PARAMETER ESTIMATION USING NEURAL NETWORK

P. 345

PROJECT REPORT 1998-99

Submitted by

J. OM PRAKASH
D. KARTHIKEYAN
R. RAMAR
P. SATHISH KUMAR

Under the Guidance of

Mrs. N. KALAIARASI B.E.,

Submitted in partial fulfilment of the requirements for the award of the Degree of

BACHELOR OF ENGINEERING IN ELECTRICAL AND ELECTRONICS ENGINEERING

Branch of the Bharathiar University, Coimbatore

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE - 641 006

Dedicated to our
beloved Parents

ACKNOWLEDGMENT

We express our heart felt gratitude to our guide **Mrs. N. Kalaiarasi, B.E .,** Lecturer in Electrical and Electronics Engineering Department helping us in all possible ways, right from procuring materials to writing the project report . we also record our deep gratitude for her valuable guidance and constructive criticism given throughout the Project work.

Our heartiest thanks are due to our beloved professor **Dr. K.A. Palanisamy, B.E., M.Sc.(Engg), Ph.D., MISTE, C. Eng (I), FIE,** Head of the Department of Electrical and Electronics Engineering for his constant encouragement.

We sincerely thank our respected Principal **Dr. K.K. Padmanabhan, B.Sc. (Engg), M.Tech, Ph.D.,** and the management for their patronage and facilities which were made available for our project.

We are also thankful to all teaching and non- teaching staffs of Electrical and Electronics Engineering Department for their kind help and encouragement in making our project successful.

Last but not least, we extend our sincere thanks to all our friends who have contributed their ideas and encouraged us for completing the project successfully.

SYNOPSIS

Parameter estimation is defined as the determination of unknown parameters in a mathematical model of a physical system from a knowledge of input and output data of the system. One of the major problems in modern control theory is the estimation of unknown structural parameters contained in a mathematical model of a system using the measured input and output data. This has led to the parameter estimation problem.

A dynamically system that can be represented in terms of an ordinary differential equation will be called lumped parameter system. When it requires the use of partial differential equation to describe its dynamically behavior, it will be called distributed parameter system.

Artificial Neural network can achieve high computational rates by employing a massive number of simple processing elements. Neural Networks with feed back connections provide a computing model capable of solving a rich class of optimization problems. Here the technique adopted to estimate the parameters of different systems is Hopfield neural network and perception network.

In this project a Hopfield neural square estimator is constructed to estimate the parameters. Here we use neural network as a new scheme to solve the parameter estimation problems.

The main advantage of the scheme is the tremendous speed which only depends on the time constant of the network and is not related to the scale of the problem.

The parameter estimation problem is also solved using a perception least mean square estimator. A perception LMS estimator is simulated and tested.

The project deals with the problem of parameter estimation in linear lumped and distributed parameter systems, using neural networks. The approach followed starts by eliminating all derivatives in the mathematical model of the system by successive integration i.e., it converts the differential equation into integral one. This successive integration procedure introduces the unknown initial and boundary conditions explicitly in the mathematical model. These conditions are assumed in the form of truncated laguerre series whose coefficients are to be determined.

The main advantage of the scheme is the tremendous speed which only depends on the time constant of the network and is not related to the scale of the problem.

The parameter estimation problem is also solved using a perception least mean square estimator. A perception LMS estimator is simulated and tested.

The project deals with the problem of parameter estimation in linear lumped and distributed parameter systems, using neural networks. The approach followed starts by eliminating all derivatives in the mathematical model of the system by successive integration i.e., it converts the differential equation into integral one. This successive integration procedure introduces the unknown initial and boundary conditions explicitly in the mathematical model. These conditions are assumed in the form of truncated laguerre series whose coefficients are to be determined.

CONTENTS

Chapter 1

Introduction

CHAPTER -1

INTRODUCTION

Parameter estimation is defined as the determination of unknown parameters in a mathematical model of a physical system from a knowledge of input-output data of the system, such that over a desired range of operating conditions the model output are close, in some well defined sense , to the output of the physical system, when both are subjected to the same inputs.

In this project Hopfield least square estimator is constructed to estimate the parameters. A Hopfield Neural network approach is based on the modified Hopfield energy function. Generally speaking, there are three numerical techniques that can be employed for least square estimation. They are generalized matrix inversion, singular value decomposition and QR decomposition. Although the performances and complexities of these techniques are different, they are computationally intensive and difficult for real time processing. Here we use neural network as a new scheme to solve the parameter estimation problems. The main advantage of the scheme is the tremendous speed which only depends on the time constants of the network and is not related to the scale of the problem.

The parameter estimation problem is also solved using a perceptron LMS estimator. A perception LMS estimator is simulated and tested.

The project deals with the problem of parameter estimation in linear lumped and linear distributed parameter systems using neural networks. The approach followed starts by eliminating all derivatives in the mathematical model of the system by successive integration i.e., it converts the differential equation into an integral one. This successive integration procedure introduces the unknown initial boundary conditions explicitly in the mathematical model. These conditions are assumed in the form of truncated laguerre series will then yield an estimate of the initial and boundary conditions. This way the present identification problem reduces to that of determining the coefficients of the aforementioned truncated laguerre in truncated laguerre series and are introduced into the integral equation where upon equating coefficients of like laguerre polynomials, a linear system of equation in both the unknown model parameters and unknown laguerre coefficients of the initial and boundary conditions is derived.

1.1 INTRODUCTION TO NEURAL NETWORK

1.1.1 HISTORY OF ARTIFICIAL NEURAL NETWORKS

The progress of neurobiology has allowed researchers to build mathematical models of neurons to simulate neural behavior. The idea dates back to the early of 1940 's when one of the first abstract models of a neuron was introduced by Mc Culloch and Pitts (1943). Hebb (1949) proposed a learning law that explained how a network of neurons learned. Other researchers

pursued this notion through the next two decades, such as minsky (1954) and Rosenblatt (1958). Rosenblatt is credited with the perceptron learning algorithm . at about the same time, Windrow and Hoff developed an important variation of perceptron learning, known as the Widrow - Hoff rule.

Later, Minsky and Pafert (1969) pointed out theoretical limitations of single - layer neural network models in their landmark book perceptrons. Due to this pessimistic projection, research on artificial neural networks lapsed into an eclipse for nearly two decades. Despite the negative atmosphere, some researchers still continued their research and produced meaningful results. For example, Anderson (1977) and Grossberg (1980) did important work on psychological models. Kohonen(1977) developed associative memory models.

In the early 1980's, the neural network approach was resurrected. Hopfield (1982) introduced the idea of energy minimization in physics into neural networks. His influential paper enclosed this technology with renewed momentum. Feldman and Ballard(1982) made the term "connectinist" popular. Sometimes, connectionism is also referred to as subsymbolic processes, which have become the study of cognitive and AI systems inspired by neural networks. Unlike symbolic AI, connectionism emphasizes the capacity of learning and discovering representations. Insidiously, connectionism has become a common ground traditional AI and neural network research.

In the middle 1980's, the book parallel Distributed Processing by Rumelhart and MeClelland (1986) generated great impacts on computer

cognitive and biological sciences. Notably , the backpropagation learning algorithm developed by Rumelhart, Hinton, and Williams (1986) offers a powerful solution to training a multi layer neural network and shattered the curse imposed on perceptrons. A spectacular success of this approach is demonstrated by the NET talk system developed by Sejonwski and Rosenberg (1987), a system that converts English text into highly intelligible speech. It is interesting to note, however, that the idea of backpropagation had been developed by Werbos (1974) and Parker (1982) independently.

Although the neural network approach rejects the notion of separating knowledge from the inference mechanism, it does not reject the importance of knowledge in many tasks that require intelligence. It just uses a different way to store and manipulate knowledge.

The symbolic approach which has long dominated the field of AI was recently challenged by neural network approach. There have been speculations about whether one approach should coexist and combine. More evidence favors the integration alternative in which the low-level pattern recognition capability offered by the neural network approach and the high-level cognitive reasoning ability provided by the symbolic approach compliment each other. The optimal architecture of future intelligent systems may well involve their integration in one way or another.

kinds of knowledge in building a knowledge - based system : deep knowledge and surface knowledge.

Surface knowledge is the heuristic, experiential knowledge learned after solving a large number of problems. It is the knowledge that human experts often rely on. It usually offers a quick, satisfactory solution, which is not necessarily the best though. The main problem with surface knowledge is its in adequacy in dealing with novel situations.

Deep knowledge refers to the basic laws of nature and the fundamental structural and behavioral principles of the domain. Invocation of deep knowledge for problem solving is sometimes called reasoning from first principles. In comparison with surface knowledge, deep knowledge has a stronger formal basis. It allows a derivation of a solution even for a novel situation, but the process may be time-consuming. One way to make it more efficient for use to compile it. However, compiled deep knowledge may not correspond to surface knowledge since they come from different sources. In addition, there is no guarantee that every piece of surface knowledge can be proven based on deep knowledge. What is important in practice is how the two kinds of knowledge can be integrated so as to optimize the system performance.

In some knowledge - based systems, we make distinctions between metalevel and object-level knowledge. Object-level knowledge is the knowledge for solving the problem in the defined domain. Metalevel knowledge is the knowledge which controls the use of object-level knowledge. The employment of

metalevel knowledge is intended to provide a better control of object-level knowledge. However, metalevel knowledge is not the same as the control knowledge housed in the inference engine. As a matter of fact, metalevel knowledge is also controlled by the inference engine. In a metalevel reasoning system, metalevel knowledge is invoked first, which then selects appropriate object-level knowledge to make inference.

The inference engine governs the use of the knowledge stored in the knowledge - base. While the design of the inference engine is full of variety, we identify a general knowledge - based algorithm as follows :

1. The inference engine selects a piece of knowledge from the knowledge - base.

2. The inference engine executes the selected knowledge either to transform the goal or to generate a new fact.

3. If the goal is solved, then exit and succeed. If a certain stopping condition is met such as the case when the knowledge available is exhausted but the goal is not soled yet, then exit and fail. Otherwise go to step 1.

1.1.3. NEURAL INFORMATION PROCESSING

Biological neurons transmit electrochemical signals over natural pathways. Each neuron receives signals from other neurons through special junctions called synapses. Some inputs tend to excite the neuron, others tend to inhibit it. When the cumulative effect exceeds a threshold, the neuron fires and sends a signal down to other neurons. An artificial neuron models these simple

biological characteristics. Each artificial neuron receives a set of inputs. Each input is multiplied by a weight analogues to a synaptic strength. The sum of all weighted inputs determines the degree of firing called the activation level. Notatially, each input X_i is modulated by a weight W_i and the total input is expressed as

$$\Sigma X_i W_i$$

or in vector form , X. W where $X = [X_1, X_2, X_3, \ldots\ldots, X_n]$ and $W = [W_1, W_2, \ldots W_n]$ The input signal is further processed by an activation function to produce the output signal which, if not zero, is transmitted along. The activation function can be a threshold function or a smooth function like a sigmoid or a hyperbolic tangent function.

A neural network is represented by a set of nodes and arrows, which is a fundamental concept in graph theory. A node cores ponds to a neuron, and an arrow corresponds to a connection along with the direction of signal flow between neurons.

The dynamic behavior of the neural network is described by either differential equations or difference equations. The former representation assumes continuous time and can be used to simulate the network on an analog computer, whereas the latter uses discrete time and is usually taken to simulate the network on a digital computer.

Chapter 2

Hopfield Network

CHAPTER II

HOPFIELD NETWORK

2.1 INTRODUCTION :

Hopfield and tank have shown that several optimization problem can be solved rapidly by Hopfield networks which are recursive networks of simple neuron like analog processors. With Hopfield Hopfield networks , the arguments of an object function is converged to a vector of a hypercube. Therefore their applications are restricted to decision optimization problem. Here we will deal with problems in which the arguments of objective functions are real numbers. Neural network that solves least square estimation problem is derived with this network, the objective function can converge to any inner point of a hypercube, giving a real valued solution with a tremendous speed. Because of the convex, nature of the chosen energy function, the problem of convergence to a local minimum does not arise. A space iterative search technique is introduced so as to find the optimum solution that can exist at any point within the space.

Hopfield Neural network have surprising powerful computational properties. Hopfield and Tank have shown that these networks are able to solve severed optimization problems, such as travelling salesmen problem which is NP-Complete Problem, A/D Conversion, Signal decision, and so on.

Basically a Hopfield network perform a kind of constrained least square (LS) or Quadratic programming operation employing a gradient procedure, such

that the optimization results is conveyed to a vertex of hypercube. The optimization results are in binary form.

In signal and image processing one of the most important computation is LS estimation. Such as linear prediction, parameter estimation and deconvolution. The estimated coefficients of an LS problem are usually required to take on real value rather than binary.

In this project we used the continuous work mode of Hopfield network , which offers the advantage of fast convergence and simplicity of system configuration. The energy function is chosen as one half of the error squares of the problem. By introducing a space iterative search technique the estimated coefficient are shown to lie at anywhere within the space technique, resulting in a greatly extended dynamic range. Since the convergence time of neural network doesn't depend on the number of neurons in the network, this is very suitable for solving large scale problems whenever high speed, such as real time processing, is one of the requirements.

2.2 The Neural LS Estimator

the LS problem is to find a vector V minimizing the Euclidean length of Av-b where $A \in R^{m \times n}$ and $b \in R^{m}$ are known , $V \in R^{n}$. in other word, it is required to find V such that ll AV - bll is minimized. The energy function for the neural network is defined as

$$E(v) = 1/2 \|AV-b\|$$

$$= 1/2 \left(\sum_{I=1}^{n} Vi\,Ai = b \right)^{T} \left(\sum_{i=1}^{n} Vi\,Ai - b \right)$$

where Vi is the i^{th} element of V, Ai is the i^{th} column of matrix A. The n elements of vector V are associated with the states of neurons in the network. When the energy function reaches minimum the state of the neurons corresponds to the LS solution.

Here we used the continuous work mode of the neural networks. Equation (1) is used as the energy function and from this energy function a network is constructed. Whose stable point corresponding to the minimum of the function E(V). The Hopfield network is inherently gradient descent. Therefore the network is derived from the gradient descent point of Y le^{0}ue; that is

$$du/dt = - \Delta E(V) \qquad (2.2)$$

$$V_i = g_i(U_i) \quad I = 1,2,3,4.....n \qquad (2.3)$$

where U_i is the summation of all the inputs to the i^{th} neuron in the network , V_i is the state (output) of i^{th} neuron g_i is the nonlinear activation function of the i^{th} neuron which is an output bounded function. Thus V is restricted in a bounded region which is usually a hypercube in an n-dimensional space from (2.1) and (2.2), we get the network as

$$du / dt = - (A^{T}A) V + A^{T} \qquad (2.4)$$

or more specifically,

$$dui/dt = \sum_{j=1}^{n} Tij\, Vj + Ij \qquad i = 1.2,.....n \qquad (2.5)$$

where Tij is the synaptic weight from j^{th} neuron to i^{th} neuron

I_j is the external input to the i^{th} neuron and by they are given by

$$Tij = - Ai^T Aj = - \sum_{k=1}^{n} a_{ki}\, a_k \qquad (2.6)$$

$$I_i = - A_i\, b = - \sum_{k=1}^{n} a_{ki}\, b_k \qquad (2.7)$$

The connection matrix Tij is symmetric.

The chance of the energy function E as the network evolves with the time is given by

$$dE / dt = \sum_{k=1}^{n} \frac{\partial E}{\partial V_k}\, dvk/dt$$

$$= \sum_{k=1}^{n} (-dUn / dt)\ dvk/dt$$

$$= \sum_{k=1}^{n} gk^{1}(Uk)\,(dvk/dt)^2 \qquad (2.8)$$

Thus the sufficient condition for $dE/dt \leq$ is

$$g_i^{(u)} \geq 0 \qquad i = 0,2....n \qquad (2.9)$$

The second derivative of the energy function as given by

$$\Delta^2 E(v) = A^T \qquad (2.10)$$

is a non-negative definite matrix ; thus E(v) is a convex function on any convex subset of N- dimensional Euclidean space.

2.3 SPACE ITERATIVE SEARCH FOR OPTIMUM STATE

If we consider using hyperbolic tangent as the activation function for the neurons such that $g_i(u_i) = b \tanh(\lambda w_i)$ (2.11)

Where b and λ are the amplitude and steepness of the function respectively. A subset of R^n is defined and is given by

$D^n = \{ V \in R^n : -b < v_i < b , i = 1,2,3.....,n \}$

If the network is initialized at an interior point of D^n , the state space flow is always within D^n. If E(v) has its minimum in D^n , then the networks described in the previous section is readily applicable. Otherwise, network will finally settle down at a point very close the boundary of D^n , which may not be the optimum solution. This problem can be resolved as follows.

Suppose that the desired solution V lies near a point c and within D^n $\{V \in R^n : -b < v_i C_i < b, I = 1,2,3.....n \}$ we can shift the origin of the state space to C such that

$V_c = V - c$ (2.12)

Where V_c and v are respectively , the vector in new and old coordinate system. Substituting the shifted vector into (1) yields

$E(V_c) = 1/2 \| Avc - bc \|^2$ (2.13)

where bc = b-Ac comparing (2.13) with (2.1) and using (2.6) (2.7) , it is found that the network configuration and synaptic weight remains the same. The only change is external input values to the network as given by

$$I_{ci} = A_i b_c = \sum_{k=1}^{m} a_{ki} b_{ck} = \sum_{k=1}^{m} a_{ki} (b_k - \sum_{j=1}^{n} a_{kj} c_j) \qquad (2.14)$$

The state of the network will converge to v_c when the function E reaches its minimum. Then the final solution for Ls problem is

$$V^* = V_c^* + C$$

unfortunately, in most cases , we do not have a prior knowledge of the location of the LS solution i.e., we do not know as where the coordinate system ought to be shifted. For such general cases, we use a space iterative search technique to find the LS solution in space R^n.

The basic procedure is that the system starts search at a point in D^n , and settle down at C_1 If C_1 is not near the boundary of D^n then it can be considered as the minimum of E and we have achieved the solution. If C_1 is very close to the boundary of D^n , the minimum E may even be outside D^n. In this case , we shift the origin of the state space to C_1 by changing extended inputs to the network according to (14) and state the search again. This time let the network converge at point C_2 in the new coordinate system. If C_2 is not at or very close to the boundary of D^n. $C_1 + C_2$ is the solution. If it is not so , shift the coordinate system to point C_2 and continue to search again. Suppose that after i^{th} the shift of the coordinate system, the network settles down at point

C_{i+1} which is inside D^n

Then the final LS solution is

$$Y = C_1 + C_2 + \ldots\ldots + C_{2+1} \qquad (2.15)$$

CHAPTER III

PERCEPTRON

3.1 INTRODUCTION

The perceptron, first introduced in the late 1950's by Frank Rosenblatt at Cornell University, is a two layer feedforward network of threshold logic units.

The perceptron was among the first and the simplest learning machines that are trainable. The mode of training is supervisory, because the steps in algorithm involves the comparison of actual outputs with desired outputs associated with the set of training patterns. Supervisory training applies as well to Least Mean Square (LMS) algorithm. The LMS algorithm was developed for adaptive systems research geared towards diverse applications, including noise cancellation and adaptive equalization. Accompanying the structural simplicity of perceptron is a powerful convergence theorem.

In general, a perceptron is not a single TLU, because it is also has a layer of fixed processing units. Perceptron denotes the class of two-layer feed forward networks.

a) Whose first-layer units have fixed functions and with fixed connections weights from the inputs and

b) Whose connecting weights linking this first layer to the second layer of the outputs are learnable together with the thresholds of the units in that outputlayer.

Fig 3.1 gives a two layer perceptron network with 4 neurons in the input layer and three neurons in the output layer.

GENERAL PERCEPTRON LEARNING ALGORITHM

i) Select random weights between input and output layer as W_{ij}

ii) Specify a random input vector e. Let the selected vector be briefly described as

$e = (e_1, e_2, \ldots\ldots e_n)$ where $e_k = 1$ or 0.

iii) Change the weights through

$w_{ij}^{new} \quad w_{ij}^{old} + \triangle w$

with $\triangle w_{ij} = \alpha \cdot e_j \varepsilon_i$

Here $\varepsilon_i = a_i - f(\varepsilon_k \; w_{ik} e_k) = a_i - f(net_i)$ is the difference between the end output and the actual output at the place of I, and $\alpha > 0$ is a small number.

(iv) Continue with step (ii)

Here e_i the input and a_i the output

the algorithm ends, when the network produces the correct final vector for all input vectors.

3.2 WIDROW-HOFF LMS ALGORITHM FOR TRAINING THE PERCEPTRON

The widrow - Hoff or LMS algorithm uses a linear rule for training a perceptron. Widrow called a trainable TLU, allowing both analog and real value inputs, an Adaline. Let d (k) be the desired real - valued output for the

augumented pattern y (k) , and suppose that the corresponding weight vector is w (k). The error at k^{th} iteration is

$$e_k = d(k) - w(k) \cdot y(k)$$

The LMS algorithm is a gradient descent algorithm requiring the squared error function.

$$e^2_k = (d(k) - W(k)\ y\ (k))^2 = (d(k) - w^T(k)\ y(k))$$

to be minimized at the k^{th} iteration. Y (k) and w (k) are random vector, so that e_k^2 is not the ensemble mean squared error, $E(e_k^2)$, where E (.) denotes the expectation operator.

The difference between LMS algorithm and the perceptron algorithm is that the error minimized by the LMS algorithm is a continuous quantity. Both can be applied to the same Neural network architecture. Here e_k^2 is the squared error from one pattern.

Then $\hat{\Delta} k = \delta e_k^2 / \delta w = -2 e_k y_k$

Where $\hat{\Delta} k$ is an approximation of the true gradient Δ_k of the ensemble mean squared error. If we restrict the parameter vector w (k) to be deterministic, it is routine to verify that

$$\Delta_k = \delta E(e_k^2) / \delta w(k) = 2(Q\, w(k) - P)$$

Where $Q = E(y(k)\ y^T(k))$ is the $N \times N$ auto correlation matrix of all the N training pattern $\{ y(k)_{i=1} \}^N$ and $P = E(d(k)\ y(k))$ is a vector of

cross correlation's. Applying the gradient decent rule we find that using ∇

yields $w(k+1) = W(k) + 2 \alpha e_k y(k)$.

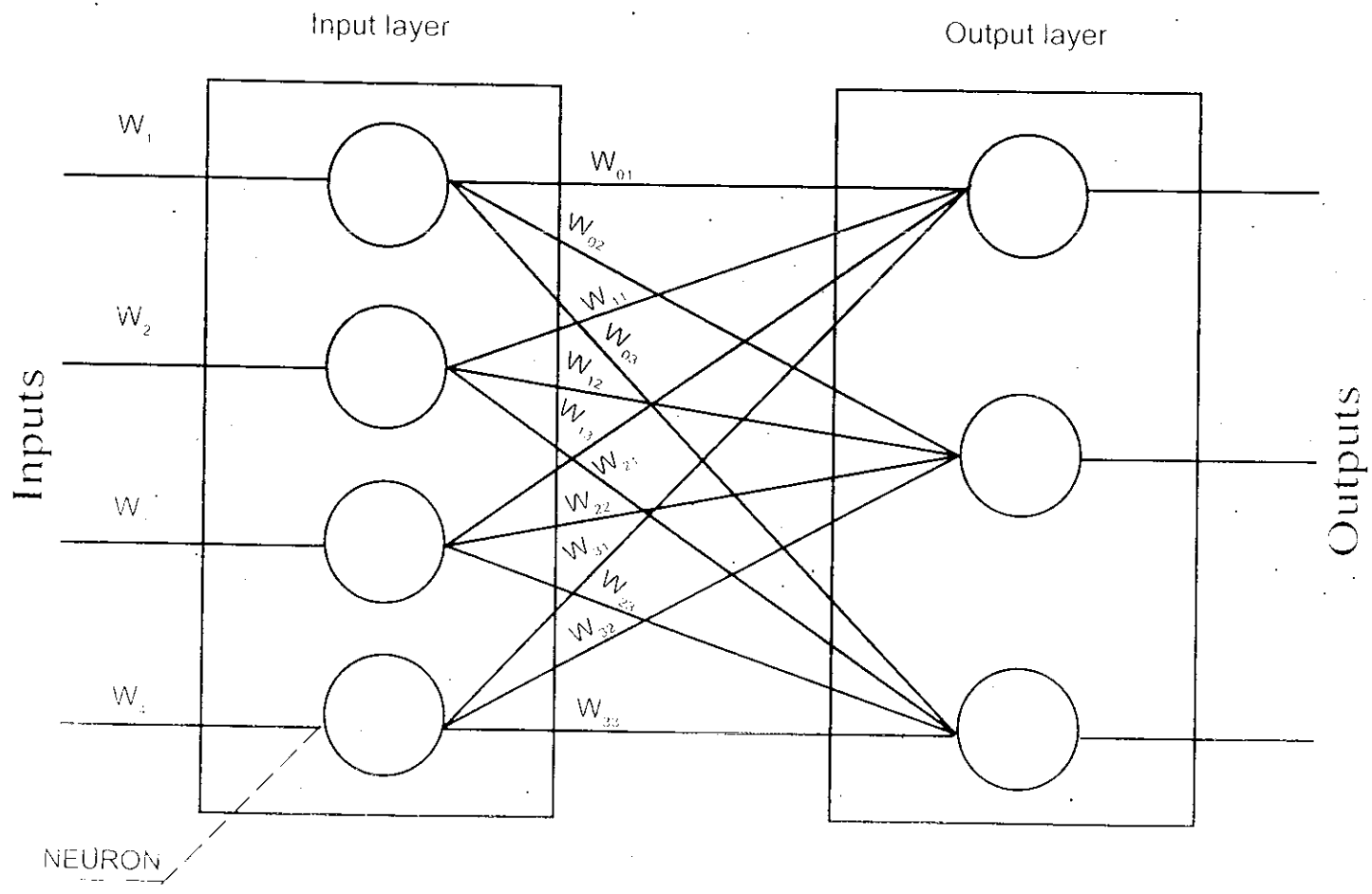

Fig. 3.1 GENERAL TWO LAYER PERCEPTRON NETWORK

CHAPTER IV

4.1 INTRODUCTION TO LAGUERRE POLYNOMIAL

4.1.1 INTRODUCTION

Orthogonal functions are used in the field of signal representation and approximation, system identification.etc. To do the above tasks, it is necessary to represent a time function, by the superposition of members of a set of simple functions. Only orthogonal sets of functions can be made to synthesize completely any time function to a required degree of accuracy.

Further, the characteristics of an orthogonal set are such that the determination of a particular member of the set combined in a given time function can be made using quite simple mathematical operations. A set of functions whether orthogonal and orthonormal or not said to be complete or closed. if no function exist which is orthogonal to every other function, of the set unless the integral of the square of the function is itself zero.

We summarize below some of the important properties of orthogonal functions which are relevant to Laguerre Polynomial.

i) A complete set of orthonormal functions $f_i(t)$ over the interval t lies between a and b, can be used to express a time function x (t), which is square integrable in t lies between a and b.

$$\text{i.e. } x(t) = \sum_{i=0} c_i f_i(t) \qquad (4.1)$$

Where $C_i = \int_a^b x(t)\, f_i(t)\, dt$ (4.2)

ii) The approximation of the signal by a finite number of orthonormal function in the minimum mean square error sense leads to the same coefficients as given by equation 4.2.

iii) The notion of orthogonality and orthonormality can be extended by introducing a weighting function w (t). This offers the possibility of emphasizing the contributions of the mean.square error in a predetermined way.

4.1.2 DEFINITION OF LAGUERRE POLYNOMIAL

The laguerre polynomial is defined as

$$\lambda_i(t) = \exp(t)\frac{d^i}{dt^i}\{t^i \exp(-t)\} \quad (4.3)$$

Thus,

$\lambda_0(t) = 1$
$\lambda_1(t) = 1-t$

$$(l+1)\lambda_{l+1}(t) = (1+2l-t)\lambda_l(t) - l\lambda_{l-1}(t) \quad (4.4)$$

The above polynomials are orthonormal in t lies between 0 and ∞ with a weighting function exp (-t)

$$\int_0^\infty \exp(-t)\lambda_l(t)\lambda_j(t)\,dt = \{1 \text{ if } l = j, 0 \text{ if } l \neq j\} \quad (4.5)$$

Any function x (t) which is square integrable in t.lying between 0 and ∞ may be represented approximately by a finite Laguerre series.

$$x(t) = \sum_{l=0}^{N-1} c_l \lambda_l(t) = \lambda(t)\, C \qquad (4.6)$$

Where C and $\lambda(t)$ are Laguerre coefficient vector and laguerre vector respectively.

$$C = [\, c_0 \;\; c_1 \;\; \ldots\ldots\ldots\ldots \;\; c_{N-1} \,]^T \qquad (4.7)$$

Where T stands for transposition

$$\lambda(t) = [\lambda_0(t), \lambda_1(t), \ldots\ldots \lambda_{N-1}(t)] \qquad (4.8)$$

The laguerre coefficients

$$C_l = \int_0^{\infty} \exp(-t)\, x(t)\, \lambda_j(t)\, dt \qquad (4.9)$$

are obtained by the minimization of integral square error.

$$\varepsilon = \int_0^{\infty} \exp(-t)\, [\, x(t) - \lambda(t)\, c \,]^2\, dt \qquad (4.10)$$

When we multiply both sides of equation 4.6 by exp (-t) we get

$$\exp(-t)\; x(t) = \sum_{l=0}^{N-1} c_l \exp(-t)\, \lambda_l(t) \qquad (4.11)$$

Where { exp (-t) $\lambda_l(t)$ } is the set of orthonormal function in t in the interval 0 and ∞ called laguerre function. Here we may use the property (iii) and the multiplication of x (t) by exp (-t) gives more weightage to the initial transient signal than to the steady state signal. At the same time coefficient C are same

as that for laguerre series. Hence we can still make use of the operational properties of Laguerre polynomials.

Differentiating 4.3 leads to a differential form

$$\frac{d}{dt}\lambda_i(t) = \frac{d}{dt}\lambda_{i-1}(t) - \lambda_{i-1}(t) \; ; \; i = 1, 2, \ldots \qquad (4.12)$$

Integration of the equation 4.12 we get

$$\int_0^t \lambda_i(t')\,dt' = \lambda_i(t) - \lambda_{i+1}(t) \qquad (4.13)$$

In the matrix form equation 4.13 can be written as

$$\int_0^t \lambda_i(t')\,dt' = \begin{bmatrix} 1 & -1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & -1 & \ldots & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & -1 \\ 0 & 0 & 0 & \ldots & 0 & 1 \end{bmatrix} \begin{bmatrix} \lambda_0(t) \\ \lambda_1(t) \\ \ldots \\ \ldots \\ \lambda_{N-2}(t) \\ \lambda_{N-1}(t) \end{bmatrix}$$

$$\int_0^t \lambda_i(t')\,dt' = P_N\,\lambda_N(t) \qquad (4.14)$$

Where $\lambda_N(t)$ has been truncated in the integration of $\lambda_{N-1}(t)$ and the subscript in the parentheses denotes the order of the matrix. P_N is called the Laguerre operational matrix, which is upper bidiagonal.

CHAPTER V

LINEAR LUMPED SYSTEM

5.1 LEAST SQUARE ESTIMATION OF COEFFICIENTS OF LAGUERRE POLYNOMIAL - [SINGLE VARIABLE CASE]

For the simulation of laguerre coefficients on a computer we present a recursive algorithm below for evaluating the coefficient of laguerre polynomial.

From the equation 4.11, we get

$$\exp(-k\Delta t)\ x(k\Delta t) = \sum_{i=0}^{N-1} c_i \exp(-k\Delta t)\, \lambda_i(k\Delta t) \qquad 5.1$$

where k = 1,2,3, ...
Δt = sampling interval

by discretizing equation 5.1 we get,

$$X_K = A_k\ C_k \qquad 5.2$$

Where

$$X_K = [\exp(-\Delta t)\, X(\Delta t)\ \exp(-2\Delta t)\ x\ (2\Delta t) \ldots\ \exp(-k\Delta t)\, x\, (k\Delta t)]^T \qquad 5.3$$

$$C_k = [c_0\ c_1 \ldots\ c_{N-1}]^T \qquad 5.4$$

$$A_k = \begin{bmatrix} \lambda_0(\Delta t)\exp(-\Delta t) & \lambda_1(\Delta t)\exp(-\Delta t) & \ldots & \lambda_{N-1}(\Delta t)\exp(-\Delta t) \\ \lambda_0(2\Delta t)\exp(-2t) & \lambda_1(2\Delta t)\exp(-2\Delta t) & \ldots & \lambda_{N-1}(2\Delta t)\exp(-2\Delta t) \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \lambda_0(k\Delta t)\exp(-k\Delta t) & \lambda_1(k\Delta t)\exp(-k\Delta t) & \ldots & \lambda_{N-1}(k\Delta t)\exp(-k\Delta t) \end{bmatrix} \qquad 5.5$$

Thus the problem of finding the coefficient of Laguerre polynomials have been converted to the problem of finding the vector C_k

$$C_k = A_k^{-1} X_k \qquad 5.6$$

Where A_k^{-1} is the generalized inverse of A_k

We make use of Greville's algorithm to find the generalized inverse of the matrix, which help to find the value of coefficients of laguerre polynomials at each data point.

5.2 IDENTIFICATION OF LINEAR LUMPED SYSTEM USING LAGUERRE POLYNOMIAL

In this chapter, we consider the problem of identifying a linear time invariant lumped parameter system using laguerre polynomials.

In this method the differential equation model is first converted to an integral equation model. By expanding the input - output signals and the initial boundary conditions in Laguerre polynomial, ultimately a linear system of algebraic equation is derived.

The equations are solved using the Neural network to get an estimate of the parameters.

5.2.1 PROBLEM FORMULATION :

Consider a linear time invariant lumped parameter system described by the following differential equation.

$$a_2 d^2 y(t)/dt + a_1 dy(t)/dt + a_0 y(t) = U(t) \qquad 5.7$$

Where a_2, a_1 & a_0 are unknown constant parameter.

Given a record of output y (t) and input U (t), the problem is to estimate the parameters a_2, a_1 and a_0.

5.2.2 MATHEMATICAL PRELIMINARIES :-

A function y (t) where t lies between 0 and ∞, which is square integrable can be expanded in a Laguerre series.

$$Y(t) = y_0 \lambda_0(t) + y_1 \lambda_1(t) + y_2 \lambda_2(t) + \ldots\ldots$$

$$= \sum_{I=0}^{\infty} y_I \lambda_I(t) \qquad .5.8$$

Where λ_i (t), i = 0, 1,2, ... are the laguerre polynomial and y_0, y_1 ..... are the coefficient of laguerre polynomials given by.

$$Y_I = \int_0^{\infty} \exp(-t)\, y(t)\, \lambda_I(t)\, dt \qquad 5.9$$

An approximation of y (t) in terms of the first N terms of the laguerre polynomial will be

$$y(t) = \sum_{I=0}^{\infty} y_I \lambda_I(t) = y_N^T \lambda_N(t) \qquad .5.10$$

Where

$y_N = [y_0, y_1, \ldots\ldots y_{N-1}]$

$\lambda_N = [\lambda_0(t)\ \lambda_1(t)\ \ldots\ldots \lambda_{N-1}(t)]$

The vector function λ_N (t) has the property

$$\int_0^t \cdots \int_0^t \lambda_N(t)\, dt \ldots dt = P_N^r \lambda_N(t) \; ; \; t \in [0, \infty) \qquad 5.11$$

r - times r - times

Where P_N^r is an $N \times N$ constant matrix which has the general form as given below ,

$$P_N^r = \begin{bmatrix} rC_0 & -rC_1 & \ldots & (-1)^{N-1} & rC_{N-1} \\ 0 & rC_0 & \ldots & (-1)^{N-2} & rC_{N-2} \\ 0 & 0 & \ldots & & \\ \ldots & \ldots & \ldots & & \\ 0 & 0 & \ldots & & rC_0 \end{bmatrix}$$

Where $rC_i = [\, r(r-1)(r-2) \ldots (r-i+1) \,] / i! \quad ; i = 0,1 \ldots N-1$

Applying the above property to the truncated series of y (t) we get,

$$\int_0^t \cdots \int_0^t Y_N^T \lambda_N(t)\, dt \ldots dt = Y_N^T P_N^r \lambda_N(t) \qquad 5.12$$

r - times r - times

5.2.3 THE IDENTIFICATION PROCESS :

Integrating equation 5.7 twice with respect to t we can remove all derivatives. The advantage is that the above conversion incorporate the boundary and initial conditions in an easy manner. Also, the result possesses the

smoothing property inherent in all integrals in contrast to differential or derivatives. We have, therefore, by not considering the initial conditions.

$$a_2 y(t) + a_1 \int_0^t y(t)\,dt + a_0 \int_0^t \int_0^t y(t)\,dt\,dt = \int_0^t \int_0^t U(t)\,dt\,dt \qquad 5.13$$

Now the input and output signals can be expanded into Laguerre series.

$$u(t) = U_{MN}\, \lambda_N(t) \qquad 5.14$$

$$y(t) = Y_{MN}\, \lambda_N(t) \qquad 5.15$$

Substituting the equations 5.11, 5.14, 5.15 in 5.13, we have

$$[a_2 P_M^T Y_{MN} + a_1 Y_{MN} P_N + a_0 P_M^T Y_{MN} P_N]\, \lambda_N(t) = [P_M^T U_{MN} P_N]\, \lambda_N(t) \qquad 5.16$$

Equating the corresponding coefficient of the Laguerre polynomials $\lambda_l(t)$, $l = 0, 1, \ldots N-1$, we have

$$a_2 P_M^T Y_{MN} + a_1 Y_{MN} P_N + a_0 P_M^T Y_{MN} P_N = P_M^T U_{MN} P_N \qquad 5.17$$

Equation 5.17 can be written as

$$A\,0 = h \qquad 5.18$$

Where

$$A = \begin{bmatrix} (P_M^T Y_{MN})_1 & (Y_{MN} P_N)_1 & (P_M^T Y_{MN} P_N)_1 \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ (P_M^T Y_{MN})_N & (Y_{MN} P_N)_N & (P_M^T Y_{MN} P_N)_N \end{bmatrix}$$

$$h = \begin{bmatrix} (P_M^T U_{MN} P_N)_1 \\ \cdots \\ \cdots \\ (P_M^T U_{MN} P_N)_N \end{bmatrix}$$

$() = [a_2\ a_1\ a_0]^T$

Where $(P_M^T\ Y_{MN})_i$, $(Y_{MN}\ P_{.N})_i$ ($P_{.i}$ $Y_{MN}\ P_N)_i$ etc indicates th column of the corresponding matrix. Matrix A and h can be evaluated row by row.

By giving value of A and h in the Neural network we can calculate $()$ accurately.

CHAPTER VI

LINEAR DISTRIBUTED SYSTEM

6.1 LEAST SQUARE ESTIMATION OF COEFFICIENT OF LAGUERRE POLYNOMIAL - [DOUBLE VARIABLE CASE]

Laguerre polynomials $\lambda_i(t)$ forms an orthonormal basis over t which lies between 0 and ∞. Also $\psi_j(x)$ are Lageuerre polynomials in variable X, which lies between 0 and ∞. An orthonormal basis over $t \in [0, \infty)$, $X \in [0, \infty)$, is therefore formed by the double laguerre polynomials $\{\lambda_i(t) \cdot \psi_j(x)\}$.

Now consider the two variable function $y(x,t)$; $x,t \in [0, \infty)$. Then $y(x,t)$ is expandable in double Laguerre series.

$$(\text{i.e})\ y(x,t) = \sum_{j=0}^{\infty} \sum_{i=0}^{\infty} y_{ij}\ \lambda_i(t) \cdot \psi_j(x) \qquad 6.1$$

Where y_{ij} are coefficients of the double Laguerre polynomial expansion given by

$$y_{ij} = \int_0^{\infty} \int_0^{\infty} \exp(-t) \exp(-x)\ y(x,t)\ \lambda_i(t)\ \psi_j(x)\, dx\, dt \qquad 6.2$$

An approximation of y (x,t) will then be

$$\begin{aligned} y(x,t) &= \sum_{j=0}^{M-1} \sum_{i=0}^{N-1} y_{ij}\ \lambda_i(t) \cdot \psi_j(x) \\ &= \psi_M^T(x)\ Y_{MN}\ \lambda_N(t) \\ &= \lambda_N^T(t)\ Y_{MN}^T\ \psi_M(x) \end{aligned} \qquad 6.3$$

Where,

$$\psi_M(x) = [\psi_0(x)\ \psi_1(x) \ldots\ldots \psi_{M-1}(x)]$$

$$\lambda_N(t) = [\lambda_0(t)\ \lambda_1(t) \ldots\ldots \lambda_{N-1}(t)]$$

$$Y_{MN} = \begin{bmatrix} Y_{00} & Y_{01} \ldots\ldots & Y_{0,N-1} \\ Y_{10} & Y_{11} \ldots\ldots & Y_{1,N-1} \\ \ldots\ldots & & \\ \ldots\ldots & & \\ Y_{M-1,0} & Y_{M-1,1} \ldots\ldots & Y_{M-1,N-1} \end{bmatrix} \qquad 6.4$$

Multiplying equation 6.1 on both sides by $\{\exp(-x)\exp(-t)\}$

and then discretizing we have

$$\exp(-k\Delta x)\ \exp(-l\Delta t)\ y(k\Delta x, l\Delta t) =$$

$$\sum_{j=0}^{M-1}\ \sum_{i=0}^{N-1} \exp(-k\Delta x)\ \exp(-l\Delta t)\ y_{ij}\ \lambda_i(l\Delta t)\ \psi_j(k\Delta x) \qquad 6.5$$

Normally at any instant of time we can measure the value of y(x,t) at all the possible points, and then measurement for another instant of time. Equation 6.5 can be rewritten as

$$Y_{kl} = A_{kl}\theta_{kl} \qquad 6.6$$

Where

$$Y_{kl} = [\exp(-\Delta x)\exp(-\Delta t)\ y(\Delta x, \Delta t) \ldots. \exp(-k\Delta x)\exp(-\Delta t),$$
$$y(k\Delta x, \Delta t)\exp(-\Delta x)\exp(-2\Delta t)\ y(\Delta x, 2\Delta t) \ldots..$$
$$\exp(-k\Delta x)\ \exp(-2\Delta t)\ y(kx, 2\Delta t) \ldots.. \exp(-k\Delta x)$$
$$\exp(-l\Delta t)\ y(k\Delta x, l\Delta t)]^T \qquad 6.7$$

$$\theta_{ki} = [\, y_{00} \;\; y_{10} \;\; \ldots\ldots \;\; y_{M-1,0} \;\; y_{01} \;\; y_{M-1,1} \;\; \ldots\ldots \;\; y_{0,N} \;\; \ldots\ldots \;\; y_{M-1,N-1} \;\; y_{N-1} \,] \qquad 6.8$$

$$A_{ki} = \begin{bmatrix} \lambda_0(\lambda t)\exp(-\lambda t)\psi_0(\lambda x)\exp(-\lambda x) & \ldots\ldots & \exp(-\lambda x)\lambda_{n-1}(\lambda t)\exp(-\lambda t)\psi_{n-1}(\lambda x) \\ \ldots\ldots & \ldots\ldots & \ldots\ldots \\ \lambda_0(k\lambda t)\exp(-k\lambda t)\psi_0(k\lambda x)\exp(-k\lambda x) & \ldots\ldots & \exp(-k\lambda x)\lambda_{n-1}(k\lambda t)\exp(-k\lambda t)\psi_{n-1}(k\lambda x) \\ \ldots\ldots & \ldots\ldots & \ldots\ldots \\ \lambda_0(l\lambda t)\exp(-k\lambda x)\exp(-l\lambda t)\psi_0(k\lambda x) & \ldots\ldots & \lambda_{n-1}(l\lambda t)\exp(-l\lambda t)\psi_{n-1}(k\lambda x)\exp(-k\lambda x) \end{bmatrix}$$

6.9

Using the recurrence relation for $\lambda_i(t)$ as given in the equation

$$(i+1)\lambda_{i+1}(t) = (1+2i-t)\lambda_i(t) - i\lambda_{i-1}(t)$$

and also for $\psi_i(x)$ it is possible to generate A_k matrix row by row corresponding to each measurement of y (x,t). Thus we can use Greville's algorithm for the two variable also to find the least - square estimate of θ_{ki}.

6.1 IDENTIFICATION OF LINEAR DISTRIBUTED SYSTEM USING LAGUERRE POLYNOMIAL

In this chapter, we consider the problem of identification of a linear time invariant distributed parameter system using Laguerre polynomials. In this method, the partial differential equation model is first converted to an integral equation model. By expanding the input - output equation and initial and

P-345

boundary conditions in Laguerre polynomials, ultimately, a linear system of algebraic equation is derived.

The equations are solved using the Neural network to get an estimation of the parameter.

6.1.1 PROBLEM FORMULATION :

Consider a linear time - invariant distributed parameter distributed parameter system described by the following partial differential equation

$$a_2 \partial y(x,t)/\partial t + a_1 \partial y(x,t)/\partial x + a_0 y(x,t) = U(x,t) \qquad 6.10$$

Where a_2, a_1 and a_0 are unknown constant parameters.

Given a record of output y (x,t) and input u (x,t) the problem is to estimate the parameters a_2, a_1, and a_0.

6.1.2 MATHEMATICAL PRELIMINARIES :-

A function y (t), where t lies between 0 and ∞, which is square integrable can be expanded in a Laguerre series

$$y(t) = y_0 \lambda_0(t) + y_1 \lambda_1(t) + y_2 \lambda_2(t) + \ldots$$

$$= \sum_{i=0}^{\infty} y_i \lambda_i(t) \qquad 6.11$$

Where λ_i (t), i = 0, 1,2, ... are the Laguerre polynomials and $y_0, y_1, \ldots$ are the coefficient of Laguerre polynomials given by

$$y_i = \int_0^{\infty} \exp(-t)\, y(t)\, \lambda_i(t)\, dt \qquad 6.12$$

an approximation of y(t) in terms of the first N terms of the Laguerre polynomials will be

$$y(t) = \sum_{i=0}^{N-1} y_i \lambda_i(t) = Y_N^T \lambda_N(t) \qquad 6.13$$

where,

$$Y_N = [\, y_0 \; y_1 \ldots y_{N-1}]$$

$$\lambda_N = [\lambda_0(t) \; \lambda_1(t) \ldots \lambda_{N-1}(t)\,]$$

The vector function $\lambda_N(t)$ has the property

$$\underbrace{\int_0^t \int_0^t \ldots \int_0^t}_{r\text{ - times}} \exp(-t)\, y(t)\, \lambda_i(t)\, dt \ldots dt = P_N^r \, \lambda_N(t) \qquad 6.14$$

where P_N^r is an NX N constant matrix which has the general form as

$$P_N^r = \begin{bmatrix} {}^rC_0 & -{}^rC_1 & \ldots & (-1)^{N-1}\, {}^rC_{N-1} \\ 0 & {}^rC_0 & \ldots & (-1)^{N-2}\, {}^rC_{N-2} \\ 0 & 0 & \ldots & \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & {}^rC_0 \end{bmatrix} \qquad 6.15$$

Where ${}^rC_i = [\, r(r-1)(r-2) \ldots (r-i+1)\,] / i! \quad ; i = 0,1 \ldots N-1$

Applying the above property to the truncated series of y (t) we get,

$$\int_0^t \cdots \int_0^t Y_N \lambda_N(t)\, dt \cdots dt = Y_N P_N \lambda_N(t) \qquad 6.16$$

r - times r - times

Consider a two variable function y (x,t) which is square integrable in the interval x, t $\in$ [0, ∞). Then the function y (x,t) can be expanded in a double Laguerre series.

$$y(x,t) = \sum_{j=0}^{\infty} \sum_{i=0}^{\infty} y_{ij} \lambda_i(t) \psi_j(x) \qquad 6.17$$

where y_{ij} are the coefficients of the double Laguerre expansion, given by

$$y_{ij} = \int_0^{\infty} \int_0^{\infty} \exp(-t) \exp(-x)\, y(x,t)\, \lambda_i(t) \psi_j(x)\, dx\, dt \qquad 6.18$$

An approximation to y (x,t) will be

$$y(x,t) = \sum_{j=0}^{M-1} \sum_{i=0}^{N-1} y_{ij} \lambda_i(t) \psi_j(x)$$

$$= \psi_M^T(x)\, Y_{MN}\, \lambda_N(t)$$

$$= \lambda_N^T(t)\, Y_{MN}\, \psi_M(x) \qquad 6.19$$

Where

$$Y_{MN} = \begin{bmatrix} y_{00} & y_{01} & \cdots & y_{0\,N-1} \\ y_{10} & y_{11} & \cdots & y_{1\,N-1} \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ y_{M-1\,0} & y_{M-1\,1} & \cdots & y_{M-1\,N-1} \end{bmatrix}$$

$$\psi_M(x) = [\psi_0(x), \psi_1(x), \ldots \psi_{M-1}(x)]^T$$

$$\lambda_N(t) = [\lambda_0(t)\ \lambda_1(t) \ldots \lambda_{N-1}(t)]^T$$

So we have

$$\int_0^x \int_0^x \ldots \int_0^t \int_0^t \psi_M^T(x)\ Y_{MN}\ \lambda_N(t)\, dt \ldots dt\, dx \ldots dx$$

(p times; q times; q times; p times)

$$= \psi_M^T(x)\,(P_M^T)^{-p}\ Y_{MN}\ P_N^{-q}\ \lambda_N(t)$$

$$= \lambda_N^{-1}(t)\,(P_N^T)^{-q}\ Y_{MN}^T\ P_M^{-p}\ \psi_M(x) \qquad 6.20$$

6.1.3 THE IDENTIFICATION PROCESS

Integrating equation 6.10 once with respect to x and t we can remove all derivatives. The advantage is that the above conversion incorporates the initial and boundary conditions in an easy manner. Also, the result possesses the smoothening property inherent in all integral in contrast to differential or derivatives. We have therefore

$$a_2 \int_0^x y(x,t)\,dx + a_1 \int_0^t y(x,t)\,dt + a_0 \int_0^x \int_0^t y(x,t)\,dt\,dx$$
$$- a_2 \int_0^x y(x,0)\,dx - a_1 \int_0^t y(0,t) = \int_0^x \int_0^t U(x,t)\,dt\,dx \qquad 6.21$$

To determine the unknown function y (o,t) and y (x,0) , we start by approximating by a finite Laguerre series.

$$y(0,t) = \sum_{i=0}^{r-1} C_i\, \lambda_i(t) = C_r^T\, \lambda_r(t); \quad r \le N$$

$$= \sum_{i=0}^{r-1} C_i\, \psi_M^T(x)\ E_{i+1}\ \lambda_N(t) \qquad 6.22$$

and

$$y(x,0) = \sum_{i=0}^{l-1} B_i \psi_i(x) = B^T \psi_l(x), \quad l \le M$$

$$= \sum_{i=0}^{l-1} B_i \psi_M^T(x) E_{i+1,1} \lambda_N(t) \qquad 6.23$$

where E_{ij} is an M xN matrix having the ij^{th} element unity and the remaining elements zero. Clearly, once $C_0, C_1, \ldots C_{r-1}$ & $B_0, B_1, \ldots B_l$ are determined then y (0,t) and y (x,0) are approximately constructed from the equations 6.22 and 6.23.

Now the input and output can be expanded into double Laguerre series.

$$u(x,t) = \psi_M^T(x) U_{MN} \lambda_N(t)$$
$$y(x,t) = \psi_M^T(x) Y_{MN} \lambda_N(t) \qquad 6.24$$

Substituting the equations 6.22, 6.23 and 6.24 in equation 6.21, we have

$$\psi_M^T(x) [a_2 P_M^T Y_{MN} + a_1 Y_{MN} P_N + a_0 P_M^T Y_{MN} P_N - a_2 \sum_{i=0}^{l-1} B_i P_M^T E_{i+1,1}$$

$$- a_1 \sum_{i=0}^{l-1} C_i E_{i,i+1} P_N] \lambda_N(t) = \psi_M^T(x) [P_M^T U_{MN} P_N] \lambda_N(t) \qquad 6.25$$

Equating the corresponding coefficients of the Laguerre polynomial product $\{\lambda_j(t) \psi_i(x)\}$ i = 0,1, ... , N-1; j = 0.1, ... M-1, we have

$$a_2 P_M^T Y_{MN} + a_1 Y_{MN} P_N + a_0 P_M^T Y_{MN} P_N + a_2 \sum_{i=0}^{l-1} B_i P_M^T E_{i+1,1}$$

$$+ a_1 \sum_{i=0}^{l-1} C_i E_{i,i+1} P_N = P_M^T U_{MN} P_N \qquad 6.26$$

Where $\underline{B}_i = -a_2 B_i$ and $\underline{C}_i = -a_1 C_i$

Equation 6.26 can be written as

$$A\theta = h \qquad 6.27$$

Where

$$A = \begin{bmatrix} (P_M^T Y_{MN})_1 & (Y_{MN} P_N)_1 & (P_M^T Y_{MN} P_N)_1 & (P_M^T E_{11})_1 \\ \cdots & \cdots & \cdots & \cdots \\ (P_M^T Y_{MN})_N & (Y_{MN} P_N)_N & (P_M^T Y_{MN} P_N)_N & (P_M^T E_{11})_N \\ \cdots & (P_M^T E_{11})_1 & (E_{11} P_N)_1 & (E_{11} P_N)_1 \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & (P_M^T E_{11})_N & (E_{11} P_N)_N & (E_{11} P_N)_N \end{bmatrix} \qquad 6.28$$

$$h = \begin{bmatrix} (P_M^T U_{MN} P_N)_1 \\ \cdots \\ \cdots \\ (P_M^T U_{MN} P_N)_N \end{bmatrix} \qquad 6.29$$

$$\theta = [a_2\ a_1\ a_0\ \underline{B}_1\ \underline{B}_2 \ldots \underline{B}_{n-1}\ \underline{C}_0\ \underline{C}_1 \ldots \underline{C}_{n-1}]^T \qquad 6.30$$

Where $(P_M^T Y_{MN})_i$, $(Y_{MN} P_N)_i$, $(P_M^T Y_{MN} P_N)_i$ etc indicates ith column of the corresponding matrix. Matrix A and h can be evaluated row by row.

By giving value of A and h in the Neural network we can calculate θ accurately.

Chapter 7

Software

CHAPTER VII

SOFTWARE

7.1 GENERAL PROCEDURE TO FIND PARAMETERS OF A SYSTEM

STEP 1 : Be ready with input, output and system equation.

STEP 2 : Convert the input equation into truncated Laguerre series.

STEP 3 : Convert the output equation into truncated Laguerre series.

STEP 4 : Convert the system equation into truncated Laguerre series with help of truncated Laguerre series of input and output equation.

STEP 5 : Estimate the parameters of the system with help of truncated Laguerre series of input, output and system equations by using Perceptron Least mean square Algorithm.

7.2 STEP BY STEP PROCEDURE TO ESTIMATE PARAMETERS OF LINEAR LUMPED SYSTEM

STEP 1 : Be ready with input, output and system equations of linear lumped system.

STEP 2 : Estimate coefficients of truncated Laguerre series for input and output equation using Least square estimation procedure for single variable case (refer Appendix 1 for more details).

STEP 3 : Estimate the inputs to two layer perceptron network using Algorithm given in Appendix 2

PROGRAMS :

(i). Program for the Least Square estimation of coefficients of Laguerre Polynomials - double variable case for input and output equation

(ii). Program to find inputs to two layer Perceptron network.

(iii). Program to estimate parameters of Linear Distributed system with Perceptron LMS Algorithm.

(Iv). Display the outputs.

STEP 4 : Estimate the parameters of the linear lumped system with use of perceptron LMS Algorithm. (Refer Appendix 5)

PROGRAMS :

(I) Program for least square estimation of coefficients of Laguerre polynomials for single variable case for input and output equations

(Ii) Program to find inputs to two layer Perceptron network.

(Iii) Program to estimate parameters of linear lumped system with Perceptron LMS Algorithm.

(Iv) Display the outputs.

7.3 STEP BY STEP PROCEDURE TO ESTIMATE PARAMETERS OF LINEAR DISTRIBUTED SYSTEM

STEP1 : Be ready with input, output and system equations of Linear Distributed system.

STEP 2 : Estimate coefficients of truncated Laguerre Series for input and output equations using Least Square Estimation procedure for double variable case.(Refer Appendix 3)

STEP 3 : Estimate the inputs to two layer Perceptron network with the help of Algorithm in Appendix 2.

STEP 4 : Estimate the parameters of the Linear Distributed system with use of Perceptron LMS Algorithm. (Refer Appendix 4)

```
        /*** PROGRAM FOR TWO LAYER PERCEPTRON NETWORK ***/
      /* THIS IS A PROGRAM TO SOLVE SIMULTANEOUS EQUATIONS */

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
#include<dos.h>
#include<alloc.h>
#include<time.h>
# define LEARN 0.5
# define N 1

                                /*** FUNCTIONS TO NORMALIZE ***/

void norm(float **coef,float *cons,int n)
{
        int register c1,c2;
        float max=0.0;

        for(c1=0;c1<n;c1++)
        {
                        for(c2=0;c2<n;c2++)
                        {
                                if (coef[c1][c2] >max)
                                max=coef[c1][c2];
                        }
        }
        for(c2=0;c2<n;c2++)
        {
                if(cons[c2]>max)
                {
                        max=cons[c2];
                }
        }
        for(c1=0;c1<n;c1++)
        {
                for(c2=0;c2<n;c2++)
                {
                        coef[c1][c2]=coef[c1][c2]/max;
                }
        }
        for(c2=0;c2<n;c2++)
        {
                cons[c2]=cons[c2]/max;
        }
}

                                /***FUNCTION TO FIND ERROR***/

void find_op(float *sol,float **coef,float *cons,int n,float *err,float *cum_err
)
{
        int register c1,c2;
        *cum_err = 0;
        for(c1=0;c1<n;c1++)
        {
                err[c1]=0.0;
```

```
        }
        for(c1=0;c1<n;c1++)
        {
                for(c2=0;c2<n;c2++)
                {
                        err[c1]+=coef[c1][c2]*sol[c2];
                }
        }
        for(c1=0;c1<n;c1++)
        {
                err[c1]=cons[c1]-err[c1];
                *cum_err+=0.5*err[c1]*err[c1];
        }
}

                                /***FUNCTION TO CHANGE WEIGHT***/

void wtchange(float *cum err,int n, float *sol,float **coef,float *err)
{
        int register c1,c2;
        for(c1=0;c1<n;c1++)
        {
                for(c2=0;c2<n;c2++)
                {
                        sol[c2]+=LEARN*exp(-1*(*cum err))*err[c1]*coef[c1][c2];
                }
        }
}
                                /***FUNCTION TO GIVE INITIAL RANDOM INPUT***/

void serand(float *sol, int n)
{
        int register c1;
                for(c1=0;c1<n;c1++)
                {
                        sol[c1]=random(N+1);
                }
}
                                /*** MAIN PROGRAM ***/

main()
{
        /*** INTIALIZING ***/

        float**coef,*cons;
        float *sol,*err,cum_err=0,a[10][10],b[10],p[10][10];
        int n,c1,c2,i,j,k,col,row;
        long int count=0;
        clrscr();


        textbackground(2);
        textcolor(4);
        printf("ENTER THE NO.OF VARIABLES-");
        scanf("%d",&n);
        printf("Enter the numberof rows in the A matrix ");
        scanf("%d",&row);
        printf("Enter the number of cloumns in the A matrix");
        scanf("%d",&col);
        if(row==col)
        {
```

```
                coef=(float **)calloc(n,sizeof(float *));
                cons=(float *)calloc(n,sizeof (float));
                err=(float*)calloc(n,sizeof(float));
                for(c1=0;c1<n;c1++)
                {
                        coef[c1]=(float*)calloc(n,sizeof(float));
                }
                printf ("ENTER THE A MATRIX \n");
                for(c1=0;c1<n;c1++)
                {
                        for(c2=0;c2<n;c2++)
                        {
                                scanf("%f",&coef[c1][c2]);
                        }
                }
                printf("\nENTER THE B MATRIX\n");
                for(c2=0;c2<n;c2++)
                {
                        scanf("%f",&cons[c2]);
                }
        }
else
        {
                printf("Enter the A matrix ");
                for(i=0;i<row;i++)
                {
                        for(j=0;j<col; j++)
                        {
                                scanf("%f",&a[i][j]);
                        }
                }
                printf("Enter the b matrix ");
                for(i=0;i<row;i++)
                {
                        scanf("%f",&b[i]);
                }
                coef=(float**)calloc(n,sizeof(float*));
                cons=(float*)calloc(n,sizeof(float));
                err=(float*)calloc(n,sizeof(float));
                for(c1=0;c1<n;c1++)
                {
                        coef[c1]=(float*)calloc(n,sizeof(float));
                }
                for(i=0;i<col;i++)
                {
                         for(j=0;j<col;j++)
                         {
                                        coef[i][j]=0.0;
                         }
                }
                for(i=0;i<col;i++)
                {
                        for(j=0;j<col;j++)
                        {
                                for(k=0;k<row;k++)
                                {
                                        coef[i][j]=coef[i][j]+a[k][i]*a[k][j];
                                }
                        }
                }
                printf("the coef matrix is");
                for(i=0;i<col;i++)
                {
```

```
                        for(j=0;j<col;j++)
                        {
                                printf("%f",coef[i][j]);
                        }
                }
                for(i=0;i<col;i++)
                cons[i]=0.0;
                for(i=0;i<col;i++)
                {
                        for(j=0;j<row;j++)
                        {
                                cons[i]=cons[i]+a[j][i]*b[j];
                        }
                }
                printf("the cons matrix");
                for(i=0;i<col;i++)
                {
                        printf("%f",cons[i]);
                }
        }
        serand(sol,n);
        norm(coef,cons,n);
        do
        {
                find_op(sol,coef,cons,n,err,&cum_err);
                wtchange(&cum_err,n,sol,coef,err);
                printf("    ERROR-  %f\n",cum_err);
                for(c1=0;c1<n;c1++)
                {
                        printf("  %f",sol[c1]);
                }
                count=count+1;
        }
                while(cum_err>0.0000001);

                for(c1=0;c1<n;c1++)
                {
                        printf("\n\tsol[%d]=%f",c1,sol[c1]);
                }
                printf("\nThe number of iterations:");
                printf("%ld",count);
                return 0;
}
 /*** MAIN PROGRAM ENDS HERE ***/
```

```
        /* PROGRAM TO FIND PARAMETERS OF THE LINEAR LUMPED SYSTEM
              USING MATRIX INVERSION METHOD */

#include<stdio.h>
                        /*** MAIN PROGRAM STARTS HERE ***/

main()
{
                        /* INTIALIZATION */

        float XK[100],CK[100],x[100],AK[100][100],AKI[100][100],a[100],d[100],mu
l[100],c[100],check[100],w[100],b[100],s[100][100];
        int order,i,j,k,novar,itno=1,
        l,m,c2;
        float u=0.0,y=0.0,z=0.0,W=0.0;
                float YMN[10],PMT[10][10],PN[10][10],PMTYMN[10],UMN[10],PMTUMN[1
0],
                H[10],A[10][10];
        int PM=1,M,N;
        clrscr();

        /* PROGRAM TO CHANGE GIVEN SYSTEM EQUATION TO TRUNCATED LAGUERRE SERIES
*/


        printf("Enter number of row & coloumnsof laguere truncated\n");
        printf("coefficient of input & output\n");
        scanf("%d %d",&M,&N);
        for(i=0;i<N;i++)
        {
                for(j=0;j<N;j++)
                {
                        PN[i][j]=0.0;
                        PMT[i][j]=0.0;
                        PMTYMN[i]=0.0;
                        PMTUMN[i]=0.0;
                        A[i][j]=0.0;
                        H[i]=0.0;
                        UMN[i]=0.0;
                        YMN[i]=0.0;
                }
        }

        printf("This is for equation of type\n");
        printf("a2*d^2y(t)/dt+a1*dy(t)/dt+a0*y(t)=u(t)\n");


        printf("Enter the laguere coeficient of input u(t) as UMN\n");
        for(i=0;i<N;i++)
        {
                scanf("%f",&UMN[i]);
        }


        printf("Enter the laguere coeficient of output y(t) as YMN\n");
        for(i=0;i<N;i++)
        {
                scanf("%f",&YMN[i]);
        }

        for(i=0;i<N;i++)
```

```
{
        for(j=0;j<N;j++)
        {
                if(i==j)
                {
                        PN[i][j]=1.0;
                        if((j+1)<N)
                        {
                                PN[i][j+1]=-1.0;
                        }
                }
        }
}

for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                PMT[i][j]=PN[j][i];
        }
}

for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                if(j==0)
                {
                        A[i][j]=PM*YMN[i];
                }
        }
}


for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                A[i][1]+=YMN[j]*PN[j][i];

        }
}


for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                PMTYMN[i]+=PMT[i][j]*YMN[j];

        }
}


for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                A[i][2]+=PMTYMN[j]*PN[j][i];
        }
}
```

```
        for(i=0;i<N;i++)
        {
                for(j=0;j<N;j++)
                {
                        PMTUMN[i]+=PMT[i][j]*UMN[j];
                }
        }


        for(i=0;i<N;i++)
        {
                for(j=0;j<N;j++)
                {
                        H[i]+=PMTUMN[j]*PN[j][i];
                }
        }

        printf("\n\t\t\t\t***The A matrix is***\n\n");
        for(i=0;i<N;i++)
        {
                for(j=0;j<N;j++)
                {
                        printf("\t\t%f",A[i][j]);
                }
                printf("\n");
        }

        printf("\n\n\t\t\t***The H matrix is***\n\n");
        for(i=0;i<N;i++)
        {
                printf("\t\t\t\t%f\n",H[i]);
        }
        getch();


itno =N;
novar = N;
for(i=0;i<itno;i++)
{
        for(j=0;j<novar;j++)
        {
                AK[i][j] = A[i][j];
        }
}
for(i=0;i<novar;i++)
{
        XK[i]=H[i];
}


        /* TO FIND INVERSE OF A MATRIX USING GREVIELLE'S ALGORITHM  */


                for(j=0;j<itno;j++)
                {
                        a[j]=AK[j][0];
                        z=z+(a[j]*a[j]);
                }
```

```
for(j=0;j<itno;j++)
{
        AKI[0][j]=((1/z)*a[j]);
}

for(i=1;i<novar;i++)
{
        for(j=0;j<itno;j++)
        {
                a[j]=AK[j][i];
        }

        for(z=0;z<i;z++)
        {
                d[z]=0.0;
                for(l=0;l<itno;l++)
                {
                        d[z]=(d[z]+(AKI[z][l] * a[l]));
                }
        }


        for(l=0;l<itno;l++)
        {
                mul[l]=0.0;
                for(m=0;m<i;m++)
                {
                        mul[l]=mul[l]+(AK[l][m] * d[m]);
                }
        }

        for(j=0;j<itno;j++)
        {
                c[j]=a[j]-mul[j];
        }


        c2=0;
        for(j=0;j<itno;j++)
        {
                check[j]=0.0;
        }

        for(j=0;j<itno;j++)
        {
                 if(c[j]<=0.09 && c[j]>=0.00)
                 {
                        c[j]=floor(c[j]);
                 }
                 if(c[j]>=-0.09 && c[j]<0.00)
                 {
                         c[j]=ceil(c[j]);
                 }

                if(c[j]!= 0.0 )
                {
                        c2=1;
                        break;
                }
  }
  switch (c2)
  {
        case 0:
```

```
                {
                        y=0.0;
                        for(z=0;z<i;z++)
                        {
                                        y+=(d[z]*d[z]);
                        }
                        y=y+1;

                        for(l=0;l<itno;l++)
                        {
                                        w[l]=0.0;
                                        for(j=0;j<i;j++)
                                        {
                                                        W=0.0;
                                                        W = (d[j]*AKI[j]
[l]);
                                                        w[l]+= W;
                                         }
                        }

                        for(j=0;j<itno;j++)
                        {
                                        b[j]=(1/y)*w[j];
                        }
                        break;
                }

                case 1:
                {
                       u=0.0;
                       for(j=0;j<itno;j++)
                       {
                               u=u+(c[j]*c[j]);
                       }

                       for(j=0;j<itno;j++)
                       {
                               b[j]=(1/u)*c[j];
                       }
                       break;
                }
        }

        for(j=0;j<i;j++)
        {
                        for(l=0;l<itno;l++)
                        {
                        s[j][l]=0.0;
                        s[j][l]=s[j][l]+(d[j]*b[l]);
                        }
         }

         for(m=0;m<i;m++)
         {
                        for(j=0;j<itno;j++)
                        {
                                  AKI[m][j]=AKI[m][j]-s[m][j];
                        }
         }
         for(l=0;l<itno;l++)
         {
                         AKI[i][l]=b[l];
         }
      }
```

```
        /* TO PRINT PARAMETERS OF THE SYSTEM */

        printf("\n\n\t\t\t***The SOL Matrix is***\n\n\n");
        for(i=0;i<novar;i++)
        {
                CK[i] =0.0;
                for(j=0;j<itno;j++)
                {
                        CK[i]+=AKI[i][j]*XK[j];
                }
        }
        for(j=0;j<novar;j++)
        {
                printf("\t\t\t\t%f\n",CK[j]);
        }
        getch();
        return 0;
}

        /* MAIN PROGRAM ENDS HERE */
```

```
Enter number of row & coloumnsof laguere truncated
coefficient of input & output
1 3
This is for equation of type
a2*d^2y(t)/dt+a1*dy(t)/dt+a0*y(t)=u(t)
Enter the laguere coeficient of input u(t) as UMN
12
-10
2
Enter the laguere coeficient of output y(t) as YMN
2
-4
2
```

The A matrix is

2.000000	2.000000	2.000000
-4.000000	-6.000000	-8.000000
2.000000	6.000000	12.000000

The H matrix is

12.000000
-34.000000
34.000000

The SOL Matrix is

2.000064
2.999909
1.000035

```
/* PROGRAM TO ESTIMATE PARAMETERS OF LIN
EAR DISTRIBUTED SYSTEM
                    USING MATRIX INVERSION METHOD */

#include<stdio.h>
#include<math.h>

        /* MAIN PROGRAM STARTS HERE */

main()
{
        /* INTIALIZATION*/

        float XK[100],CK[100],x[100],AK[100][100],AKI[100][100],a[100],d[100],mu
l[100],c[100],check[100],w[100],b[100],s[100][100];
        int order,i,j,k,novar,itno=1,
        l,m,c2;
        float u=0.0,y=0.0,z=0.0,W=0.0;
        float YMN[10][10],PMT[10][10],PN[10][10],PMTYMN[10][10],UMN[10][10],PMTU
MNPN[10][10],UMNPN[10][10],
         H[10],A[10][10],PMTYMNPN[10][10],PM[10][10],YMNPN[10][10];
        int M,N;
        clrscr();

        /* PROGRAM TO CHANGE GIVEN SYSTEM EQUATION TO TRUNCATED LAGUERRE SERIES*
/


        printf("Enter number of row & coloumnsof laguere truncated\n");
        printf("coefficient of input & output\n");
        scanf("%d %d",&M,&N);
        for(i=0;i<M;i++)
        {
                for(j=0;j<N;j++)
                {
                        YMN[i][j]=0.0;
                        PMT[i][j]=0.0;
                        PN[i][j]=0.0;
                        PMTYMN[i][j]=0.0;
                        UMN[i][j]=0.0;
                        PMTUMNPN[i][j]=0.0;
                        UMNPN[i][j]=0.0;
                        PMTYMNPN[i][j]=0.0;
                        PM[i][j]=0.0;
                        YMNPN[i][j]=0.0;
                }
        }

        printf("This is for equation of type\n");
        printf("a2*dy(x,t)/dt+a1*dy(x,t)/dt+a0*y(x,t)=u(x,t)\n");


        printf("Enter the laguere coeficient of input u(t) as UMN\n");
        for(i=0;i<M;i++)
        {
                for(j=0;j<N;j++)
                {
                        scanf("%f",&UMN[i][j]);
                }
        }

        printf("Enter the laguere coeficient of output y(t) as YMN\n");
```

```
for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                scanf("%f",&YMN[i][j]);
        }
}
for(i=0;i<M;i++)
{
        for(j=0;j<M;j++)
        {
                if(i==j)
                {
                        PN[i][j]=1.0;
                        if((j+1)<M)
                        {
                                PN[i][j+1]=-1.0;
                        }
                }
        }
}


for(i=0;i<N;i++)
{
        for(j=0;j<N;j++)
        {
                if(i==j)
                {
                        PM[i][j]=1.0;
                        if((j+1)<N)
                        {
                                PM[i][j+1]=-1.0;
                        }
                }
                PMT[i][j]=PM[j][i];
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                for(k=0;k<M;k++)
                {
                        PMTYMN[i][j]+= PMT[i][k]* YMN[k][j];
                }
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                if((i+j)< N && (i==0))
                {
                        A[i+j][0]=PMTYMN[j][i];
                }
                if((i+j)<=N && i==1)
                {
```

```
                        A[i+j+1][0]= PMTYMN[j][i];
                }
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                for(k=0;k<M;k++)
                {
                        YMNPN[i][j]+=YMN[i][k]*PN[k][j];
                }
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                if((i+j)<N && i==0)
                {
                        A[i+j][1]=YMNPN[j][i];
                }
                if((i+j)<=N && i==1)
                {
                        A[i+j+1][1]= YMNPN[j][i];
                }
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                for(k=0;k<M;k++)
                {
                        PMTYMNPN[i][j]+=PMT[i][k]*YMNPN[k][j];
                }
        }
}


for(i=0;i<M;i++)
{
        for(j=0;j<N;j++)
        {
                if((i+j)<N && i==0)
                {
                        A[i+j][2]=PMTYMNPN[j][i];
                }
                if((i+j)<=N && i==1)
                {
                        A[i+j+1][2]= PMTYMNPN[j][i];
                }
        }
}

for(i=0;i<M;i++)
{
```

```
                for(j=0;j<N;j++)
                {
                        for(k=0;k<M;k++)
                        {
                                UMNPN[i][j]+=UMN[i][k]*PN[k][j];
                        }
                }
        }


        for(i=0;i<M;i++)
        {
                for(j=0;j<N;j++)
                {
                        for(k=0;k<M;k++)
                        {
                                PMTUMNPN[i][j]+=PMT[i][k]*UMNPN[k][j];
                        }
                }
        }


        for(i=0;i<M;i++)
        {
                for(j=0;j<N;j++)
                {
                        if((i+j)<N && i==0)
                        {
                                H[i+j]=PMTUMNPN[j][i];
                        }
                        if((i+j)<=N && i==1)
                        {
                                H[i+j+1]= PMTUMNPN[j][i];
                        }
                }
        }
        printf("\n\t\tThe A matrix is\n\n");
        for(i=0;i<(M+N);i++)
        {
                for(j=0;j<(M+N-1);j++)
                {
                        printf("\t%f",A[i][j]);
                }
                printf("\n");
        }

        printf("\n\t\t\t*** The H matrix is ***\n\n");
        for(i=0;i<(M+N);i++)
        {
                printf("\t\t\t\t%f\n",H[i]);
        }
        getch();
itno =M+N;
novar = N+M-1;
for(i=0;i<itno;i++)
{
        for(j=0;j<novar;j++)
        {
                AK[i][j] = A[i][j];
        }
}
for(i=0;i<itno;i++)
{
        XK[i]=H[i];
```

```
}

        /* TO FIND INVERSE OF MATRIX USING GREVIELLE'S ALGORITHM */


                for(j=0;j<itno;j++)
                {
                        a[j]=AK[j][0];
                        z=z+(a[j]*a[j]);
                }

                for(j=0;j<itno;j++)
                {
                        AKI[0][j]=((1/z)*a[j]);
                }

                for(i=1;i<novar;i++)
                {
                        for(j=0;j<itno;j++)
                        {
                                a[j]=AK[j][i];
                        }

                        for(z=0;z<i;z++)
                        {
                                d[z]=0.0;
                                for(l=0;l<itno;l++)
                                {
                                        d[z]=(d[z]+(AKI[z][l] * a[l]));
                                }
                        }


                        for(l=0;l<itno;l++)
                        {
                                mul[l]=0.0;
                                for(m=0;m<i;m++)
                                {
                                        mul[l]=mul[l]+(AK[l][m] * d[m]);
                                }
                        }

                        for(j=0;j<itno;j++)
                        {
                                c[j]=a[j]-mul[j];
                        }


                        c2=0;
                        for(j=0;j<itno;j++)
                        {
                                check[j]=0.0;
                        }

                        for(j=0;j<itno;j++)
                        {
                                if(c[j]<=0.09 && c[j]>=0.00)
                                {
```

```
                        c[j]=floor(c[j]);
                    }
                    if(c[j]>=-0.09 && c[j]<0.00)
                    {
                        c[j]=ceil(c[j]);
                    }

                    if(c[j]!= 0.0 )
                    {
                        c2=1;
                        break;
                    }
            }
            switch (c2)
            {
                case 0:
                {
                    y=0.0;
                    for(z=0;z<i;z++)
                    {
                        y+=(d[z]*d[z]);
                    }
                    y=y+1;

                    for(l=0;l<itno;l++)
                    {
                        w[l]=0.0;
                        for(j=0;j<i;j++)
                        {
                            W=0.0;
                            W = (d[j]*AKI[j]
[l]);
                            w[l]+= W;
                        }
                    }

                    for(j=0;j<itno;j++)
                    {
                        b[j]=(1/y)*w[j];
                    }
                    break;
                }

                case 1:
                {
                    u=0.0;
                    for(j=0;j<itno;j++)
                    {
                        u=u+(c[j]*c[j]);
                    }

                    for(j=0;j<itno;j++)
                    {
                        b[j]=(1/u)*c[j];
                    }
                    break;
                }
            }

            for(j=0;j<i;j++)
            {
                    for(l=0;l<itno;l++)
                    {
                    s[j][l]=0.0;
```

```
                        s[j][l]=s[j][l]+(d[j]*b[l]);
                        }
                }

                for(m=0;m<i;m++)
                {
                        for(j=0;j<itno;j++)
                        {
                                AKI[m][j]=AKI[m][j]-s[m][j];
                        }
                }
                for(l=0;l<itno;l++)
                {
                        AKI[i][l]=b[l];
                }
        }

/* TO PRINT PARAMETERS OF THE SYSTEM */

printf("\n\n\t\t\t***The SOL Matrix is***\n\n\n");
for(i=0;i<novar;i++)
{
        CK[i] =0.0;
        for(j=0;j<itno;j++)
        {
                CK[i]+=AKI[i][j]*XK[j];
        }
}
for(j=0;j<novar;j++)
{
        printf("\t\t\t\t%f\n",CK[j]);
}
getch();
return 0;
}

/* MAIN PROGRAM ENDS HERE */
```

CHAPTER VIII

CONCLUSION

Software in C language has been developed to estimate the parameters in Linear and Distributed systems using Neural network. A Lease square estimator is simulated using Hopfield and Perceptron networks for the estimation.

The advantage of using Neural network method over the matrix inversion method for parameter estimation is that as the complexity of the system increases, estimating the parameters becomes difficult and time consuming, whereas in Neural network method the time consumption is less i.e., it is faster in convergence. Speed depends on the time constant of the network and is not related to the scale of the problem.

REFERENCES

1. E.V. KRISHNAMURTHY AND S. K. SEN, " NUMERICAL ALGORITHM - COMPUTATION IN SCIENCE AND ENGINEERING", EAST - WEST PRESS PRIVATE LIMITED, NEW DELHI, 1986.
2. Dr. V. RANGANATHAN, " STUDIES IN PARAMETER ESTIMATION USING LAGUERRE POLYNOMIAL APPROACH - Ph.D. THESIS" DELHI, 1985.
3. CHALLA RAMA DEVI, " PARAMETER ESTIMATION USING NEURAL NETWORKS", COIMBATORE,1998.
4. ALISON CARLING," INTRODUCING NEURAL NETWORKS", GALGOTIA PUBLICATIONS PRIVATE LIMITED, NEW DELHI, 1996.
5. VALLURU RAO AND HAYAGRIVA RAO," C++ NEURAL NETWORKS AND FUZZY LOGIC", BPB PUBLICATIONS, NEW DELHI, 1996.

Appendix

$A_k \mid a_1\ a_2\ \ldots\ldots\ a_k]$

3. Compute d_k using $d_k = A_{k-1}^{-1}\ a_k$
4. Compute C_k using $C_k = a_k - A_{k-1}\ d_k$
5. If $C_k \neq 0$ then goto 6 else goto 7.
6. Compute b_k using $b_k = 1 / (C_k^T\ C_k)\ C_k^T$
7. Compute b_k using $b_k = 1 / (1 + d_k^T\ d_k)\ d_k^T\ A_{k-1}^{-1}$
8. Compute Ak^{-1} using

$$Ak^{-1} = \begin{bmatrix} A_{k-1}^{-1} - d_k\ b_k \\ b_k \end{bmatrix}$$

STEP 5 : Find Ck matrix at each iteration using $C\ k = Ak^{-1}\ Xk$

STEP 6 : Continue above process from step 2 till we get same values for Ck at consecutive iteration.

APPENDICES

APPENDIX - 1

ALGORITHM TO ESTIMATE COEFFICIENTS OF TRUNCATED LAGUERRE SERIES FOR INPUT AND OUTPUT EQUATIONS USING LEAST SQUARE ESTIMATION PROCEDURE FOR SINGLE VARIABLE CASE.

STEP 1 : Initialize AK matrix and XK matrix to null values and ITNO as1.

STEP 2 : Find XK matrix at each iteration using formula as

$$X_K = [\exp(-\lambda t) X(\lambda t) \quad \exp(-2\lambda t) \times (2\lambda t) \ldots \exp(-k\lambda t) \times (k\lambda t)]^T$$

STEP 3 : Find AK matrix at each iteration using the formula

$$A_k = \begin{bmatrix} \lambda_0(\lambda t)\exp(-\lambda t) & \lambda_1(\lambda t)\exp(-\lambda t) & \ldots & \lambda_{N-1}(\lambda t)\exp(-\lambda t) \\ \lambda_0(2\lambda t)\exp(-2t) & \lambda_1(2\lambda t)\exp(-2\lambda t) & \ldots & \lambda_{N-1}(2\lambda t)\exp(-2\lambda t) \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \lambda_0(k\lambda t)\exp(-k\lambda t) & \lambda_1(k\lambda t)\exp(-k\lambda t) & \ldots & \lambda_{N-1}(k\lambda t)\exp(-k\lambda t) \end{bmatrix}$$

STEP 4 : AK inverse is found using Greville's Algorithm as below

1. Initialise AK^{-1} Matrix as $AK^{-1} = 1/(a_1^T a_1)\, a_1^T$ and AK & CK matrix to 0.

And initialise other variables accordingly.

2. Enter AK matrix $AK = [a_1 \; a_2 \ldots a_k \ldots a_n]$

Where a_k is k^{th} column of given matrix AK

APPENDIX - 2

ALGORITHM TO ESTIMATE INPUTS TO TWO LAYER PERCEPTRON NETWORK.

STEP 1 : Initialize the variables required accordingly.

STEP 2 : Find P_N matrix using

$$P_N' = \begin{bmatrix} rC_0 - rC_1 \ldots\ldots (-1)^{N-1}\ rC_{N-1} \\ 0 \quad rC_0 \quad (-1)^{N-2}\ rC_{N-2} \\ 0 \quad 0 \ldots\ldots\ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ 0 \quad 0 \ldots\ldots\ldots\ldots\ldots rC_0 \end{bmatrix}$$

Where $rC_i = [\, r\,(r-1)\,(r-2) \ldots (r-i+1)\,] \,/\, i! \quad ; i = 0,1 \ldots\ldots N-1$

STEP 3 : Get values for YMN and UMN and find PMT matrix by using transpose of P_N matrix.

STEP 4 : Find A matrix using

$$A = \begin{bmatrix} (P_{M1}^T\ Y_{1MN})_1 \quad (Y_{MN}\ P_N)_1 \quad (P_{M1}^T\ Y_{1MN} P_N)_1 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ (P_{M1}^T\ Y_{1MN})_N \quad (Y_{MN}\ P_N)_N \quad (P_{M1}^T\ Y_{MN} P_N)_N \end{bmatrix}$$

and h matrix using ,

$$h = \begin{bmatrix} (P_M{}^T \; U_{MN} P_N)_1 \\ \cdots\cdots\cdots\cdots \\ \cdots\cdots\cdots\cdots \\ (P_M{}^T \; U_{MN} P_N)_N \end{bmatrix}$$

To find $()$ matrix using

$() = A^{-1} h$

where $() = [a_2 \; a_1 \; a_0]^T$

APPENDIX - 4

ALGORITHM FOR 2 LAYER PERCEPTRON NETWORK

STEP 1 : Initialise the variables required in the program appropriately.

STEP 2 : Initialize the solution matrix (output matrix)

STEP 3 : Get the input matrix values.

STEP 4 : Get the output matrix values.

STEP 5 : Find the output matrix by using formula

Output matrix = input matrix · weight matrix

STEP 6 : Find the error matrix by using formula

Error matrix = new output matrix - old output matrix.

STEP 7 : Find the cumulative error by using the formula

total cumulative error = 0.5 × error at each output neuron

STEP 8 : Find the new weight matrix using the formula

new weight matrix = learning factor · old weight matrix · error matrix.

STEP 9 : If total cumulative error is equal to 0 then goto step 10 else continue from step 5.

STEP 10 : Print the solution matrix.