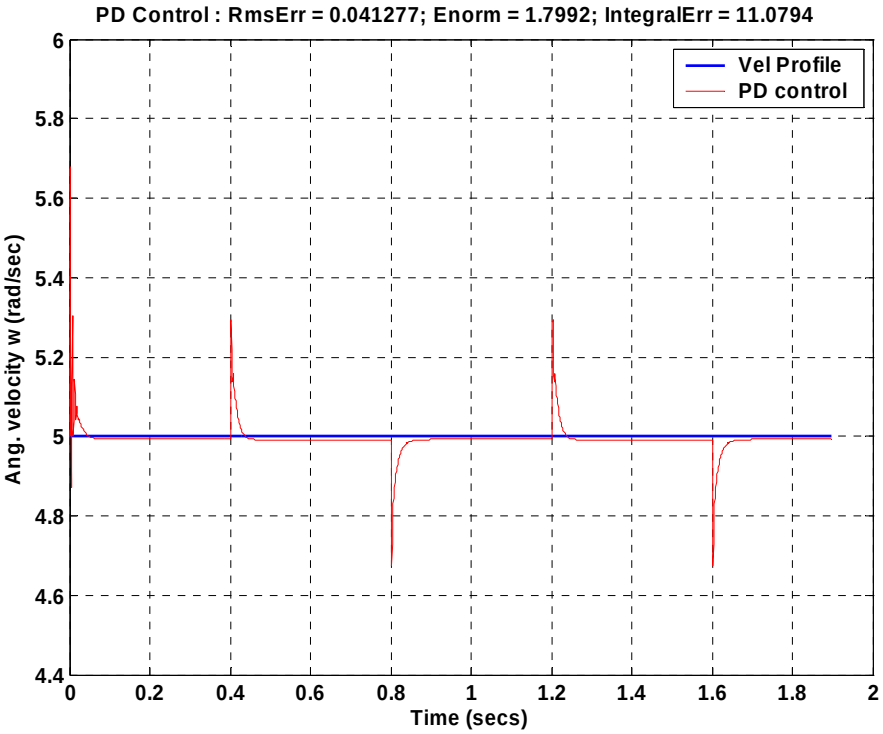


Fig. 5.9(a) PD Control

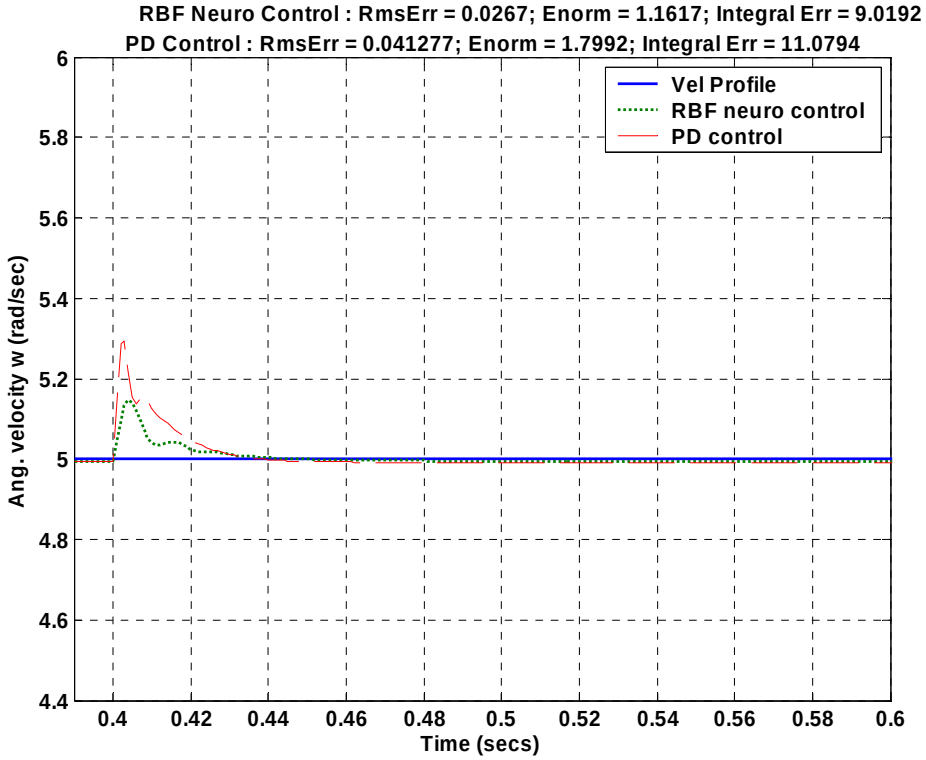


Fig. 5.9(b) Open loop vs. RBF Neuro Control
[Time scale 0.4 to 0.48 seconds]

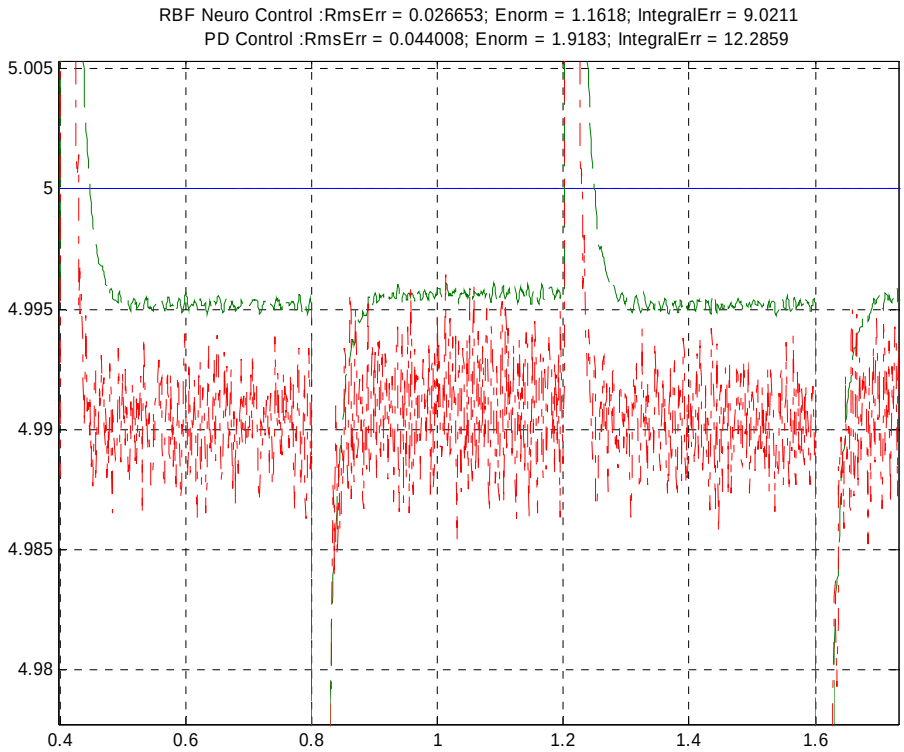


Fig. 5.10 RBF Neuro Controller vs. PD Control with zero mean, white measurement noise

Fig. 5.9(a) shows the trajectory tracking of a PD controller, while *Fig. 5.9(b)* shows a comparison of trajectory tracking by both PD and RBF neuro controllers. It can be seen that the peak and steady state errors are both reduced considerably in case of the neuro-controller. Adding white noise to the measurements considerably deteriorates the performance of the PD controllers. However, due to its smooth interpolating surface and Gaussian activation functions, the RBF neuro controller still tracks the trajectory effectively as shown in *Fig. 5.10*. *Fig. 5.11* shows the performance of both the controllers for different trajectories other than those they were tuned for.

The RBF controller can be seen to track well due to its nonlinear nature, as long as the error inputs to the network are within the training bounds. However, PD controller gains have to be tuned again to derive optimum performance. Tuned gains used are :

Ks = 0.5, ***Kp*** = 0.58, ***Kd*** = 1.8*10^–4.

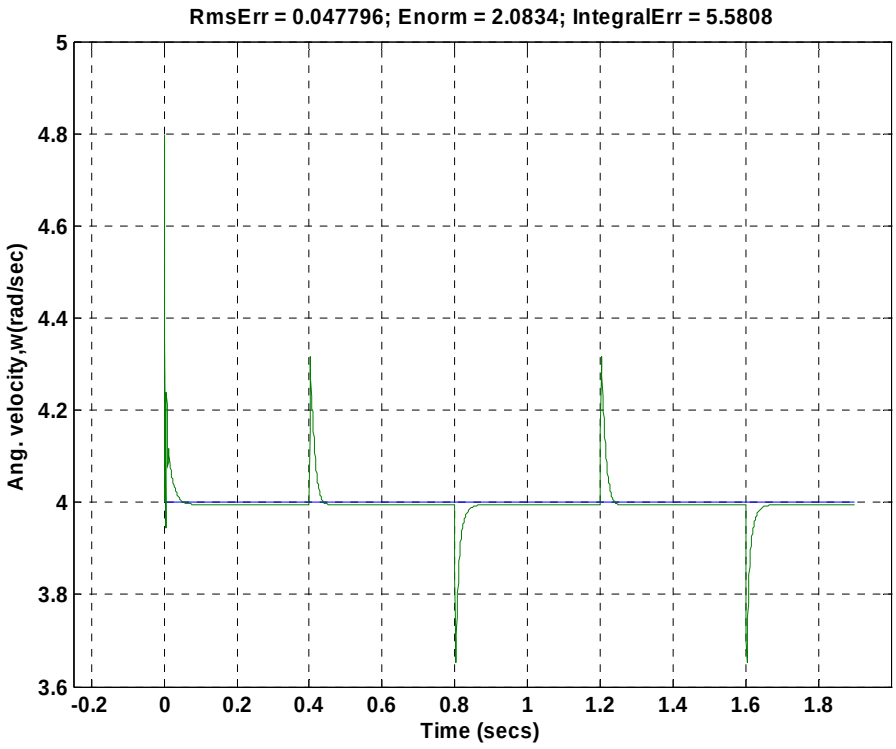


Fig. 5.11(a) PD Controller (ωd = 4 rad/sec)

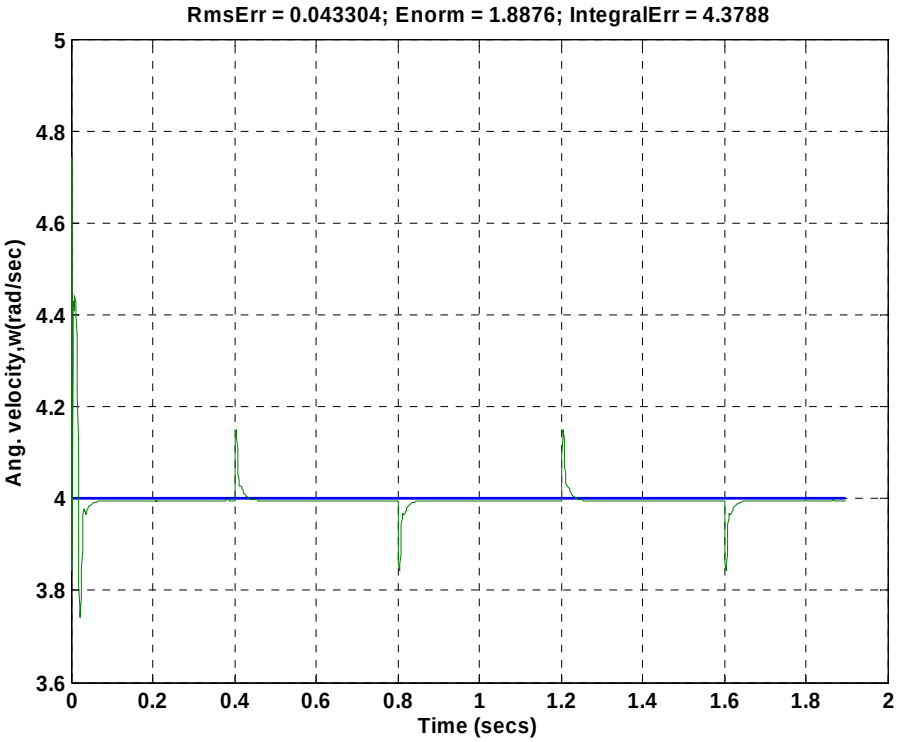


Fig. 5.11(b) Neuro Controller (ωd = 4 rad/sec)

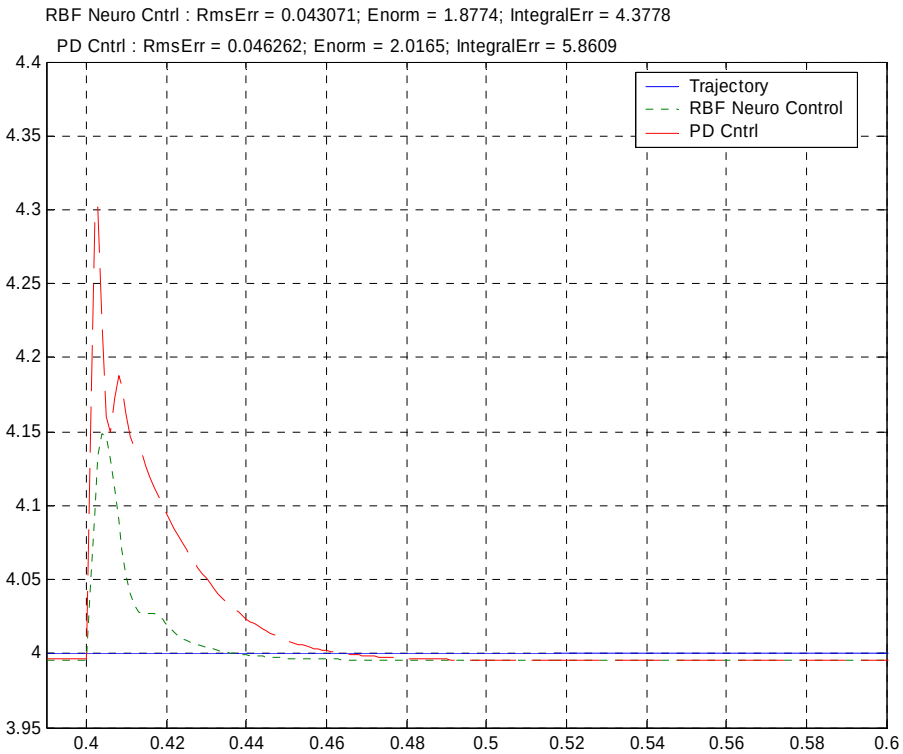


Fig. 5.11(c) [Fig 5.11(a),(b)] Time scale 0.4 to 0.48 seconds

5.3 Conclusions

It can be seen that for any given trajectory, the RBF controller tuned within a limited subspace learns the nonlinear dynamics of the plant and thus exhibits robustness to external disturbances and measurement noise. Given adequate training data the RBF controller learns the plant dynamics for the entire subspace and exhibits robustness throughout the operation range. They best way of operation for the current version of the controller is to use it in conjunction with a nominal controller to drive the plant to this limited error subspace where the RBF controller can then act as a compensating controller. The idea here has been to propose a new and efficient adaptive control that would reduce the burden of the control engineer to retune gain parameters. Listed below are the advantages and disadvantages of using the RBF neural network for control of a PMS motor.

Advantages

1. Auto tuning capability for a given tracking error subspace
2. Robustness to external noise disturbances
3. Provides nominal control for errors outside the input subspace, ensuring safe operation of the controller.

Disadvantages

1. No model information has been used
2. Complex controller mathematical model.
3. Requires large training time.
4. No proof of stability. More complex as the dimensions of the ANN increases.
5. Current model still offline, even though on-line version is possible.

6. Limited to single dimensional prototype vectors due to computational burden.

However, it has to be noted that some disadvantages of the neuro-controller like the proof of stability, using prior knowledge of system dynamics are unavoidable like any other model free method such as a PD or a PID controller. It has been shown that the RBF-neuro controller is a robust controller which provides a adequate control even when the tracking error is outside the error subspace for which it has been trained, thus ensuring safe operation of the system unlike the derivative based methods.

5.4 Future work

Much work has already been done to learn nonlinear plant dynamics in feed forward mode for system identification and then use the inverse model to control the system. The current work is only a preliminary step towards automatic direct adaptive control using radial basis function neural networks for PMS motors. However, with a little more effort, by making the optimization routines more efficient, this work could be extended to make an on-line version of this neuro-controller, thus making it comparable to other adaptive control techniques. Also, a detailed stability analysis such as a Lyaponov analysis could be done. The other possible extension is to increase the number of delayed feedback samples input to the RBF controller (increasing the input dimensions of the RBFNN) and thus make it more robust.

1 Arbib, Michael A. “Brains, Machines, and Mathematics: Second Edition” Springer-Verlag, New York, NY, 1987.

2 Bose, N. K. and Liang, P. ``Neural Network Fundamentals with Graphs, Algorithms, and Applications''. McGraw-Hill, New York, NY, 1996.

3 Broomhead,D.S and Lowe,D. (1988), "Multivariable interpolation and adaptive networks",Complex systems2,321-355.

4 Brown, R.H., Rutchi, T.L., and Feng, X., "Artificial Neural Networks identification partially known dynamic non-linear systems", Procs. of the 32nd Conference on Decision an Control, Vol.4,pp.3694-9,1992

5 Cairo L. Nascimento Jr., 1994, “Artificial Neural Networks in Control and Optimization”, PhD. Dissertation.

6 Chen,F. and Khalil,K.H (1992), "Adaptive control of a class of nonlinear systems using neural networks",IEE Trans. on Automatic Control, vol.40,no.5,791-801.

7 Christopher M. Bishop, “Neural Networks for Pattern Recognition: Second Edition”, Oxford University Press, New York, 1995.

8 DARPA Neural Network Study 1988, AFCEA International press

9 De Silva, C.W and Lee,T.H., "Knowledge-Based Intelligent Control", Measurements and Control, vol.28(2), pp. 102-113, April 1994.

10 De Silva, C.W., Fakreddine, O. Karray., "Learning and Adaptation in complex dynamic systems", Intelligent control applications and industrial applications, CRC Press 1999.

11 De Silva, C.W., Intelligent Control : Fuzzy Logic Apllications., CRC Press 1995.

12 Desilva,C.W., "Intelligent Control",Fuzzy Logic contrl Applications, CRC Press,1995.

13 Douglas W.J ., "Control of Stepping Motors" (online tutorial),THE UNIVERSITY OF IOWA, Department of Computer Science.

14 Edgar N. Sanchez, Alexander G.Loukianov, Ramon A.Feilx, "Dynamic Triangular Neural Controller for Stepper Motor Trajectory Tracking", IEEE transactions on Systems MAN and Cybernatics.

15 Filippidis, A. and Jain .L.C,"Intelligent control techniques", Intelligent control applications and industrial applications, CRC Press 1999.

16 Gumma.S, Vasarla.J, Choudary.S.G, Choudary.T.Rao, 2001, “A Character Recognition system using Artificial Neural Networks”, Bachelors Thesis.

17 Gupta, M. and Rao, H., Neural-Control Systems theory and Applications, Newyork press, 1994

18 Kawato,M. (1990),"Computational schemes and neural network models for formation and contorl of multijoin arm trajectory", Neural Networks for Control,T.Miller et. al(Eds) MIT Press,Cambridge,MA,1990

19 Khalid,M.(1990), "A neural network controller for temperature control system," IEEE Control magazine, Vol.12,No.3, pp.58-64,1992.

20 Kiyoshi Kawaguchi, 2000, “A multithreaded software model for backpropagation neural network applications”

21 Narendra,K.S and Mukopadhyay, S., "Adaptive Control Using Neural Networks and Approximate Models", IEEE Trans on Neural Networks,Vol.8,No.3,pp.475-485,1997

22 Narendra,K.S and Parthasarathy,M. (1990) "Identification and control using neural network models :design and stability analysis,Tech. report 91-09-01,Dept. of Elect. Eng. Sys.,Univ.S. California.

23 Paul Acaarnley,"Stepping Motors a guide to theory and practice", 4E 2002, McGraHill.

24 Polycarapo,M. and Ioannou,P.A (1991), "Adaptive control of unknown plant dynamics using neural networks IEEE Trans. on systems,Man, andCybernatics,vol.24,no.7,pp.971-98.

25 **Postion control of PM stepper motor using neural networks, GangFeng**

26 Psaltis,D.,Sideris A., and Yamamura,A.A., "A multi-layered neural network controller," IEEE Intl. Conf on Neural Networks, VOl.3,pp. 1926-31,SanFransico,CA,1993

27 Rjorgensne,C.C. and Scheley,S. (1990),"A neural network nbaseline problem for control of aircraft flare and touching down" Neural Networks for Control,T.Miller et. al(Eds) MIT Press,Cambridge,MA,1990

28 Robert E. King, "Computational Intelligence in Control Engineering", Marcel Dekker,Inc.,1999

29 Sadegh,N. (1993),"Aperceptron network for functional identification and control of nonlinear systems", IEEE Trans. on Neural Networks,vol4,no.6,pp.982-988.

Samad,T., "Neurocontrol:concepts and applications",IEEE Intl. Conf. on Systems,Man and
30 Cybernatics,vol.1,pp.369-74,Chicago,IL,1992
Sarle, W.S., ed. (1997), “Neural Network FAQ, part 1 of 7: Introduction”, periodic posting to the Usenet
31 newsgroup. [comp.ai.neural-nets]
Simon Fabri, Visakan Kadrikamanathan, "Dyanamic Structure Neural Networks for Stable Adaptive Control o
32 Nonlinear Control Systems"
Simon, D., and Feucht,D., “DSP-Based Field-Oriented Step Motor Control,” SHARC International DSP
33 Conference, Boston, MA, pp. 303-309, September 2001.
Simon,D., "Get Your Motor Running", Embedded Systems ProgrammingEmbedded Systems Programming,
34 vol. 16, no.5, pp. 20-26, May 2003
Simon,D., “Training Radial Basis Neural Networks with the Extended Kalman Filter,” Neurocomputing, vol.
35 48, pp. 455-475, October 2002.
Sofge,D.A. and White,D.A. (1990),"Neural network based process optimization and control," Proc. of the 29t
36 Conf. on Decision and Control,1990.
Thibult,J. and Gradjean,B.P.A., " Neural Networks in process control - a survey " Advanced Control of
37 Chemical Process,pp.251-60,1992.
Tipsuwanpron. V., Piyarat .W. and Tarasantisuk.C., "Identification and Contrl of Brushless DC mottors using
38 on-line Tranined Artifical Nerual Networks", Proceeding of the power conservation conference, osaka,2002.
Werbos,P.J., "An overview of neural networks for control", IEE Control Systems Magazine,Vol.11,ISS.1,pp
39 40-41,1991.
Wu, Q.M., Stanely, K.,De Silva, C.W., Jain .L.C,"Neural control systems an applications", Intelligent control
40 applications and industrial applications, CRC Press 1999.
Zribi,M. and Chiasson, J.,"Position control of PM stepper motor by Exact linearization", IEE transactions of
41 automatic control vol.36. No.5,May 1991.
42 Zurada, Jacek M. “Introduction to Artificial Neural System”, West Publishing Company, St. Paul, MN, 1992.

APPENDIX

SOFTWARE LISTING

```
========================================================================

                M.file,function,"offLineTraining"

========================================================================
function offLineTraining(nc,randCase,fileName)
global initTraj


format short e;


% record all command line activites
cd results
   diary(fileName)
cd ..


% Cost fn for optimization
COSTFN = '.2*norm(errVec) + .8*norm(diff(errVec))';
fprintf('\n The costfn for this trial is: \n \t E = %s \n',COSTFN);


% -------------------- Initalize Control parameters ---------------- %



% Intialize rbf,centers and parameters
rbf = rbfInit(randCase,nc);


% Command window info. display ...
```

```
fprintf('\n This trial is for %d inputs to RBF in feedback :
\n',rbf.nin);disp(rbf);
fprintf('\n');disp(rbf.w2);

% ------------------ Initialize system parameters --------------------
%
  sys = specs;                  % get system specs
  Wd =  5;                      % angular velocity to be tracked
  tf = 1.9;                     % final time
  Y0 = [-.0183;Wd(1);0;0.119];
  fd = 10^3;                    % control frequncy
  dtCntrl = 1/fd;               % desired freq. rad/sec
  trajWd = [];
  fprintf('Wd = %s \t tf = %2.2f \t fd = %d \n',num2str(Wd),tf,fd);

% ---------------- Construct Profile vector -------------------------
%
velProfile = Wd*ones(size(0:dtCntrl:tf-dtCntrl));
Wt = sys.Nr*Wd(1);Wtvec = [];

Evec = [];

% Bold driver grad. descent parameters
k_up = 1.2;k_down = .5;NumSplits = 1;
fprintf('eta_k = %f \t k_up = %2.2f \t k_down = %2.2f
\n',rbf.eta,k_up,k_down);
fprintf('\n');

% TRAIN RBF until error converges
```

```
iter = 1;EEvec = [];allData = [];allWeights = [];CONVG = [];



while(1)
    % ------------------------------------------------ %
    % Start clock for this iteration:
      T0 = clock;

    % do intial simulation to get standarad error, i.e,oldW
    % :::: SYSTEM SIMULATION START ::::
      runNeuroCntrlPlant;
    % :::: SYSTEM SIMULATION END ::::
    % above script updates closed loop plant output to errVec

      E = eval(COSTFN);
      Evec = [Evec;E];

    % check E for convergence
    if (iter > 1)
          % if five consecutive errors match up to 'n'th decimal place,
          % declare, CONVERGED ...
          if (length(Evec) > 5)
             CONVG = round(diff(Evec(end-5:end)).*10^5);
          else
             CONVG = 99;                 % dummy value
          end

          if ((iter > 300) | (CONVG == 0))
```

```
            fprintf('\n\tRBF-NN CONVERGED at ITER # : %d !!\n ',iter);
            diary off;
            break;
          end

        Elast = Evec(end-1);
        % Bold Driver Grad. Descent Algorithm
        if (Evec(end) > Elast)
            % increase in error, so restore prev weights
            rbf.w2 = oldW;
            E = Elast;
            Evec = Evec(1:end-1); % remove Maxima

            % decrese step size
            rbf.eta = rbf.eta * k_down;
            NumSplits  = NumSplits + 1;
            fprintf('\n \t This is LOCAL MINIMUM  ...\n ');
            disp(rbf.w2);
        else
            % if in correct direction increase step size
           rbf.eta = rbf.eta * k_up;
        end
    end
% ------------------ TRANING PHASE --------------------- %
% copy old weights
  oldW = rbf.w2;
  tempW = zeros(size(oldW));
  doE = zeros(size(oldW));doW = zeros(size(oldW));
```

```
% Adjust weights one after the other
  for ro = 1:rbf.nhidden
      for col = 1:rbf.nout
        % copy weights
        tempW = oldW;

        % change ith weight
        tempW(ro,col) = oldW(ro,col)+ (rbf.gradient * oldW(ro,col));

        % copy new weights
        rbf.w2 = [];rbf.w2 = tempW;

        % :::: SYSTEM SIMULATION START ::::
        runNeuroCntrlPlant;
        % :::: SYSTEM SIMULATION END ::::

        % Error with Weights(ro,col) changed:
        delE = eval(COSTFN);
        doE(ro,col) = (delE - E)./(tempW(ro,col) - oldW(ro,col));
      end
   end

% update new weights with GRADIENT DESCENT
  rbf.w2 = oldW - rbf.eta*(doE);%./doW);
  T1 = clock;Tlap = etime(T1,T0);
  allWeights = [allWeights oldW];
  if iter < 2
  fprintf('---------------------------------- ...
           ----------------------------------------\n');
```

```
  fprintf('Iter# \t   CstFn       eta_k       ENORM       ...
           ERMS       EINTEG       iTime  \n');
  fprintf('------------------------------------ ...
           ----------------------------------------\n');
  end
  ENORM = norm(errVec);ERMS = sqrt(mean(errVec.^2));EINTEG =
trapz(errVec);
  fprintf('   %-3d. %10.5f  %10.5e  %10.5f  %10.5f  %10.5f  ...
              %10.5f\n',iter,Evec(end),rbf.eta,ENORM,ERMS,EINTEG,Tlap);
  allData = [allData;iter,Evec(end),rbf.eta,ENORM,ERMS];
  iter = iter + 1;
end
disp(rbf.w2);

% If we are here means the rbf converged. So, run final simulation
runNeuroCntrlPlant; % script

figure;
subplot(221)
plot(initTraj);grid;
subplot(222)
plot(allData(:,4));grid;
subplot(223)
plot([velProfile trajWd errVec]);grid;
subplot(224)
plot(Kpvec);grid;

cd results
myFig = strcat(fileName,'_finalView.fig');
```

```
saveas(gcf,myFig);
save(fileName,'allWeights','allData','rbf','Evec', ...
     'velProfile','sys','Wd','tf','fd','Y0');
cd ..
=====================================================================
                  M.file,function,"specs"
=====================================================================
function sys = specs

%  given motor specs
sys.Km = 0.05;               %(V.s/rad),motor torque constant
sys.Nr = 50;                 % number of rotor teeth
sys.B  = 5 * 10^(-4);        %(N.M.s^2/rad),viscous friction
sys.J  = 3.6 * 10^(-6);      %(N.m.s^2/rad), Rotor load Inertia
sys.L  = 0.001 ;             %(H), self inductance in each of the phase
windings
sys.R  = 8.4;                %(Ohms),Resistance in each of the phase
windings
sys.Tl = 3*10^-4;

% external torque distrubance
sys.dtdistr = .4;
sys.tdistr = sys.dtdistr;    % torque disturbance
```

```
=========================================================================
                  M.file,function,"rbfInit"
=========================================================================
function rbf = rbfInit(randCase,nc)
global initTraj

% ---- Radial Basis Function Neural Net, intializations ---------- %
rbf.nin      = 1;     % no.of inputs to the RBF neural network
rbf.nhidden  = nc^rbf.nin;
                      % no.of hidden layerd neurons
rbf.nout     = 1;     % no.of outputs from the RBF
rbf.c        = [];    % the centers of the RBF (hidden neurons)
                      % (nhidden,nin)
rbf.wi       = [];    % width (ro), of the Gaussian
                      % (nin,nhidden)
rbf.b2       = zeros(1,rbf.nout);
                      % bias for second layer of the RBF
                      % (1,nout)
rbf.w2       = [];
                      % weight matrix for second layer of the RBF
                      % (nhidden,nout)

rbf.gradient = 0.001; % petrubation introduced in weights to calculate
                      % the partial diff.
rbf.eta      = 0.01;
```

```
% ------------------------------------------------------------------ %
% run nominal control simulation (openloop/PD control)
runNominalCntrl;

% get initial vector
initTraj = trajWd;
errVec = velProfile - trajWd;

if rbf.nin <= 1
   xMax =  2;     %max(e);
   xMin = -2;     %min(e);
   eMax = [eMax xMax];eMin = [eMin xMin];
else
   errdlg('This controller works only for 1-d, currently');
   return;
end

% put centers sparesely between eMin:eMax
Centers = [];Radius = [];
for i = 1:rbf.nin
    if nc <= 1
     Centers(:,i) = (eMax(i) + eMin(i))/2;
     Radius(i,:) = ones(1,rbf.nhidden).*(abs(eMax(i) - eMin(i))/2);
     continue;
    end
     interval = abs(eMax(i) - eMin(i))/(nc-1);
     dInt = [dInt interval];
     Centers(:,i) = (eMin(i):interval:eMax(i))';
     Radius(i,:) = ones(1,nc).*(interval/2);
```

```
 end


if rbf.nin <= 1
   rbf.c = Centers;
   rbf.wi = Radius;
    figure;
    subplot(211)
      plot(e,'r.');hold on;grid on;
    N = length(e);
      index = [];errBars = [];
      index = repmat(1:N,rbf.nhidden,1);
      errBars = repmat(rbf.c,1,N);
      plot(index',errBars');
else
   errdlg('This controller works only for 1-d, currently');
   return;
end


subplot(212)
plot([velProfile trajWd]);grid;
switch randCase
    case 1
         rand('seed',555);
     case 2
         rand('seed',-99);
     case 3
         rand('seed',-100);
     case 4
         rand('seed',777);
```

```
     case 5
         rand('seed',1);
     case 6
         rand('seed',-1);
     otherwise
         rand('seed',0);
 end


rbf.b2       = ones(1,rbf.nout);
rbf.w2       = zeros(rbf.nhidden,rbf.nout);
nw2          = (rbf.nhidden*rbf.nout);
random       = randn(1,nw2)*(10^0);
rbf.w2       = reshape(random,rbf.nhidden,rbf.nout);
rbf.w2       = rbf.w2./nw2;


return;
```

```
====================================================================

                  M.file,function,"rbffwd"

====================================================================

function [a, z, n2] = rbffwd(rbf,inputs)

% inputs = inputs to the rbf neural net   : (ndata,nin)

%   a    = output from the rbf

%   z    = activations from the first layer of rbf

%   n2    = calculated squared norm matrix : (ndata,nhidden)

%   rbf.nin      = no.of inputs to the RBF neural network

%   rbf.nhidden  = no.of hidden layerd neurons

%   rbf.nout     = no.of outputs from the RBF

%   rbf.c        = the centers of the RBF (hidden neurons)

                                          : (nhidden,nin)

%   rbf.wi       = width (ro), of the Gaussian

                                          : (nin,nhidden)

%   rbf.b2       = bias for second layer of the RBF

                                          : (1,nout)

%   rbf.w2       = weight matrix for second layer of the RBF

                                          : (nhidden,nout)


[ndata, dime] = size(inputs);

if dime ~= rbf.nin

    errdlg('In consistent matrix size');

end
```

```
% --------------- RBF = Only for Gaussian -------------------- %
  z = zeros(1,rbf.nhidden);
  for i = 1:rbf.nhidden
      % for each hidden node
      di2 = (rbf.c(i,:) - inputs)';      % Distance vector,
                                         % (nin x 1)
      wi2 = diag(rbf.wi(:,i));           % Co-variance Matrix,
                                         % (nin x nin)
      n2 =  di2' * inv(wi2) * di2 ;      % ~x~ function of x
                             % (1 x [nin x nin] x [nin x nin] x 1)
      z(i) = exp(-0.5 * n2);         % guassian activation  : (1 x 1)
  end
% ------------------------------------------------------------- %

% network outputs                          %(ndata, nout)
a = z*rbf.w2 + ones(ndata, 1)*rbf.b2;
```

```
=====================================================================

                  M.file,function,"extrapolate"

=====================================================================


function [nextY0] = extrapolate(t0,t1,Y0,sys,Va,Vb)
% $$$$$$$$$  Nonlin sim. $$$$$$$$$$$$$$$$$$$$$$$$$$$$$
dt = 1/10^4;                  % sampling frequency 10kHz


% Y0(1) : Displacement (rad)
% Y0(2) : Angular Velocity (rad/sec)
% Y0(3) : Current in winding B (Amps)
% Y0(4) : Current in winding A (Amps)


for t = t0+dt:dt:t1
% simplfy calc.
    s = sin(sys.Nr*Y0(1));
    c = cos(sys.Nr*Y0(1));
 % non linear system eqns %+ ThetaNoise*randn
    dy = [Y0(2);
          (- Y0(4) * s * sys.Km + Y0(3)* c * sys.Km . . .
          - sys.B*Y0(2) - sys.Tl)/sys.J;
          (-sys.R*Y0(3) - sys.Km* c * Y0(2) + Vb)/sys.L;
          (-sys.R*Y0(4) + sys.Km* s * Y0(2) + Va)/sys.L];
% use Euler inegration
    Y0 = Y0 + dy*dt;
```

```
    Y0(1) = mod(Y0(1),2*pi);
end
nextY0 = Y0;

========================================================================
                  M.file,script,"runNominalCntrl"
========================================================================
% script runs PMS model given by the m-file specs.m
% select open loop //PD control
type = 'P';                      % use 'open' alternatively
% PD with noise (Mag)
Kp = .63;
Kd = 1.8*10^-4;
iProf = 1;
% ----------------- Initialize system parameters ------------------- %
Wd = [5];                         % desired angular velocity
sys = specs;                      % get system specs
sys.Tl = abs(sys.Tl);             % system Torque intializations
sys.tdistr = sys.dtdistr;         % torque disturbance
% system timing paramters and noise
tf = 1.9;                        % final time
Y0 = [-.0183;Wd(1);0;0.119];     % sys. initial conditions
fd = 10^3;                       % control frequncy
dtCntrl = 1/fd;                  % desired freq. rad/sec
trajWd = [];Yvec = [];Kpvec =[];Tvec =[];

% ---------------- Construct Profile vector ------------------------ %
velProfile = Wd*ones(size(0:dtCntrl:tf-dtCntrl));
Wt = sys.Nr*Wd(1);Wtvec = [];
```

```
% cntrl @ t = t0
Mag = 1;Va = Mag*cos(Wt*0);Vb = Mag*sin(Wt*0);



% misc. intializations
enew = 0;eold = enew;compen = 0;
Measurement = 0;MeasNoise = 0.01;
Controls = []; velMeasure = [];
rand('seed',0);

for tt = 0:dtCntrl:tf-dtCntrl
% ------------  get the Measurement  ------------------- %%
% calculate the error from Refernce and Measurement
  Measurement = Y0(2) + (MeasNoise*rand);
  enew = velProfile(iProf) - Measurement;
  Kpvec = [Kpvec;compen];


  Yvec = [Yvec;Y0'];
  velMeasure = [velMeasure; Y0(2)-Measurement];
  Wtvec = [Wtvec;Wt];
  Controls = [Controls; Va Vb];
  trajWd = [trajWd;Y0(2)];

% for every 'tdist', GENERATE Ext. torque disturbance
  if tt  >=  sys.tdistr
     % change
     sys.tdistr = (round(tt./sys.dtdistr)*sys.dtdistr) + sys.dtdistr;
     sys.Tl = -sys.Tl;
  end
```

```
% ------------- Extrapolate system ---------------------- %%
% time,current states,control,system parameters : gives interated error
  [Y0] = extrapolate(tt,tt+dtCntrl,Y0,sys,Va,Vb);

% Get Control for next time step from MEASUREMENTS @ tt
switch type
      case 'open'
           % ----------- openloop ---------------------- %
           Wt = sys.Nr * Wd(1);
           Ks = .8;
           Mag =  Ks;
       case 'P'
            % ----------- p control ---------------------- %
            Wt = sys.Nr * velProfile(iProf);
                Ks = .8;
            Mag = Kp*enew + (Kd *(enew-eold)/dtCntrl)+ Ks;
    end % // end case

  % determined controls for next time step
  Va = Mag*cos(Wt*(tt+dtCntrl));
  Vb = Mag*sin(Wt*(tt+dtCntrl));

  eold = enew;
  Tvec = [Tvec;tt];
  iProf = iProf + 1;
end

errVec = velProfile - trajWd;
```

```
ENORM = norm(errVec);ERMS = sqrt(mean(errVec.^2));EINTEG =
trapz(errVec);

% plot trajectory tracking
figure;
plot(Tvec,[velProfile trajWd])%,'-r-.');
title(['RmsErr = ' num2str(ERMS) '; Enorm = ' num2str(ENORM) ';
IntegralErr = ' num2str(EINTEG)]);
grid;legend('velProfile','trajectory',0);
xlabel('Time (secs)');
ylabel('Ang. velocity,w(rad/sec)');

% plot controls
figure;
subplot(211)
plot(Tvec,Controls(:,1));
grid;title('Control input ua');

subplot(212)
plot(Tvec,Controls(:,2));
grid;title('Control input ub');

% plot states
figure;
subplot(221)
plot(Tvec,Yvec(:,1));grid;
title('Displacement (rad)');
xlabel('Time (secs)');
```

```
subplot(222)
plot(Tvec,Yvec(:,2));grid;
title('Ang. Velocity (rad./sec)');
xlabel('Time (secs)');

subplot(223)
plot(Tvec,Yvec(:,3));grid;
title('Current windingB (A)');
xlabel('Time (secs)');

subplot(224)
plot(Tvec,Yvec(:,4));grid;
title('Current in WindingA (A)');
xlabel('Time (secs)');

% clear variables
clear enew iProf Wt dWt maxWd deadBand dRPM Wd sys Y0 tt tf fd dtCntrl

======================================================================
                  M.file,script,"runNeuroCntrlPlant"
======================================================================
% ------------------ Initialize system parameters --------------------
% system Torque intializations
  sys.tdistr = sys.dtdistr;      % torque disturbance
  sys.Tl     = abs(sys.Tl);
  Y0 = [-.0183;Wd(1);0;0.119];   % intial state vector
  trajWd = [];Yvec = [];Kpvec =[];Tvec =[];errVec = [];
  Wt = sys.Nr*Wd(1);Wtvec = [];
```

```
% cntrl @ t = t0
Mag = 1;Va = Mag*cos(Wt*0);Vb = Mag*sin(Wt*0);




% misc. intializations
enew = 0;eold = enew;compen = 0;
Measurement = 0;MeasNoise = 0.01;
Mag = 1;                                % rbf.b2 : just the bias *****

% cntrl @ t = t0
Va = Mag*cos(Wt*0);Vb = Mag*sin(Wt*0);
enew = 0;eold = enew;compen = 0;

% misc. initializations %
eBuff = zeros(1,rbf.nin+1);
Measurement = 0;MeasNoise = 0.01;
rand('seed',0);

iProf = 1;

% RBF control (closed loop), vel tracking
for tt = 0:dtCntrl:tf-dtCntrl
% ------------  get the Measurement  ------------------- %%
  % calculate the error from Refernce and Measurement

  Measurement = Y0(2) + (MeasNoise*rand);
  enew = velProfile(iProf) - Measurement;
```

```
Yvec = [Yvec;Y0'];
Wtvec = [Wtvec;Wt];
Magvec = [Magvec;Mag];
trajWd = [trajWd;Y0(2)];

%  --- FIFO buffer ---%
eBuff = [eBuff(2:end) enew];

% for every 'tdist', GENERATE Ext. torque disturbance
if tt  >=  sys.tdistr
   % change
   sys.tdistr = (round(tt./sys.dtdistr)*sys.dtdistr) + sys.dtdistr;
   sys.Tl = -sys.Tl;
end

% ------------- Extrapolate system ---------------------- %%
% time,current states,control,system parameters
  [Y0] = extrapolate(tt,tt+dtCntrl,Y0,sys,Va,Vb);

% Get Control for next time step from MEASUREMENTS @ tt
% wait 'rbf.nin' control steps until buffer fills
if (round(tt/dtCntrl) >= (rbf.nin+1))
   a = rbffwd(rbf,eBuff(1:end-1));
                  % Tracking error is input to the RBF
   Mag = a;       % RBF output directly used as control magnitude
end
Wt = sys.Nr*velProfile(iProf);   % standard OpenLoop Velocity
Va = Mag*cos(Wt*(tt+dtCntrl));
Vb = Mag*sin(Wt*(tt+dtCntrl));
```

```
  eold = enew;
  Tvec = [Tvec;tt];
  iProf = iProf + 1;
end
```