Spring 1990

An artificial neural network for redundant robot inverse kinematics computation

Wibawa Utama
New Jersey Institute of Technology

AN ARTIFICIAL NEURAL NETWORK

FOR

REDUNDANT ROBOT INVERSE KINEMATICS COMPUTATION

WIBAWA UTAMA

Thesis submitted to the Faculty of
the Graduate School of the New Jersey Institute of Technology
in partial fulfilment of the requirements for the degree of
Master of Science in Electrical Engineering
1990

APPROVAL SHEET

Title of Thesis : **An Artificial Neural Network for Redundant Robot Inverse Kinematics Computation**

Name of Candidate : Wibawa Utama
Master of Science in Electrical Engineering , 1990

Thesis and Abstract :
Approved by

Dr. Edwin S. H. Hou Assistant Professor Department of Electrical Computer Engineering	Date

Other Members of :
Examination Committee

Dr. A. Robbi Professor Department of Electrical Computer Engineering	Date
Dr. Nirwan Ansari Assistant Professor Department of Electrical Computer Engineering	Date

VITA

Name	:	Wibawa Utama
Address	:	
Degree and date to be conferred	:	M.S. Electrical Engineering, May, 1990
Date of birth	:	
Place of birth	:	
Secondary Education	:	The First State Senior High School Jakarta Indonesia

Collegiate Institutions Attended	*Dates*	*Degree*	*Date of Degree*
N.J.I.T, Newark	1988-90	M.S.	May, 1990
University of Indonesia, Jakarta, Indonesia	1979-1985	"Sarjana"	August, 1985

Major : Electrical Engineering

Abstract

Title of Thesis : **An Artificial Neural Network for Redundant Robot Inverse Kinematics Computation**

Name of Candidate : **Wibawa Utama**, Master of Science, 1990.

Thesis Directed By : **Dr. Edwin S. H. Hou**, Assistant Professor
Department of Electrical & Computer Engineering

A redundant manipulator can be defined as a manipulator that has more degrees of freedom than necessary to determine the position and orientation of the end effector. Such a manipulator has dexterity, flexibility, and the ability to maneuver in presence of obstacles. One important and necessary step in utilizing a redundant robot is to relate the joint coordinates of the manipulator with the position and orientation of the end-effector. This specification is termed as the direct kinematics problem and can be written as

$$\mathbf{x} = f(\mathbf{q})$$

where $\mathbf{x}$ is a vector representing the position and orientation of the end-effector, $\mathbf{q}$ is the joint vector, and f is a continuous non-linear function defined by the design of the manipulator. The inverse kinematics problem can be stated as: Given a position and orientation of the end-effector, determine the joint vector that specifies this position and orientation. That is,

$$\mathbf{q} = f^{-1}(\mathbf{x}).$$

For non-trivial designs, f^{-1} cannot be expressed analytically. This paper presents a solution to the inverse kinematics problem for a redundant robot based on multilayer feed-forward artificial neural network.

ACKNOWLEDGEMENT

I wish to express my grateful appreciation to my advisor, Dr. Edwin S. H. Hou, for his guidances, encouragement, and patience.

He who learns but does not think is lost.

He who thinks but does not learn is in great danger.

Confucius

Table of Contents

List of Figures

List of Tables

Chapter 1

Inverse Kinematics : Problems and Solutions

1.1 Introduction

A redundant manipulator can be defined as a manipulator that has more degrees of freedom than necessary to determine the position and orientation of the end effector. This redundancy provides the manipulator with dexterity, flexibility, and the ability to maneuver in presence of obstacles. One important and necessary step in utilizing redundant robot is to relate the joint coordinates of the manipulator with the position and orientation of the end–effector. This specification is termed as the direct kinematics problem and can be written as

$$\mathrm{x} = f(\mathrm{q}) \tag{1.1}$$

where $\mathbf{x}$ is a (m x 1) vector (m is the degrees of freedom of the robot work space) representing the position and orientation of the end–effector, $\mathbf{q}$ is a (n x 1) joint vector, (n is the number of links) and f is a continuous non-linear function defined by the design of the manipulator. The inverse kinematics problem can be stated as: Given a position and orientation of the end–effector, we want to determine the joint vector that specifies this position and orientation. That is,

$$\mathbf{q} = f^{-1}(\mathbf{x}). \tag{1.2}$$

It should be noted that for non–trivial designs, f^{-1} cannot be expressed analytically.

1.2 Solutions to The Inverse Kinematics Problem

Various methods have been proposed to solve the inverse kinematics problem. They can be categorized into 4 different groups : Closed Form Solution [4], Numerical Solution [27], Generalized Inverse Solution [1], and Neural Network Solution [6]. The last two methods are currently being studied by researchers particularly for redundant manipulator which does not have closed form solution and cannot be solved by iterative numerical methods.

1.2.1 Closed Form Solution

Closed form solution can be obtained by solving Eq. (1.1) algebraically. It should be noted that most commercial robot manipulators have 6 degrees of freedom and it is possible to write their inverse kinematics solution in closed form if they satisfy one of the following sufficient conditions :

1. Three adjacent joint axes intersecting.

2. Three adjacent joint axes parallel to one another.

Examples for these solutions are inverse transform technique for Euler angle solution and geometric approach solution [4].

Inverse Transform Technique

In this technique, pure algebraic approach is used. The ill conditions (such as $\psi = \cos^{-1}(\frac{0.5}{\sin\theta})$), under which the solution is inconsistent, inaccurate or undefined, appeared in this approach can be eliminated by introducing a consistent trigonometric function, $atan(x, y)$, or arc tangent with two arguments, as proposed by Paul [21]. Joint coordinates are then solved by equating and premultiplying or postmultiplying matrix equations. Although this technique provides us a general approach in determining the joint solution of a manipulator, it does not give a clear indication on how to select an appropriate solution from the several possible solutions for a particular arm configuration. This merely depends on the user's geometric intuition.

Geometric Approach

By geometric approach, a consistent joint angle solution can be derived. In addition, for a particular arm configuration, it also provides a means for the user to select a unique solution. For example, in the six axis PUMA–like robot arm, a consistent result can be obtained by the assistance of three configuration indicators (ARM, ELBOW, and WRIST). These indicators identify the various arm configurations and must be prespecified by the user for finding the inverse solution. There are two steps to find the inverse solution with this technique. The first three joints (joint 1, 2, and 3) is derived in the first step. Then, in the second step, the last three joint is solved using the result of the first step and certain transformation and orientation matrices. The results are four possible solutions to the first three joints, and for each of these four solutions there are two possible solutions to the last three joints. One can choose a solution from the eight possible solutions by determining the ARM, ELBOW, and WRIST indicator.

1.2.2 Numerical Solution

Numerical solutions are commonly used if the closed form solution does not exist. This can be applied either for non-redundant manipulators ($m = n$) which do not have closed form solutions or for redundant manipulators which have redundant degrees of freedom ($m < n$). Numerical solutions are usually based on iterative methods. For example, Whitney [27] used Newton Raphson method to solve Eq. (1.2). By differentiating Eq. (1.1), we have

$$\frac{d\mathbf{x}}{d\mathbf{q}} = J(\mathbf{q}) \tag{1.3}$$

where $J(\mathbf{q})$ is the Jacobian of f with respect to $\mathbf{q}$.

If we rewrite Eq. (1.3) as

$$\Delta \mathbf{x} \simeq J(\mathbf{q})\, \Delta \mathbf{q} \tag{1.4}$$

then wherever J^{-1} exist, we may write

$$\Delta \mathbf{q} = J^{-1}\, \Delta \mathbf{x}. \tag{1.5}$$

Given initial values, $\mathbf{q}_0$ and $\mathbf{x}_0$, we can calculate the final value $\mathbf{q}_f$, by means of Newton-Raphson method [26].

Khosla et al. [10] have also solved the inverse kinematics problem of the CYRO robot (six degrees of freedom) which is specially designed for seam tracking applications. Their work is based on the T_6 and dT_6 [1] matrices and the assumption that three-dimensional seam data are available. Even though they used a different approach, the resulting algorithm still constituted iterative Newton-Raphson method. Because of their iterative nature, the numerical solutions are generally much slower than the corresponding closed form solutions, and convergence is not guaranteed.

1.2.3 Generalized Inverse Technique

To overcome the drawbacks encountered in solving Eq. (1.1) in terms of $\mathbf{q}$, a method called Generalized Inverse Technique is introduced [27].

[1] T_6 is the matrix representation of $f(\mathbf{q})$ in Eq. (1.1) and dT_6 is the differential change matrix.

In generalized inverse technique, we deal with the differential motion relationship between $\mathrm{x}(t)$ and $\mathrm{q}(t)$, that is

$$\dot{\mathrm{x}} \quad = \quad J(\mathrm{q})\,\dot{\mathrm{q}} \tag{1.6}$$

where $\dot{\mathrm{q}}$ and $\dot{\mathrm{x}}$ are joint velocities and end–effector velocity respectively, and $J(\mathrm{q})$ is the m x n Jacobian matrix.

If J is square matrix and nonsingular, then J^{-1} exists. $\dot{\mathrm{q}}$ can be computed as

$$\dot{\mathrm{q}} \quad = \quad J^{-1}\,\dot{\mathrm{x}}. \tag{1.7}$$

For redundant linkages, J is not a square matrix, but a rectangular ($m < n$) matrix and for each velocity vector $\dot{\mathrm{x}}$ there exist various joint velocities $\dot{\mathrm{q}}$ which satisfy Eq. (1.6). If J is full rank (rank $= m$), then there exists a pseudoinverse, J^{+}, such that $\dot{\mathrm{q}}$ can be calculated as

$$\dot{\mathrm{q}} \quad = \quad J^{+}(\mathrm{q})\,\dot{\mathrm{x}}. \tag{1.8}$$

J^{+} is the Moore-Penrose pseudoinverse [11] given by

$$J^{+} \quad = \quad J^{T}\,(JJ^{T})^{-1}. \tag{1.9}$$

This method has a drawback that the solution does not generate joint trajectories which avoid singular configurations [2] in any practical sense [1]. This is because for almost any point x_0 in the workspace and any point $\mathbf{q}^*$ in a neighborhood of a singular

[2]Singular configurations happen if at points $\mathbf{q}_s$ in the joint space, the Jacobian J is rank deficient. These are the points where infinite $\dot{\mathrm{q}}$ could be chosen in order to maintain a given finite $\dot{\mathrm{x}}$.

configuration, there is an initial configuration $\mathbf{q}(0) = \mathbf{q}_0$ and a workspace path $\mathrm{x}(t)$, with $\mathrm{x}(0) = \mathrm{x}_0$, such that the trajectory $\mathbf{q}(t)$ as defined by Eq. (1.8) passes through $\mathbf{q}^*$.

Without further refinement this method cannot be used as manipulator control. As an alternative [1], Eq (1.8) is modified to

$$\dot{\mathrm{q}} \quad = \quad J^+\dot{\mathrm{x}} \, + \, (I - J^+J)\, \mathrm{v} \tag{1.10}$$

where v is a (time varying) vector of the same dimension as $\mathbf{q}$ which remains to be specified. With this modification one can show that by appropriate choice of v, the trajectory which avoid singular configuration may be generated.

1.2.4 Artificial Neural Network Solution

There are two classes of neural network applications that can solve the inverse kinematics problem : optimization and pattern association (see chapter 2).

Hopfield "Optimization" Neural Network Solution

Guo and Cherkassky [6] solved the inverse kinematics problem for redundant manipulator using Hopfield "optimization" neural network. They first introduced an energy function E as

$$E \quad = \quad \frac{1}{2} \sum_{i=1}^{m} \left(\dot{x}_i^d - \dot{x}_i\right)^2 \tag{1.11}$$

where $\dot{x}_i^d$ is the desired end–effector velocity, and then rewrote Eq. (1.6) as

$$\dot{x}_i = \sum_{j=1}^{n} J_{ij}\dot{q}_j \tag{1.12}$$

where $J = [J_{ij}]$, $m < n$ and $i = 1, \ldots, m$.

After substituting Eq. (1.12) into Eq. (1.11) and rearranging the summation order, they obtained a Liapunov energy function, from which the fixed weights and external inputs in terms of J_{ij} and $\dot{x}_i^d$ could be obtained. Finally they derived the dynamics equation of neuron state of the Hopfield net in which joint velocities $\dot{\mathbf{q}}$ came out as its neuron state (output).

For a chosen task trajectory, their solution algorithm [3] can be cast into the following steps.

1. With the given initial condition of joint angles $\mathbf{q}$, compute the weights.

2. Differentiate the task trajectory to obtain the desired end–effector velocity.

3. Input the network with the sampled value of the desired end–effector velocity (over a sampling time T) and compute the external inputs.

4. Iterate, using Eq. (2.1) and Eq. (2.2), until convergence. Convergence means that the neuron state (joint velocities) does not change for further iteration or the process finds a local minimum of energy function E.

[3]They simulated a 4–link manipulator moving along a straight line path with constant velocity in the vertical direction in a 2–dimensional (x, y) space.

5. Sample the neuron state obtained from step 4 over a sampling time T
6. Use this values as the joint velocity command for the redundant manipulator.
7. Integrate the neuron state and update the weights.
8. Go to step 3.

This method seems quite complicated and has the following drawbacks :

1. Iteration in step 4 must be done for each desired input.
2. Output must be integrated to give joint position q.
3. Weights and external inputs must be updated.

Pattern Association using Multilayer Perceptron

In this thesis, the ANN used is of the supervised type—multilayer feed–forward neural networks with back error propagation (BEP) training algorithm. The network is trained to associate input vector or pattern to output vector. In this special case, the input–output relationship of the pattern is a nonlinear mapping described by Eq. (1.2).

The main idea of the BEP training algorithm is to minimize a "computation" energy E which is denoted by

$$E = \frac{1}{2} \sum_{p} \sum_{j=1}^{m} (X_j^d - X_{j,n})^2 \tag{1.13}$$

where :

p	is the number of training patterns,
m	is the number of nodes in the output layer,
n	is the total number of layers in the network,
X_j^d	is the desired output pattern or the desired joint coordinates,
$X_{j,n}$	is the actual pattern or output of the network.

The local minimum of E can be achieved if

$$\frac{\partial E}{\partial W} = 0 \tag{1.14}$$

where W is the weights of the network.

Using optimization technique, local (optimal) minimum point can be reached by iteration method called "gradient descent" or delta rule [23] which can be written as

$$W(t+1) = W(t) + \Delta W \tag{1.15}$$

where t indicate iteration time, and

$$\Delta W = -\eta \frac{\partial E}{\partial W} \tag{1.16}$$

where η is positive step size or gain term $(0 < \eta < 1)$, and ΔW is gradient or the change that must be made to the weights after each iteration. Hence, during training, the weights are updated until local minimum of E is found. The term Back Error

Propagation arises because a mechanism where the error [4] must be propagated backward through the network is introduced [23] in the derivation of this learning procedure.

After training, the network should provide a proper response when a command input vector that it does not know is given. If the training set is properly chosen, the network will be able to generalize its knowledge to situations different from those encountered during training. Inputs to a properly trained network will produce network output that are very close to the corresponding actual joint positions of the redundant manipulator.

By introducing a simple concept called "computation" energy, we can adopt the multilayer feed–forward neural network to solve a difficult problem such as the redundant robot inverse kinematics problem. This thesis present a multilayer feed–forward neural network approach to solve the redundant robot inverse kinematics problem.

In the following chapters, we will discuss neural networks with the emphasis on multilayer feed–forward neural network and its learning algorithm. Chapter 3 will explain our simulation model. Chapter 4 describes the simulation results. Finally, the conclusion of this thesis will be presented in chapter 5. Note that the detailed derivation of the back error propagation learning scheme and program listing are available in appendices A and C.

[4] the derivative of E with respect to $X_{i,l}$ (the output of all the nodes in each layer).

Chapter 2

Background on Neural Networks

2.1 Introduction

Artificial Neural Networks (ANN) are an endeavor to model the function and architecture of neural networks found in the human brain. These models are composed of many nonlinear computational elements or nodes operating in parallel and arranged in pattern reminiscent of biological neural networks.

The basic operation of a single artificial neuron is shown in Fig. 2.1.a. After summing N weighted inputs, the result is passed through a nonlinearity f (Fig. 2.1.(b)-(d) show different types of f). If we relate this figure to a single biological neuron then inputs can be considered as stimulation levels and weights as synaptic strengths. This fundamental building block for all ANN is characterized by the type of nonlinearity f and internal threshold or bias or external input θ which determines the node output whenever the sum of the weighted input is zero.

In general, a neural network consists of N nodes (processors), each of which is connected to all the other nodes. One can compose these nodes into infinitely many network configurations, for example, Fractal Neural Networks [2], Cyclic Neural Networks [5], and others ([18], [3], [19]). However, the underlying principle of these ANN are based on 6 major artificial neural networks [17]. They are Hopfield Net, Hamming Net, Carpenter–Grossberg Net, Perceptron, Multilayer Perceptron, and Kohonen Net.

This thesis will discuss only the multilayer perceptron.

The structure or model of a network explains the computation (algorithmic) process happen inside the network. This process, in general, can be divided into two phase, retrieving phase and learning phase [16].

2.1.1 Retrieving Phase

In response to input vectors or patterns (e.g., $X_1(0), X_2(0), \cdots X_{N_0}(0)$ in Fig. 2.2), the retrieving phase performs the propagative updating of output values of each neuron based on the predefined rule in each network model to produce the responding outputs.

The rule for the retrieving phase of an ANN model can be written as

$$Net_i(l+1) = \sum_{j=1}^{N_l} W_{i,j}(l+1) X_j(l) \tag{2.1}$$

$$X_i(l+1) = f_i(Net_i(l+1), \theta_i(l+1)) \tag{2.2}$$

where $1 \leq i \leq N_l + 1$ and $0 \leq l \leq L-1$. The index l can represent either time or spatial iterations. If l represents spatial iteration then l indicates layer number, L is the total number of layers in the network, i indicates a node number in each layer and N_l is the total number of nodes in layer l. If l represents time iteration then l is the iteration counter, L is the total number of iterations, i indicates node number in a layer, and N_l is the total number of nodes in the layer l.

Two types of inputs can be used to represent the test patterns: the stimulus inputs $X_i(0)$ and the external inputs $\theta_i(l)$.

2.1.2 Learning Phase

The learning phase performs the iterative updating of the synaptic weights for all the connections based on the input and/or target training patterns. The weight updating problem is to find a set of connection weights so as to optimize certain predefined mathematical quantity E based on a set of training patterns.

Typically, there are two steps involved in the learning phase. In the first step, the input training patterns are processed by the network based on the retrieving phase equations to generate some actual responses. In the second step, the weights are changed according to the generated actual responses and the learning rules used.

In the learning rule aspects, many algorithms have been proposed to improve the training process (that is, to reduce training time), such as Adaptive Least Squares Algorithm [14], and Ring Systolic Arrays [16].

2.2 Class of Neural Nets Applications

In general ANN's application can be grouped into 2 classes: optimization and pattern association.

2.2.1 Optimization

For this application, the artificial neural networks are utilized as a state-space search mechanism [9]. A set of fixed weights is derived before starting the search process. The derivation of these fixed weights ($W_{i,j}$) usually depend on finding a Liapunov energy function for the specific application. For example, if Hopfield neural network is used for solving an optimization problem, the Liapunov energy function has the following form [8]:

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} W_{i,j}X_iX_j - \sum_{i=1}^{N} X_i\theta_i \tag{2.3}$$

In certain applications , the Liapunov energy function has to be derived from a given cost function and the constraints in the optimization problem [25], [6]. Once the fixed weights have been assigned from the energy function, the system then iterates until a stable state configuration is reached.

It can be proved that if the weights used are symmetric, $W_{i,j} = W_{j,i}$, then the iterations following the system dynamics in Eq. (2.1) and Eq. (2.2) will converge to a (local) optimal state.

2.2.2 Pattern Association

Another very promising application of neural processing is for pattern association. In pattern association, the network should give the complete pattern even though it is input by partial information of the desired pattern. In mathematical language, the network should be able to retrieve a corresponding pattern in subset B, given a pattern in subset A (vector mapping). The retrieving phase uses the same system dynamics Eq. (2.1) and Eq. (2.2) as in the optimization applications. For this class of application, some learning schemes are often adopted to train the synaptic weights.

2.3 Robotic Applications

Most of the robotic problems currently being studied can be categorized into either one of the following three processing levels:

1. Task Planning (e.g., scheduling of assembly tasks).

2. Path Planning (e.g., robot navigation).

3. Path Control (e.g., motor control).

Most of these robotic processing can be formulated in terms of optimization or pattern association problems. Therefore neural networks can naturally be adopted to solve the robotic processing tasks.

2.4 Multilayer Feed–forward Neural Networks

In the present work, only multilayer feed–forward network or multilayer perceptron are used. These networks are popular because a sophisticated learning rule exists for training the network. A typical network using multilayered structure is shown in Fig. 2.3. It consist of an input layer, output layer, and hidden layers in between. The nodes in the hidden layer are necessary to implement nonlinear mappings between the input and output patterns.

2.4.1 Terminology

By refering to Fig. 2.3, we will use the following terminology for the rest of this thesis.

Let

$\theta_{i,l}$ be the threshold of the i^{th} node in the l^{th} layer,

$W_{i,j,l}$ be the weight between j^{th} node in the $(l-1)^{th}$ layer to i^{th} node in l^{th} layer,

$Net_{i,l}$ be the input to node i^{th} in l^{th} layer, and

$X_{i,l}$ be the output of i^{th} node in l^{th} layer.

The input to a node is given by

$$Net_{i,l} = \sum_j (W_{i,j,l} X_{j,l-1}) + \theta_{i,l} \tag{2.4}$$

where the summation index j extends over all nodes in the $(l-1)^{th}$ layer.

The output of a node is given by

$$X_{i,l} = f(Net_{i,l}). \tag{2.5}$$

In general, f is a nondecreasing function of the input to the output node.

2.4.2 Nonlinearity

There are several forms of function f we can use. If we use a linear activation function, that is $f = Net_{i,l}$, then the network is called a linear network. Although linear networks are useful for cases where the set of input patterns are linearly independent [12] , it can be shown that a multilayered network with linear nodes can be collapsed into an equivalent two layer network so that the advantages of hidden units is lost.

To enable the network to implement complex nonlinear mappings between input and output patterns, it is necessary that the function f be a nonlinear function of the node input. The most frequently used functions are the unit step (see Fig. 2.1.b) given by

$$f(Net_{i,l}) = \begin{cases} 1 & \text{if } Net_{i,l} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

and sigmoid function (see Fig. 2.1.c) given by

$$f(Net_{i,l}) = \frac{1}{1 + e^{-Net_{i,l}}} \tag{2.7}$$

The sigmoid function is characterized by an area in the vicinity of $Net_{i,l} = 0$ in which the node output increases almost linearly with the input. Outside this area, the node output saturates to a minimum or maximum value (0 or 1). The simplest network using unit step units is the perceptron [17] which has no hidden layer units

Early researchers have recognized that layered networks with nonlinear activation function are able to implement complex tasks [20]. However such networks were not popular because of the lack of systematic learning algorithms. The main difficulty encountered in developing learning algorithms for such system is that the unit step function is nondifferentiable. Instead of using the unit step function, Rumelhart [23] devised a multilayered network with a sigmoid function. Such networks can be trained by a learning algorithm called the generalized delta rule [23].

Modification

In many practical cases, outputs other than 1 may be required. We can modified Eq. (2.7) as follows:

$$f(Net_{i,l}) = \frac{\epsilon_{i,l}}{1 + e^{-\alpha_{i,l} Net_{i,l}}} \tag{2.8}$$

where $\epsilon_{i,l}$ and $\alpha_{i,l}$ are both greater than zero [22].

There are two new parameters, ϵ and α, introduced after the modification. The parameter ϵ determines the maximum output of a node and the value of α governs the size of the input zone beyond which the node output saturates and also the steepness of the sigmoid curve. When ϵ approaches zero, the size of this zone is very large. Consequently, the input–output relationship within this zone is almost linear. In this case, the node output saturates only for very large positive or negative inputs. Thus for the lower value of ϵ, the node behave like a linear unit over most of its input range. As ϵ approaches infinity, the neuron behavior becomes extremely nonlinear and in the limit approaches the unit step function. Since nonlinearities are associated with useful feature extraction abilities, neurons that exhibit a high value of ϵ after the training process can probably be viewed as important feature extractors of the network.

In our application, we need both negative and positive value of the node output, then it is necessary to symmetrically translate the sigmoid function (Eq. (2.8)) downward along the vertical axis so that it becomes

$$f(Net_{i,l}) = \frac{\epsilon_{i,l}}{1 + e^{-\alpha_{i,l} Net_{i,l}}} - \frac{\epsilon_{i,l}}{2} \tag{2.9}$$

as shown in Fig. 2.1.c.

2.4.3 Generalized Delta Rule

The generalized delta rule consists of two steps, the retrieving phase (forward pass) and the parameters adjusting phase (backward pass). In the forward pass, we present an input pattern to the network and calculate the output pattern at the output nodes with the current set of learning parameters. The learning parameters of the network are the weights and the node parameters such as $\theta_{i,l}$, $\epsilon_{i,l}$, and $\alpha_{i,l}$. The network output pattern is then compared to the desired output pattern, and the value of a predefined measure function E is calculated by computing the Euclidean distance between the actual and desired output patterns. This function can be expressed as

$$E = \frac{1}{2} \sum_{j=1}^{m} (X_j^d - X_{j,n})^2 \tag{2.10}$$

where the index j represent summation over all output nodes in the network. The quantity X_j^d is the desired output at the j^{th} output node. The total number of layer in the networks is n.

In the the backward pass, the learning rule tries to drive the value E to zero or close to zero by suitable adjustments of the learning parameters. This essentially constitutes a minimization problem that the delta rule attempts to solve using gradient techniques. The calculation of the gradient of E with respect to the learning parameters is performed by propagating the error [1] backwards through the network and involves simple local computations at nodes in the same layer, permitting parallel operation of all nodes in that layer. The details of the error propagation and gradient calculation are presented in Appendix A. Once the gradient is calculated, the learning parameters are adjusted using gradient descent methods. Rumelhart [23] uses a "steepest descent" procedure that has the advantage of being simple and requires only local calculations during parameters adjustments. The change in the learning parameters is made as follows:

$$\Delta\Omega_j = -\eta \frac{\partial E}{\partial \Omega_j} \tag{2.11}$$

where $\Omega = \{\Omega_1, \Omega_2, \Omega_3, \Omega_4\} = \{W, \epsilon, \alpha, \theta\}$ are the learning parameters, and η is a positive step size or gain term $(0 < \eta < 1)$. As discussed before, the gradient is computed only by local calculation so that the parameters updated by Eq. (2.11) can be done in parallel. After all input–output pairs in the training set have been presented, the total E is calculated. The total E can be expressed as

$$E_{total} = \sum_p E \tag{2.12}$$

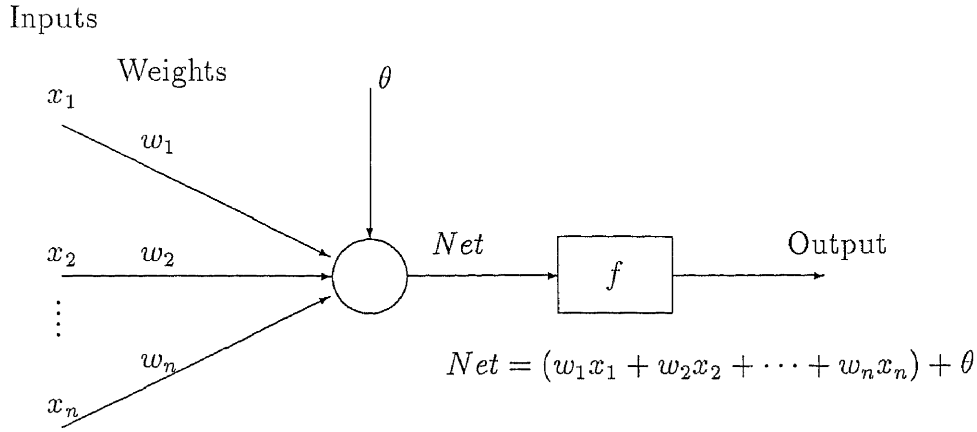
where the index p represent the summation over all patterns used for training. The

[1] the derivative of E with respect to $X_{i,l}$

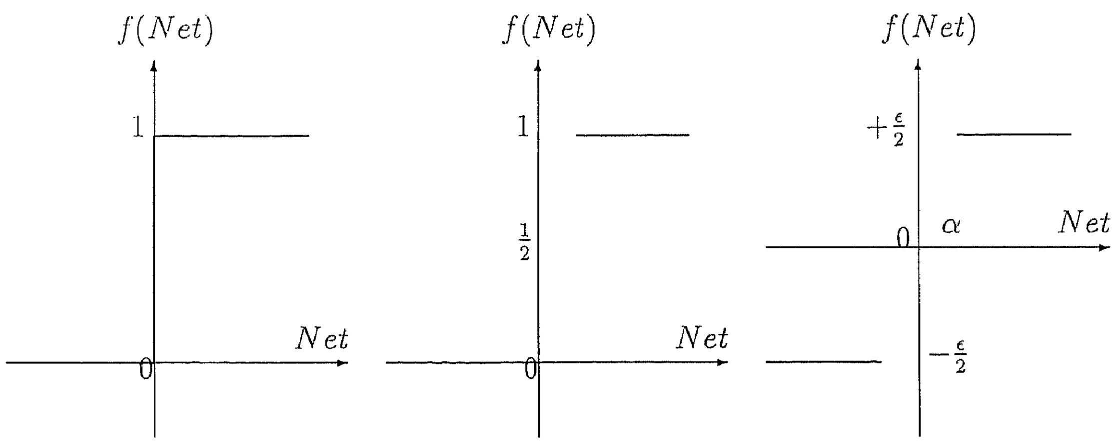
procedure is repeated until the E_{total} reaches a small number which is determined before we start training the network.

In case of a linear system, the E surface is bowl shaped and has only one minimum (at 0) so that convergence is guaranteed. With nonlinear system, however, a large number of local minima may exist. There is no guarantee that the delta rule will converge to the global minimum. However, as pointed out in [23], this apparent drawback is only of theoretical interest because in most cases with the appropriate choice of hidden units in the network structure (which leads to a high degree of redundancy) or by starting with a small number of hidden units and increasing the number until it become to drive the system to a global minimum.

In the present application, for a certain manipulator task trajectory, the network is trained so that it can map every vector in the input domain (consist of patterns or vectors representing end–effector positions) into the output domain (consist of vectors representing joint positions of the manipulator arm) as mathematically expressed by Eq. (1.2).



(a) Basic operation of an artificial neuron



(b) unit step (c) sigmoid (d) modified sigmoid

Figure 2.1: An Artificial Neuron.

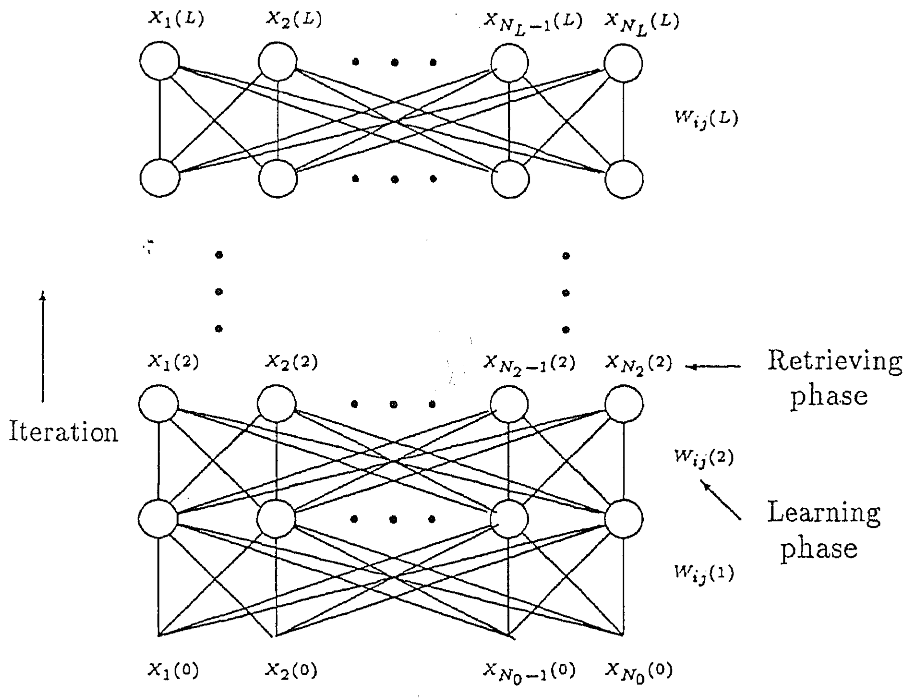


Figure 2.2: An Artificial Neural Network

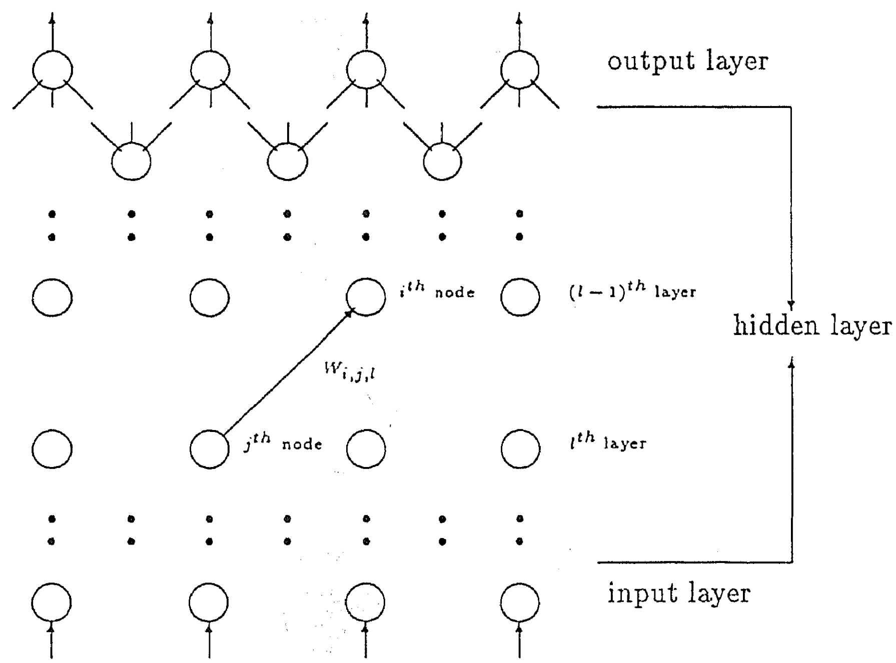


Figure 2.3: Terminology of feed-forward neural nets

Chapter 3

Manipulator Model and Neural Network Model

3.1 Manipulator Model

For the purpose of simulation, a 3 joint–links manipulator is used as the manipulator model. The length of each link is one third of a unit length. The manipulator trajectory used as follows :

A. a straight line in $z = 0$ plane as shown in Fig. 3.1,

B. a straight line as shown in Fig. 3.2, and

C. a straight line as shown in Fig. 3.3.

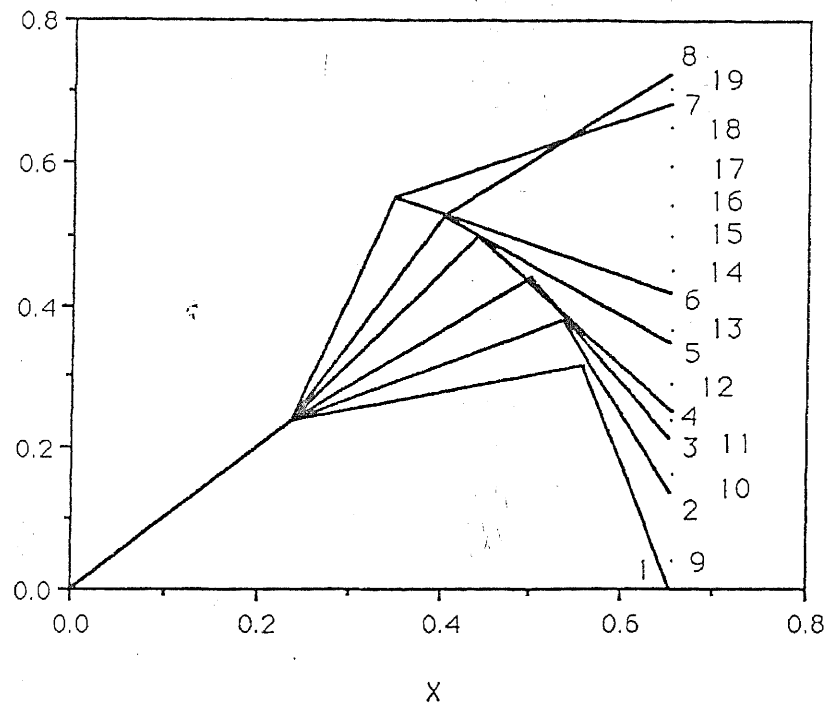


Figure 3.1: Simulation A

3.2 Neural Network Descriptions

Two network configurations will be utilized in this presentation, a network with 48 nodes for simulation A and a network with 52 nodes for simulations B and C. The nodes are specified as follow (see also Fig. 3.4):

- For simulation A, there are six nodes in the output layer; they represent the joint coordinates of the manipulator, (x_1, y_1), (x_2, y_2), and (x_3, y_3). For simulation B and C, there are nine nodes in the output layer; they represent the joint coordinates of the manipulator, (x_1, y_1, z_1), (x_2, y_2, z_2), and (x_3, y_3, z_3).

- Twenty nodes in the second hidden layer for both configurations.

- Twenty nodes in the first hidden layer for both configurations.

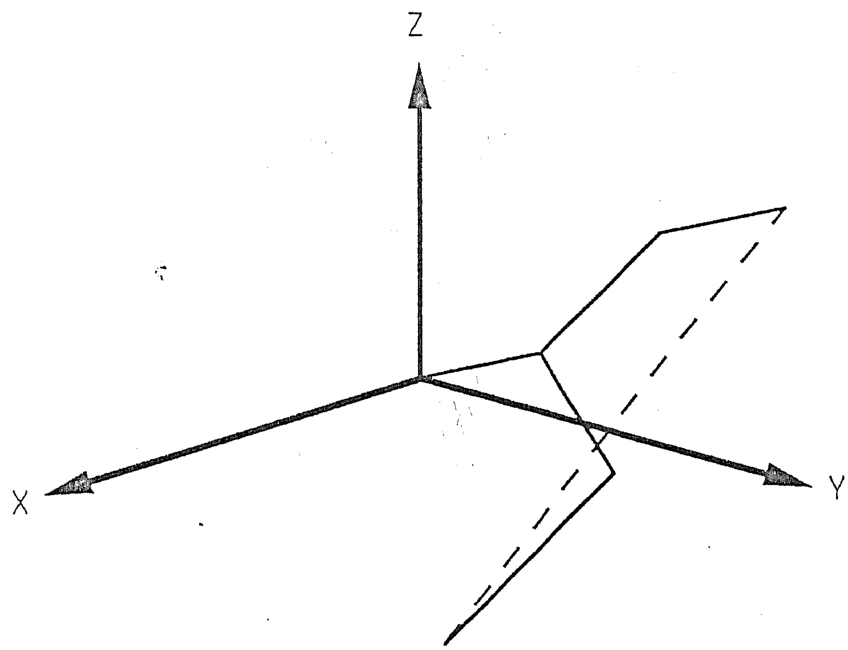


Figure 3.2. Simulation B

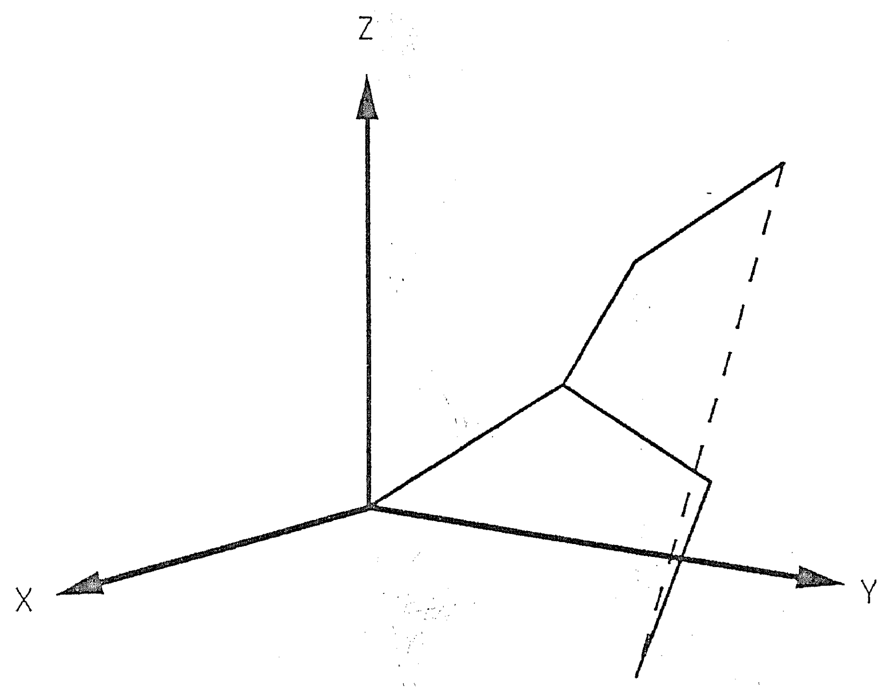


Figure 3.3. Simulation C

- For simulation A, two nodes are located in the input layer which represent either the desired end–effector coordinate during training, (x_3^d, y_3^d), or the input command after training, (x_3, y_3). For simulations B and C, three nodes are located in the input layer representing either the desired end–effector coordinate during training, (x_3^d, y_3^d, z_3^d), or the input command after training, (x_3, y_3, z_3).

The initial values of the learning parameters for this network are set to small random numbers between zero and one. The same initial learning parameters are used for simulations B and C. Table listing the learning parameters before and after training can be found in Appendix B. It should be noted that some weights in the network can be fixed. In this case, error is still propagated as usual but the fixed weights are simply not modified.

The values used for the gain term (η) and the momentum term (κ) are 0.3 and 0.7 respectively.

3.3 Obtaining Training Pattern

The set of training patterns for simulations A, B and C are obtained by the following steps :

1. The robot arm trajectory in two dimensional space is designed as shown in Fig. 3.1.

2. The coordinate of each joint is measured. The result is a set of training patterns for simulation A as shown in Table 3.1.

3. Transforming the result of step 2 into three dimensional space by multiplying it with a rotational matrix [4]

$$R_{\vec{r},\psi} = \begin{bmatrix} r_x^2 V_\psi + C_\psi & r_x r_y V_\psi - r_z S_\psi & r_x r_z V_\psi + r_y S_\psi \\ r_x r_y V_\psi + r_z S_\psi & r_y^2 V_\psi + C_\psi & r_y r_z V_\psi - r_x S_\psi \\ r_x r_z V_\psi - r_y S_\psi & r_y r_z V_\psi + r_x S_\psi & r_z^2 V_\psi + C_\psi \end{bmatrix} \tag{3.1}$$

where $V_\psi = \text{Vers}\,\psi = 1 - \cos\psi$, $\vec{r}$ is orientation vector and ψ is the angle of rotation about $\vec{r}$. For $\vec{r} = (1\,1\,1)^T$ then

$$\begin{aligned} r_x &= \frac{1}{\sqrt{r_x^2 + r_y^2 + r_z^2}} = \frac{1}{\sqrt{3}} \\ r_y &= r_z = \frac{1}{\sqrt{3}} \end{aligned}$$

In other words, we lift the $z = 0$ plane so that it has the same orientation with $\vec{r}$, and then rotate with angle ψ about $\vec{r}$. For simulation B the matrix used is $R_{(1\,1\,1)^T,45^\circ}$. For simulation C the matrix used is $R_{(1\,1\,1)^T,90^\circ}$. The results for simulation B , and simulation C are tabulated in Table 3.2 and Table 3.3.

The synthesized inputs (the inputs which are not included in the training pattern set, positions labeled by number 9 through 19 in Fig. 3.1.) are calculated using the same procedure. The result is tabulated in Table 3.4.

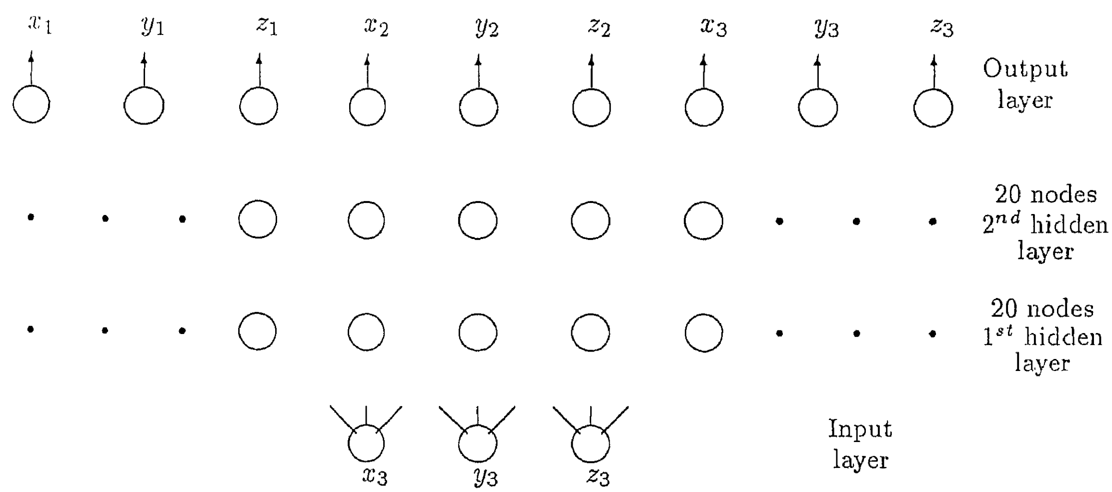


Figure 3.4: Three layers feed–forward neural net

Table 3.1. Original training set.

Pos.	(x_1, y_1, z_1)	(x_2, y_2, z_2)	(x_3, y_3, z_3)
1	(0.236, 0.236, 0.000)	(0.559, 0.317, 0.000)	(0.663, 0.000, 0.000)
2	(0.236, 0.236, 0.000)	(0.535, 0.383, 0.000)	(0.680, 0.083, 0.000)
3	(0.236, 0.236, 0.000)	(0.498, 0.442, 0.000)	(0.687, 0.167, 0.000)
4	(0.236, 0.236, 0.000)	(0.439, 0.500, 0.000)	(0.659, 0.250, 0.000)
5	(0.236, 0.236, 0.000)	(0.400, 0.526, 0.000)	(0.672, 0.333, 0.000)
6	(0.236, 0.236, 0.000)	(0.347, 0.550, 0.000)	(0.652, 0.417, 0.000)
7	(0.236, 0.236, 0.000)	(0.347, 0.550, 0.000)	(0.652, 0.683, 0.000)
8	(0.236, 0.236, 0.000)	(0.401, 0.525, 0.000)	(0.661, 0.733, 0.000)

Table 3.2. Training set A obtained after transforming the original set through matrix $R_{\vec{r},\psi}$, where $\vec{r} = (1\,1\,1)^T$ and $\psi = 45^o$.

Pos.	(x_1, y_1, z_1)	(x_2, y_2, z_2)	(x_3, y_3, z_3)
1	(0.116, 0.309, 0.046)	(0.352, 0.538, -.013)	(0.534, 0.335, -.206)
2	(0.116, 0.309, 0.046)	(0.311, 0.579, 0.028)	(0.521, 0.411, -.169)
3	(0.116, 0.309, 0.046)	(0.263, 0.607, 0.069)	(0.500, 0.481, -.129)
4	(0.116, 0.309, 0.046)	(0.198, 0.624, 0.117)	(0.453, 0.535, -.078)
5	(0.116, 0.309, 0.046)	(0.159, 0.625, 0.142)	(0.437, 0.608, -.040)
6	(0.116, 0.309, 0.046)	(0.108, 0.618, 0.171)	(0.395, 0.665, 0.008)
7	(0.116, 0.309, 0.046)	(0.108, 0.618, 0.171)	(0.313, 0.880, 0.143)
8	(0.116, 0.309, 0.046)	(0.160, 0.625, 0.141)	(0.305, 0.925, 0.166)

Table 3.3. Training set B obtained after transforming the original set through matrix $R_{\vec{r},\psi}$, where $\vec{r} = (1\,1\,1)^T$ and $\psi = 90^o$.

Pos.	(x_1, y_1, z_1)	(x_2, y_2, z_2)	(x_3, y_3, z_3)
1	(0.021, 0.293, 0.157)	(0.109, 0.615, 0.152)	(0.221, 0.604, -.162)
2	(0.021, 0.293, 0.157)	(0.085, 0.615, 0.219)	(0.206, 0.647, -.090)
3	(0.021, 0.293, 0.157)	(0.058, 0.601, 0.281)	(0.188, 0.680, -.016)
4	(0.021, 0.293, 0.157)	(0.024, 0.566, 0.348)	(0.159, 0.684, 0.067)
5	(0.021, 0.293, 0.157)	(0.005, 0.540, 0.381)	(0.143, 0.723, 0.140)
6	(0.021, 0.293, 0.157)	(-.019, 0.499, 0.416)	(0.116, 0.733, 0.220)
7	(0.021, 0.293, 0.157)	(-.019, 0.499, 0.416)	(0.051, 0.822, 0.463)
8	(0.021, 0.293, 0.157)	(0.006, 0.540, 0.380)	(0.042, 0.847, 0.506)

Table 3.4. Synthesized inputs.

Pos. No.	Original (x_3, y_3, z_3)	$R_{(1\,1\,1)^T, 45^o}$ (x_3, y_3, z_3)	$R_{(1\,1\,1)^T, 90^o}$ (x_3, y_3, z_3)
9	(0.667, 0.042, 0.000)	(0.523, 0.371, -.186)	(0.212, 0.621, -.125)
10	(0.667, 0.125, 0.000)	(0.498, 0.438, -.144)	(0.192, 0.649, -.049)
11	(0.667, 0.208, 0.000)	(0.472, 0.505, -.102)	(0.171, 0.676, 0.027)
12	(0.667, 0.292, 0.000)	(0.446, 0.572, -.060)	(0.151, 0.704, 0.103)
13	(0.667, 0.375, 0.000)	(0.420, 0.639, -.017)	(0.131, 0.732, 0.179)
14	(0.667, 0.458, 0.000)	(0.394, 0.706, 0.025)	(0.110, 0.760, 0.255)
15	(0.667, 0.500, 0.000)	(0.381, 0.740, 0.046)	(0.100, 0.774, 0.293)
16	(0.667, 0.542, 0.000)	(0.368, 0.773, 0.067)	(0.090, 0.788, 0.331)
17	(0.667, 0.583, 0.000)	(0.355, 0.807, 0.088)	(0.080, 0.802, 0.369)
18	(0.667, 0.625, 0.000)	(0.342, 0.840, 0.109)	(0.070, 0.815, 0.406)
19	(0.667, 0.708, 0.000)	(0.316, 0.907, 0.151)	(0.049, 0.843, 0.482)

Chapter 4

Simulation Results

In this chapter we will present the simulation results. There are some terms that must be annotated before the graphical and tabular results are presented.

4.1 Physically Realizable Output

When we give the trained ANN a desired end–effector coordinate, $(x_3^d,\ y_3^d)$ in simulation A or $(x_3^d,\ y_3^d,\ z_3^d)$ in simulations B and C, and after the network performs a forward calculation, the outputs will be $(x_1,\ y_1)$, $(x_2,\ y_2)$, and $(x_3,\ y_3)$ in simulation A or $(x_1,\ y_1,\ z_1)$, $(x_2,\ y_2,\ z_2)$, and $(x_3,\ y_3,\ z_3)$ in simulations B and C. Due to inaccuracy in the numerical computation, these outputs may not physically realizable. This means the length of the manipulator's links calculated from the simulation outputs are different from the length of the real manipulator's links. The physically realizable result can be achieved by replacing one of the selected coordinate element with a new calculated value. For example, y_1 in simulation A or z_1 in simulations B and C can be replaced with a value which is calculated by the formula, $\sqrt{l_1^2 - x_1^2}$ for simulation A and $\sqrt{l_1^2 - x_1^2 - y_1^2}$ for simulations B and C. Where l_1 is the length of link number one (the link attached to the base coordinate (0, 0, 0)). By simple calculation we can also determined the length of the others links. Hence, our final results are (x_1, $y_1$phy), (x_2, $y_2$phy), and (x_3, $y_3$phy) for simulation A and (x_1, y_1, $z_1$phy), (x_2, y_2, $z_2$phy), and (x_3, y_3, $z_3$phy) for simulations B and C. In the table this value is listed in the column denoted by $y$phy in simulation A or $z$phy for simulations B and C.

4.2 Error Measurements

The accuracy of the simulation is measured by calculating the error between the desired output and the physically realizable ANN output in the least squared sense, that is

$$Ls.Err. = \sqrt{(x_d - x_n)^2 + (y_d - y\text{phy})^2} \tag{4.1}$$

for simulation A, or

$$Ls.Err. = \sqrt{(x_d - x_n)^2 + (y_d - y_n)^2 + (z_d - z\text{phy})^2} \tag{4.2}$$

for simulations B and C.

The differences, $x_d - x_n$, $y_d - y$phy for simulation A and $x_d - x_n$, $y_d - y_n$, and $z_d - z$phy for simulations B and C, are tabulated in table 4.1 – 4.6.

4.3 Results and Discussion

Fig. 4.1 and Fig. 4.2 respectively plots the error on joint number 2 and joint number 3 for simulation A. Fig. 4.3, Fig 4.4, and Fig 4.5 respectively plots the error on joint number 1, joint number 2, and joint number 3 for simulation B. Fig. 4.6, Fig 4.7, and Fig 4.8 respectively plots the error on joint number 1, joint number 2, and joint number 3 for simulation C. Note that there is no graph for error on joint number 1 in the simulation A, because for this joint, the errors are zero.

Simulation results presented show that the outputs of ANN are close to the desired outputs. The resulting error is in the scale of 0.01, small enough compared to the scale of 1 of the desired values.

It should have been expected from Fig. 3.1 that positions number from 14 to 18 will give a bigger error than from another positions. This is because these position numbers, unlike other positions, is not densely covered by the training positions (patterns). To obtain a reasonably accurate output, the network should be trained by a number of patterns that are sufficient to represent the manipulator task. The more the number of patterns are used for training, the better the result. However, the number of nodes utilized in the network directly corresponds to the number of training patterns used. The number of nodes for the network should be big enough so that the network can adopt the kinematics of the manipulator with a greater degree of accuracy. A question arises from this fact is how many nodes must be used to get the best result. There is no theoretical explanation pertaining to this problem and the best results is usually determined by experiments.

Table 4.1: Result of Simulation A

Pos. No.	Joint No.	Desired Values (xd, yd)	Neural Nets Output (xn, yn)	yphy	Error (x 0.01) xd-xn	yd-yphy	Ls. Err.
1	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.559, 0.317)	(0.555, 0.317)	0.331	0.400	1.457	1.511
	3	(0.663, 0.000)	(0.675, 0.028)	0.020	1.196	2.026	2.353
2	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.535, 0.383)	(0.535, 0.389)	0.383	0.025	0.051	0.056
	3	(0.680, 0.083)	(0.671, 0.080)	0.079	0.846	0.457	0.961
3	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.498, 0.442)	(0.495, 0.449)	0.446	0.325	0.408	0.522
	3	(0.686, 0.167)	(0.669, 0.161)	0.162	1.743	0.512	1.817
4	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.439, 0.500)	(0.444, 0.493)	0.496	0.533	0.418	0.678
	3	(0.659, 0.250)	(0.667, 0.247)	0.248	0.759	0.217	0.790
5	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.400, 0.526)	(0.389, 0.529)	0.532	1.101	0.597	1.253
	3	(0.672, 0.333)	(0.665, 0.336)	0.345	0.698	1.180	1.371
6	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.347, 0.550)	(0.349, 0.552)	0.549	0.205	0.073	0.218
	3	(0.652, 0.417)	(0.664, 0.407)	0.441	1.186	2.450	2.722

Table 4.1: continued

Pos. No.	Joint No.	Desired Values (xd, yd)	Neural Nets Output (xn, yn)	yphy	E r r o r (x 0.01) xd-xn	yd-yphy	Ls. Err.
7	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.347, 0.550)	(0.355, 0.548)	0.547	0.798	0.294	0.851
	3	(0.652, 0.683)	(0.658, 0.677)	0.686	0.567	0.225	0.610
8	1	(0.236, 0.236)	(0.236, 0.236)	0.236	0.000	0.000	0.000
	2	(0.401, 0.525)	(0.393, 0.526)	0.530	0.862	0.476	0.984
	3	(0.661, 0.733)	(0.655, 0.736)	0.735	0.629	0.183	0.655

Table 4.2: Result of Simulation A, Synthesized Input

Pos. No.	Joint No.	Desired Values (xd, yd)	Neural Nets Output (xn, yn)	yphy	E r r o r (x 0.01) xd-xn	yd-yphy	Ls. Err.
9	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.548, 0.353)	0.353	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.042)	(0.673, 0.049)	0.045	0.646	0.299	0.712
10	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.519, 0.418)	0.412	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.125)	(0.670, 0.114)	0.115	0.343	1.023	1.079

Table 4.2: continued

Pos. No.	Joint No.	Desired Values (xd, yd)	Neural Nets Output (xn, yn)	yphy	E r r o r (x 0.01) xd-xn	yd-yphy	Ls. Err.
11	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.471, 0.471)	0.472	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.208)	(0.668, 0.202)	0.203	0.112	0.551	0.563
12	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.416, 0.512)	0.516	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.292)	(0.666, 0.293)	0.296	0.062	0.483	0.487
13	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.367, 0.542)	0.542	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.375)	(0.665, 0.373)	0.392	0.200	1.749	1.761
14	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.333, 0.560)	0.554	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.458)	(0.663, 0.445)	0.508	0.324	4.948	4.959
15	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.324, 0.565)	0.557	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.500)	(0.663, 0.481)	0.557	0.391	5.720	5.734
16	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.320, 0.568)	0.558	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.542)	(0.662, 0.520)	0.558	0.467	1.665	1.730

Table 4.2: continued

Pos. No.	Joint No.	Desired Values (xd, yd)	Neural Nets Output (xn, yn)	yphy	Error (x 0.01) xd-xn	yd-yphy	Ls. Err.
17	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.322, 0.566)	0.558	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.583)	(0.661, 0.563)	0.558	0.559	2.555	2.615
18	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.331, 0.561)	0.555	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.625)	(0.660, 0.611)	0.608	0.676	1.651	1.784
19	1	(xxxxx, xxxxx)	(0.236, 0.236)	0.236	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx)	(0.375, 0.537)	0.539	xxxxx	xxxxx	xxxxx
	3	(0.667, 0.708)	(0.656, 0.712)	0.717	1.018	0.836	1.317

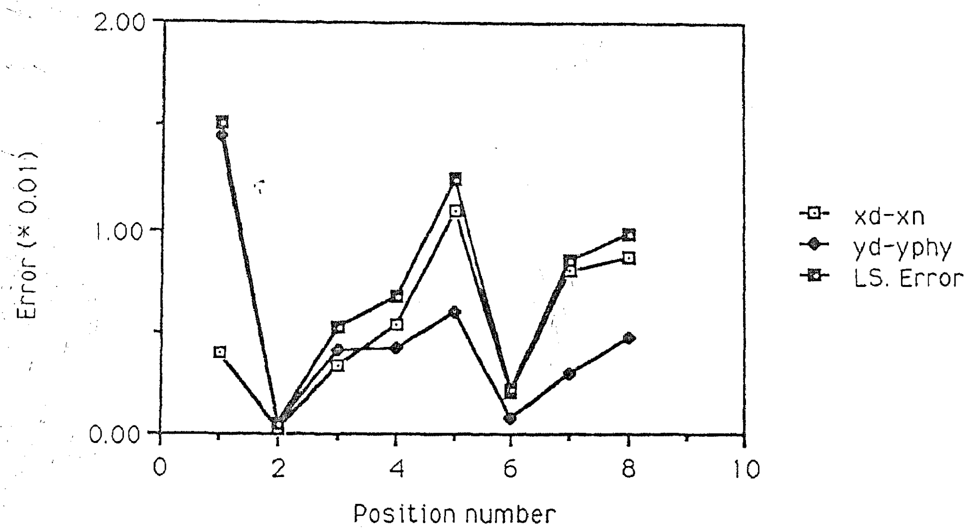


Figure 4.1: Error on joint number 2 (simulation A)

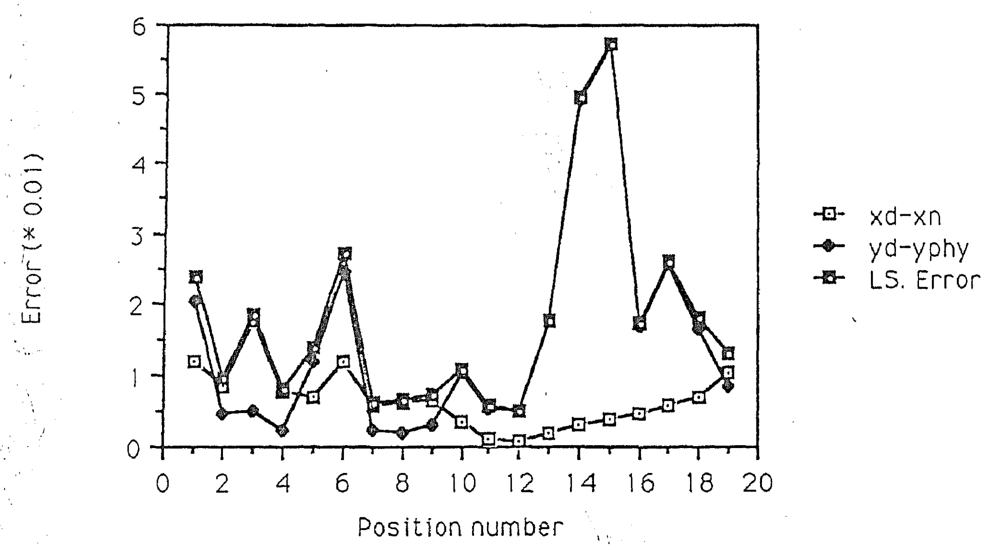


Figure 4.2: Error on joint number 3 (simulation A)

Table 4.3: Result of Simulation B

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
1	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.156)	0.158	0.009	0.024	0.046	0.053
	2	(0.109, 0.615, 0.152)	(0.108, 0.618, 0.153)	0.158	0.084	0.370	0.560	0.676
	3	(0.221, 0.604, -0.162)	(0.222, 0.603, -0.163)	-0.155	0.106	0.073	0.647	0.659
2	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.011	0.000	0.001	0.011
	2	(0.085, 0.615, 0.219)	(0.084, 0.611, 0.218)	0.234	0.077	0.313	1.526	1.559
	3	(0.206, 0.647, -0.090)	(0.205, 0.645, -0.089)	-0.075	0.138	0.155	1.518	1.533
3	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.158)	0.157	0.015	0.011	0.022	0.029
	2	(0.058, 0.601, 0.281)	(0.057, 0.597, 0.284)	0.289	0.124	0.341	0.856	0.930
	3	(0.188, 0.680, -0.016)	(0.185, 0.677, -0.012)	-0.008	0.306	0.360	0.772	0.905
4	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.007	0.002	0.003	0.008
	2	(0.024, 0.566, 0.348)	(0.029, 0.568, 0.344)	0.346	0.479	0.152	0.233	0.554
	3	(0.159, 0.684, 0.067)	(0.163, 0.689, 0.060)	0.066	0.464	0.556	0.068	0.728
5	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.010	0.007	0.011	0.016
	2	(0.005, 0.540, 0.381)	(0.002, 0.540, 0.387)	0.381	0.259	0.019	0.044	0.264
	3	(0.143, 0.723, 0.140)	(0.140, 0.718, 0.145)	0.135	0.286	0.488	0.437	0.714
6	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.029	0.003	0.010	0.031
	2	(-0.019, 0.499, 0.416)	(-0.020, 0.498, 0.417)	0.417	0.151	0.149	0.106	0.237
	3	(0.116, 0.733, 0.220)	(0.117, 0.733, 0.220)	0.225	0.112	0.005	0.475	0.488

Table 4.3: continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
7	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.000	0.003	0.006	0.007
	2	(-0.019, 0.499, 0.416)	(-0.017, 0.502, 0.413)	0.414	0.121	0.262	0.189	0.345
	3	(0.051, 0.822, 0.463)	(0.051, 0.823, 0.462)	0.474	0.049	0.074	1.044	1.048
8	1	(0.021, 0.293, 0.157)	(0.021, 0.293, 0.157)	0.157	0.025	0.000	0.003	0.025
	2	(0.006, 0.540, 0.380)	(0.005, 0.538, 0.382)	0.383	0.078	0.260	0.275	0.387
	3	(0.042, 0.847, 0.506)	(0.041, 0.847, 0.508)	0.503	0.072	0.025	0.316	0.325

Table 4.4: Result of simulation B, Synthesized Input

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xd, yd, zd)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
9	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.097, 0.615, 0.185)	0.201	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.212, 0.621, -0.125)	(0.214, 0.623, -0.128)	-0.111	0.186	0.152	1.335	1.356
10	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.071, 0.602, 0.253)	0.271	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.192, 0.649, -0.049)	(0.195, 0.654, -0.056)	-0.034	0.377	0.484	1.508	1.628

Tabel 4.4: Continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
11	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.043, 0.583, 0.316)	0.321	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.171, 0.677, 0.027)	(0.175, 0.680, 0.022)	0.031	0.312	0.389	0.363	0.617
12	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.015, 0.555, 0.367)	0.364	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.151, 0.704, 0.103)	(0.152, 0.704, 0.102)	0.099	0.068	0.014	0.364	0.371
13	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.009, 0.521, 0.403)	0.399	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.131, 0.732, 0.179)	(0.128, 0.727, 0.184)	0.177	0.248	0.466	0.210	0.568
14	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.028, 0.489, 0.425)	0.422	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.110, 0.760, 0.255)	(0.105, 0.751, 0.264)	0.265	0.522	0.880	1.010	1.438
15	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.033, 0.478, 0.431)	0.429	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.100, 0.774, 0.293)	(0.094, 0.764, 0.303)	0.313	0.614	1.014	1.986	2.313
16	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.036, 0.473, 0.433)	0.432	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.090, 0.788, 0.331)	(0.083, 0.777, 0.341)	0.362	0.663	1.080	3.151	3.397

Table 4.4: Continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
17	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.034, 0.476, 0.432)	0.431	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.080, 0.802, 0.369)	(0.073, 0.791, 0.378)	0.424	0.665	1.067	5.582	5.722
18	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.029, 0.485, 0.426)	0.425	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.070, 0.815, 0.406)	(0.064, 0.806, 0.415)	0.425	0.616	0.967	1.839	2.167
19	1	(xxxxx, xxxxx, xxxxx)	(0.021, 0.293, 0.157)	0.157	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(-0.005, 0.525, 0.396)	0.395	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.049, 0.843, 0.482)	(0.046, 0.838, 0.488)	0.498	0.368	0.502	1.578	1.697

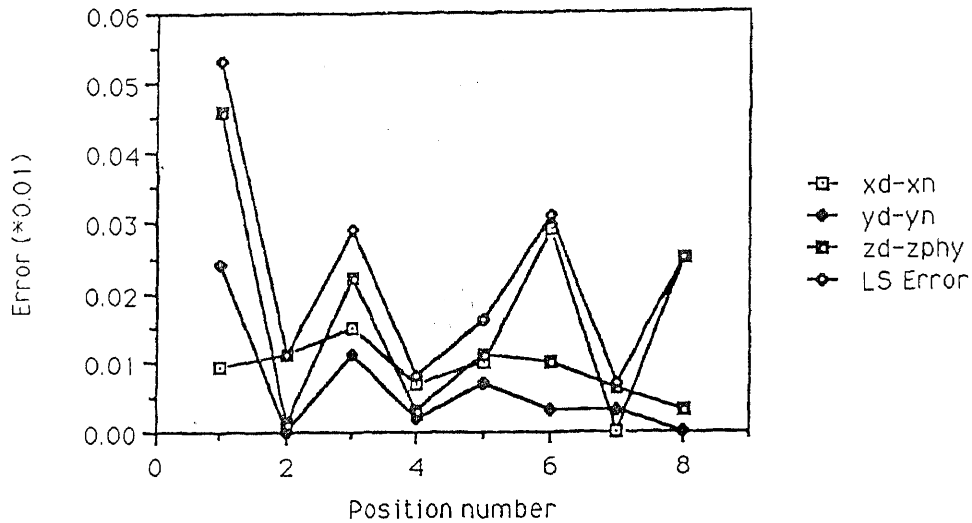


Figure 4.3: Error on joint number 1 (simulation B)

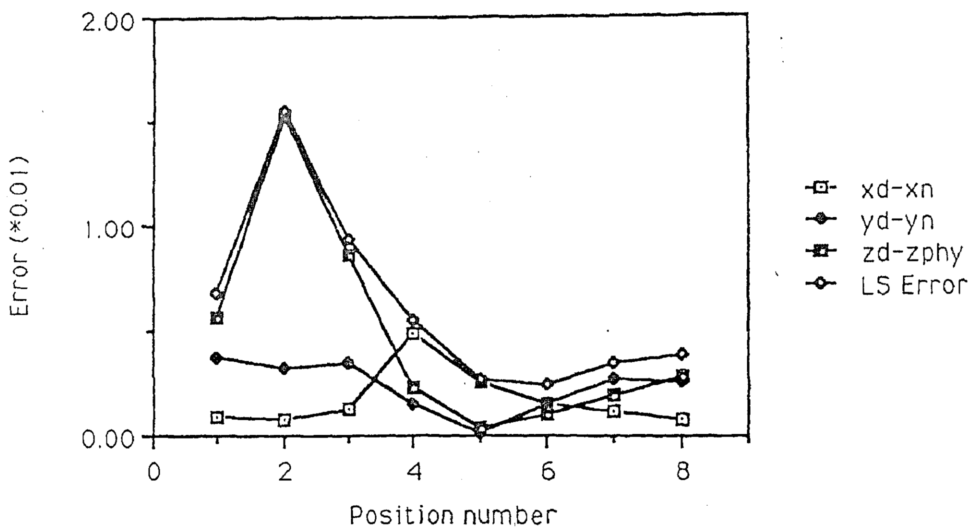


Figure 4.4: Error on joint number 2 (simulation B)

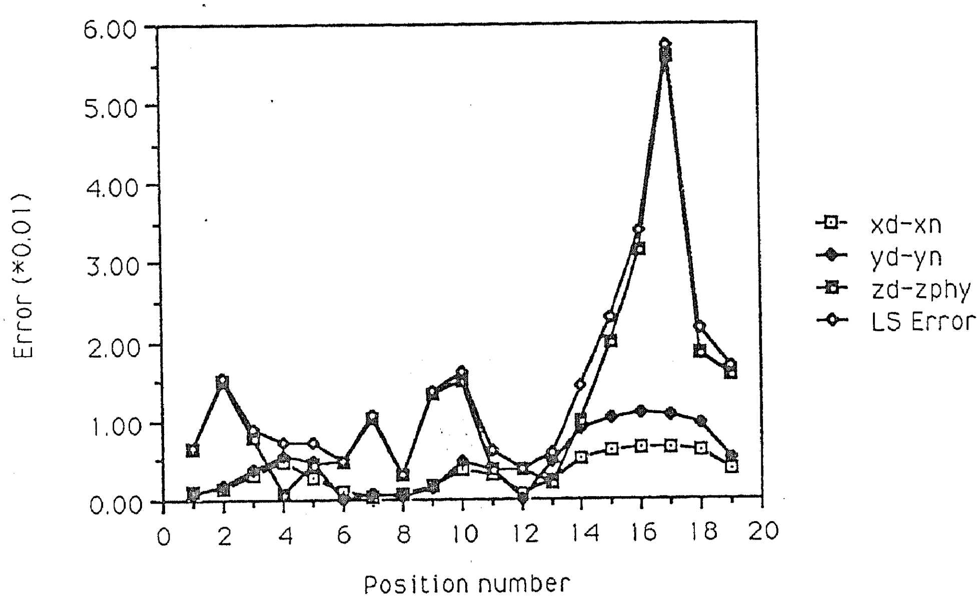


Figure 4.5: Error on joint number 3 (simulation B)

Table 4.5: Result of Simulation C

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
1	1	(0.116, 0.309, 0.046)	(0.116, 0.308, 0.046)	0.052	0.064	0.063	0.553	0.560
	2	(0.352, 0.538, -0.013)	(0.350, 0.536, -0.019)	-0.013	0.114	0.147	0.055	0.194
	3	(0.534, 0.335, -0.206)	(0.533, 0.335, -0.204)	-0.207	0.077	0.002	0.061	0.098
2	1	(0.116, 0.309, 0.046)	(0.117, 0.309, 0.046)	0.045	0.019	0.015	0.152	0.154
	2	(0.311, 0.579, 0.028)	(0.311, 0.583, 0.029)	0.045	0.029	0.414	1.663	1.714
	3	(0.521, 0.411, -0.169)	(0.520, 0.411, -0.169)	-0.149	0.108	0.038	1.973	1.976
3	1	(0.116, 0.309, 0.046)	(0.117, 0.309, 0.046)	0.041	0.059	0.053	0.535	0.540
	2	(0.263, 0.607, 0.069)	(0.265, 0.607, 0.077)	0.072	0.112	0.047	0.283	0.308
	3	(0.500, 0.481, -0.129)	(0.499, 0.481, -0.130)	-0.129	0.119	0.045	0.007	0.127
4	1	(0.116, 0.309, 0.046)	(0.116, 0.309, 0.046)	0.048	0.020	0.018	0.167	0.169
	2	(0.198, 0.624, 0.117)	(0.195, 0.618, 0.121)	0.144	0.329	0.641	2.786	2.878
	3	(0.453, 0.535, -0.078)	(0.455, 0.536, -0.079)	-0.046	0.251	0.080	3.241	3.252
5	1	(0.116, 0.309, 0.046)	(0.117, 0.309, 0.046)	0.043	0.031	0.030	0.289	0.292
	2	(0.159, 0.625, 0.142)	(0.160, 0.621, 0.140)	0.152	0.123	0.422	1.059	1.147
	3	(0.437, 0.608, -0.040)	(0.436, 0.609, -0.040)	-0.034	0.171	0.017	0.582	0.607
6	1	(0.116, 0.309, 0.046)	(0.116, 0.309, 0.046)	0.048	0.029	0.024	0.232	0.235
	2	(0.108, 0.618, 0.171)	(0.105, 0.623, 0.153)	0.160	0.286	0.465	1.083	1.213
	3	(0.395, 0.665, 0.008)	(0.394, 0.666, 0.010)	-0.001	0.109	0.048	0.884	0.892

Table 4.5: Continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
7	1	(0.116, 0.309, 0.046)	(0.116, 0.309, 0.046)	0.047	0.016	0.015	0.139	0.141
	2	(0.108, 0.618, 0.171)	(0.110, 0.623, 0.163)	0.157	0.210	0.535	1.309	1.429
	3	(0.313, 0.880, 0.143)	(0.313, 0.879, 0.143)	0.089	0.007	0.055	5.413	5.414
8	1	(0.116, 0.309, 0.046)	(0.117, 0.309, 0.046)	0.045	0.009	0.005	0.055	0.056
	2	(0.160, 0.625, 0.141)	(0.159, 0.623, 0.163)	0.148	0.111	0.212	0.690	0.730
	3	(0.305, 0.925, 0.166)	(0.305, 0.925, 0.165)	0.148	0.078	0.071	1.767	1.770

Table 4.6: Result of Simulation C, Synthesized Input

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
9	1	(xxxxx, xxxxx, xxxxx)	(0.116, 0.309, 0.046)	0.049	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.328, 0.565, 0.007)	0.026	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.524, 0.371, -0.186)	(0.524, 0.371, -0.185)	-0.161	0.035	0.007	2.503	2.503
10	1	(xxxxx, xxxxx, xxxxx)	(0.116, 0.309, 0.046)	0.047	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.276, 0.600, 0.060)	0.078	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.498, 0.438, -0.144)	(0.500, 0.438, -0.145)	-0.109	0.229	0.018	3.493	3.500

Table 4.6: Continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
11	1	(xxxxx, xxxxx, xxxxx)	(0.116, 0.309, 0.046)	0.046	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.225, 0.614, 0.103)	0.124	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.472, 0.505, -0.102)	(0.474, 0.505, -0.103)	-0.069	0.205	0.043	3.289	3.296
12	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.045	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.177, 0.620, 0.132)	0.149	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.446, 0.572, -0.060)	(0.446, 0.572, -0.060)	-0.041	0.033	0.048	1.822	1.823
13	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.045	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.136, 0.622, 0.148)	0.158	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.420, 0.639, -0.017)	(0.418, 0.639, -0.016)	-0.019	0.207	0.024	0.142	0.252
14	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.106, 0.623, 0.156)	0.156	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.394, 0.706, 0.025)	(0.390, 0.706, 0.028)	0.002	0.428	0.021	2.275	2.315
15	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.098, 0.623, 0.158)	0.154	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.381, 0.740, 0.046)	(0.376, 0.739, 0.049)	0.013	0.505	0.042	3.312	3.350
16	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.094, 0.623, 0.160)	0.153	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.368, 0.773, 0.067)	(0.363, 0.773, 0.071)	0.024	0.546	0.057	4.278	4.313

Table 4.6: Continued

Pos. No.	Joint No.	Desired Values (xd, yd, zd)	Neural Nets Output (xn, yn, zn)	zphy	Error (x 0.01) xd-xn	yd-yn	zd-zphy	Ls. Err.
17	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.097, 0.623, 0.161)	0.154	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.355, 0.807, 0.088)	(0.350, 0.806, 0.092)	0.037	0.541	0.060	5.133	5.162
18	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.106, 0.623, 0.162)	0.155	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.342, 0.840, 0.109)	(0.338, 0.840, 0.112)	0.051	0.483	0.044	5.816	5.836
19	1	(xxxxx, xxxxx, xxxxx)	(0.117, 0.309, 0.046)	0.044	xxxxx	xxxxx	xxxxx	xxxxx
	2	(xxxxx, xxxxx, xxxxx)	(0.148, 0.623, 0.163)	0.151	xxxxx	xxxxx	xxxxx	xxxxx
	3	(0.316, 0.907, 0.151)	(0.315, 0.908, 0.152)	0.101	0.180	0.058	5.064	5.068

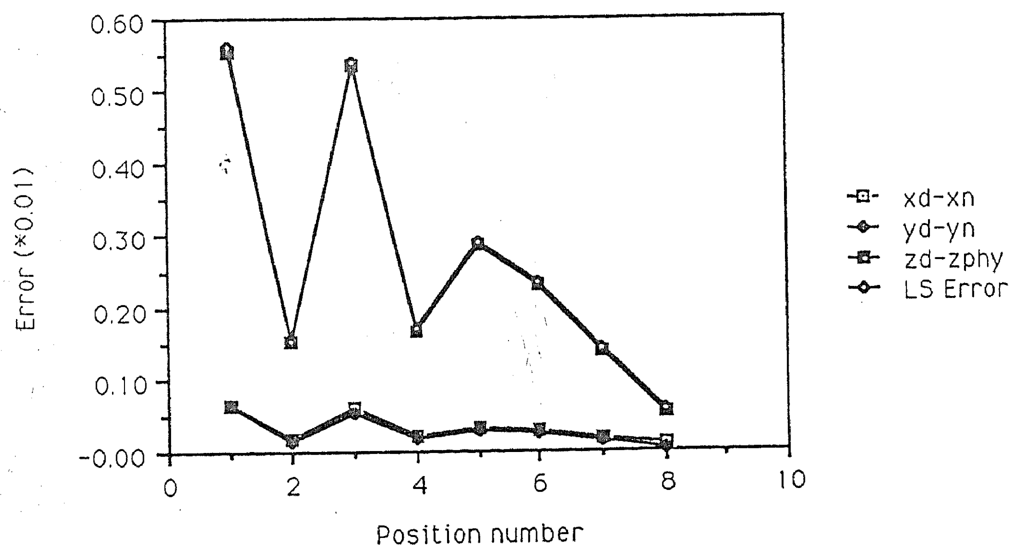


Figure 4.6: Error on joint number 1 (simulation C)

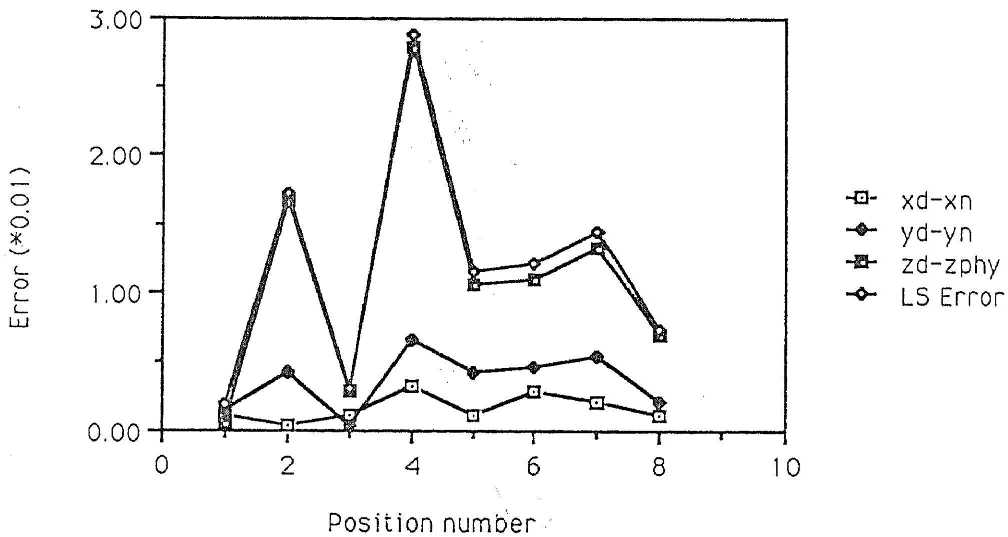


Figure 4.7: Error on joint number 2 (simulation C)

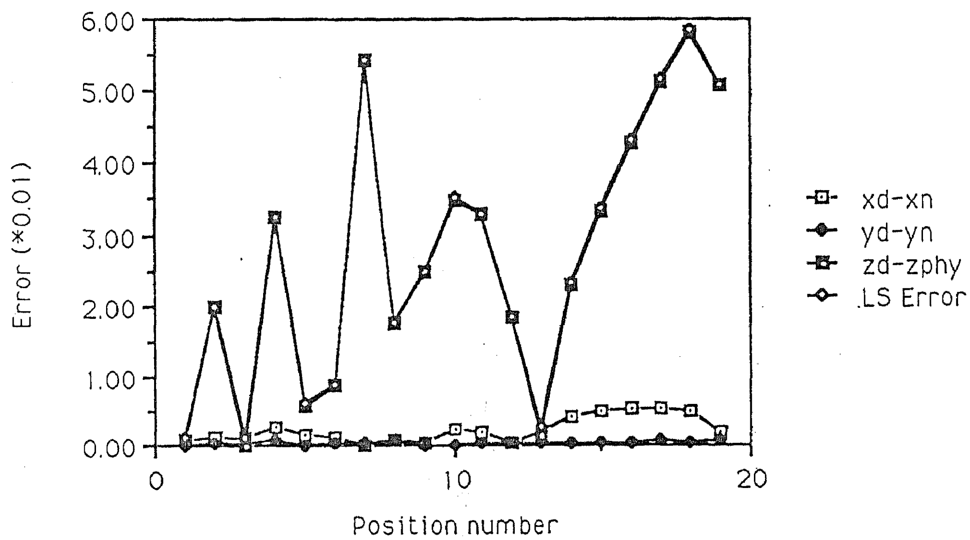


Figure 4.8: Error on joint number 3 (simulation C)

Chapter 5

Conclusion

As mentioned earlier in chapter 1, the inverse kinematics problem of most redundant robots does not have a closed form solution. The difficulties lie in the redudancy of the manipulator, and various approaches have been taken to solve this problem. Artificial neural network approach is presented in this thesis. This method according to our opinion, is the easiest and simpliest approach to the inverse kinematics problem.

This thesis demonstrates how neural networks can be used to solve the inverse kinematics problem of the redundant robot. Two kinds of neural nets can be utilized, Hopfield neural net and multilayer feed–forward net, the latter is our approach to the problem. Both approaches introduced a simple concept of a "computation" energy function and then minimized it. Different minimization algorithms are used because of their different network architectures. Unfortunately, we could not compare our results to Guo and Cherkassky's [6] results because our manipulator model and our input command control are different from theirs.

Finally, the success of neural networks methods can only be proven and tested in the real time applications when the neural network hardware technology has been established.

Bibliography

[1] J. Baillieul, "Kinematic Programming Alternatives for Redundant Manipulators," *IEEE Int'l Conf. on Robotics and Automation*, pp. 722-728, 1985.

[2] Y. Baram, "Associative Memory in Fractal Neural Networks," *IEEE Trans. Systems, Man and Cybernetics*, Vol. 19, No. 5, pp. 1133-1141, 1989.

[3] L. O. Chua and Lin Yang, "Cellular Neural Networks: Theory," *IEEE Trans. Circuits and Systems*, Vol. 35, No. 10, pp. 1257-1272, Oct 1988.

[4] K. S. Fu et al., *Robotics: Control, Sensing, Vision, and Intelligence,* New York, NY: McGraw-Hill, 1987.

[5] M. T. Gately, "CYCLES: A Simulation Tool for Studying Cyclic Neural Networks," *Proc. IEEE Conf. on Neural Information Processing systems–Natural and Synthetic* (Denver, CO, Nov 1987).

[6] J. Guo and V. Cherkassky, "A Solution to the Inverse Kinematic Problem in Robotics Using Neural Network Processing," *Proc. of Int'l Joint Conf. on Neural Networks*, (Washington, D.C., June 18-22 1989), pp. II–299–304.

[7] J. J. Hopfield, "Neural Networks are Physical Systems with Emergent Collective Computation Abilities," *Proc. of the National Academy of Science USA*, Vol. 79, pp. 2554-2558, 1982.

[8] J. J. Hopfield, "Neuron with Graded Response Have Collective Computational Properties Like those Two-state Neurons," *Proc. of the National Academy of Science USA*, Vol. 81, pp. 3088-3092, 1984.

[9] J. J. Hopfield and D. W. Tank, "Computing with Neural Circuits: A Model," *Science,* Vol.233, pp. 625-633, August 1986.

[10] P. K. Khosla et al., "An Algorithm for Seam Tracking Applications," *The Int'l Journal of Robotics Research*, Vol. 4, No. 1, pp. 27-41, Spring 1985.

[11] C. A. Klein and C. H. Huang, "Review for Pseudoinverse Control for Use with Kinematically Redundant Manipulators," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 13, No. 3, pp. 245-250, Mar/Apr 1983.

[12] T. Kohonen, *Assosiative Memory: A System Theoretical Approach,* New York, NY: Springer-Verlag, 1977.

[13] T. Kohonen, *Self Organization and Associative Memory,* New York, NY: Springer-Verlag, 1984.

[14] S. Kollias and D. Anastassiou, "An Adaptive Least Squares Algorithm for the Efficient Training of Artificial Neural Networks," *IEEE Trans. Circuits and Systems*, Vol. 36, No. 8, pp. 1092-1101, August 1989.

[15] S. Y. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[16] S. Y. Kung and J. N. Hwang, " Neural Network Architectures for Robotic Applications," *IEEE Trans. Robotics and Automation*, Vol. 5, pp. 641-657, Oct 1989.

[17] R. P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Mag.*, Vol. 4, pp. 4-22, Apr 1987.

[18] R. J. Marks et al., " Alternating Projection Neural Networks," *IEEE Trans. Circuits and Systems*, Vol. 36, No. 6, pp. 846-857, June 1989.

[19] W. T. Miller, "Real-Time Application of Neural Networks for Sensor-Based Control of Robots with Vision," *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. 19, No. 4, pp. 825-831, 1989.

[20] N. J. Nilsson, *Learning Machines: Foundations of Trainable Pattern Classifying Systems,* New York, NY: McGraw-Hill, 1965.

[21] R.P. Paul, *Robot Manipulators: Mathematics, Programming, and Control,* Cambridge: MIT Press, 1981.

[22] S. S. Rangwala and D. A. Dornfeld, "Learning and Optimization of Machining Operations," *IEEE Trans. Systems, Man, and Cybernetics* ,Vol. 19 No. 2, 1989.

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

[24] T. Shamir, "The Singularities of Redundant Robot Arms," *The International Journal of Robotics Research*, Vol. 9, No. 1, pp. 113-121, Feb 1990.

[25] D. W. Tank and J. J. Hopfield (1986), "Simple "Neural" Optimization Networks: An A/D Converter, Signal Decision Circuit, and a Linear Programming Circuit,"*IEEE Trans. Circuits and Systems*, Vol. CAS-33, No. 5, pp. 533-541, May 1986.

[26] D. E. Whitney, "Optimum Stepsize Control for Newton-Raphson Solution of Nonlinear Vector Equations," *IEEE Transaction on Automatic Control,* Vol. AC-14, No. 5, pp. 572-574, Oct 1969.

[27] D. E. Whitney, "The Mathematics of Coordinated Control of Prosthetic Arms and Manipulators," *Trans. ASME J. Dynamics, Measurements, and Controls*, Vol. 94, pp. 303-309, Dec 1972.

Appendix A

This appendix derives the Back Error Training Algorithm. Refer to section 2.4 and Fig. 2.3 for further explanation on the terminologies used.

Since we use sigmoid function as the squashing function, the input–output relationship of each node can be expressed by

$$X_{i,l} = \frac{\epsilon_{i,l}}{1 + e^{-\alpha_{i,l}\{\sum_j (W_{i,j,l} X_{j,l-1}) + \theta_{i,l}\}}} - \frac{\epsilon_{i,l}}{2} \qquad (A.1)$$

where:

$\epsilon_{i,l}$ determine the saturation value of squashing output,

$\alpha_{i,l}$ determine the steepness of the sigmoid function,

$\theta_{i,l}$ is the threshold of the i^{th} node in the l^{th} layer,

$W_{i,j,l}$ is the weight between j^{th} node in $(l-1)^{th}$ layer to i^{th} node in l^{th} layer, and

$X_{j,l-1}$ is the output of the j^{th} node in $(l-1)^{th}$ layer.

$\epsilon_{i,l}$, $\alpha_{i,l}$, $\theta_{i,l}$, and $W_{i,j,l}$ are called learning parameters.

For a network with m outputs and after presentation of one pair of input and desired output pattern, the "computation" energy function E is defined by

$$E = \frac{1}{2} \sum_{j=1}^{m} (X_j^d - X_{j,n})^2 \qquad (A.2)$$

where

n is the total number of layers in the network,

m is the number of nodes in the ouput layer,

X_j^d is the desired output pattern, and

$X_{j,n}$ is the actual output pattern.

After presentation of p pairs of input and desired output patterns, E_{total} is defined by

$$E_{total} = \sum_p E \qquad (A.3)$$

The learning algorithm is the procedure that minimize E_{total} in Eq. (A.3) by means of adjusting the learning parameters, $\epsilon_{i,l}$, $\alpha_{i,l}$, $\theta_{i,l}$, and $W_{i,j,l}$. This procedure can be derived using delta rule or gradient descent method as proposed by Rummelhart [8]. The rule for changing the learning parameters following presentation of a pair of input and output pattern is given by

$$\left.\begin{array}{rcl} W_{i,j,l}(t+1) & = & W_{i,j,l}(t) + \Delta W_{i,j,l} \\ \theta_{i,l}(t+1) & = & \theta_{i,l}(t) + \Delta\theta_{i,l} \\ \alpha_{i,l}(t+1) & = & \alpha_{i,l}(t) + \Delta\alpha_{i,l} \\ \epsilon_{i,l}(t+1) & = & \epsilon_{i,l}(t) + \Delta\epsilon_{i,l} \end{array}\right\} \qquad (A.4)$$

where index t indicate iteration time, and

$$\left.\begin{aligned} \Delta W_{i,j,l} &= -\eta \frac{\partial E}{\partial W_{i,j,l}} \\ \Delta \theta_{i,l} &= -\eta \frac{\partial E}{\partial \theta_{i,l}} \\ \Delta \alpha_{i,l} &= -\eta \frac{\partial E}{\partial \alpha_{i,l}} \\ \Delta \epsilon_{i,l} &= -\eta \frac{\partial E}{\partial \epsilon_{i,l}} \end{aligned}\right\} \tag{A.5}$$

with

η is the learning rate or gain term ($0 < \eta < 1$), and

$\Delta W_{i,j,l}$ is the gradient or the change that must be made to the weight from j^{th} node in $(l-1)^{th}$ layer to i^{th} node in l^{th} layer.

To find $\Delta W_{i,j,l}$, $\Delta \theta_{i,l}$, $\Delta \alpha_{i,l}$, and $\Delta \epsilon_{i,l}$ in terms of network parameters, $X_{i,l}$, $W_{i,j,l}$, $\theta_{i,l}$, $\alpha_{i,l}$, and $\epsilon_{i,l}$, is our goal in the next development of the learning algorithm.

Let

$$Net_{i,l} = \sum_j (W_{i,j,l} X_{j,l-1}) + \theta_{i,l} \tag{A.6}$$

so that Eq. (A.1) becomes

$$X_{i,l} = \frac{\epsilon_{i,l}}{1 + e^{-\alpha_{i,l} Net_{i,l}}} - \frac{\epsilon_{i,l}}{2} \tag{A.7}$$

The following derivatives are computed from Eq. (A.6) and Eq. (A.7)

$$\left.\begin{aligned}
\frac{\partial Net_{i,l}}{\partial W_{i,j,l}} &= X_{j,l-1} \\
\frac{\partial Net_{i,l}}{\partial \theta_{i,l}} &= 1 \\
\frac{\partial X_{i,l}}{\partial \epsilon_{i,l}} &= \frac{X_{i,l}}{\epsilon_{i,l}} \\
\frac{\partial X_{i,l}}{\partial \alpha_{i,l}} &= (X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,Net_{i,l} \\
\frac{\partial X_{i,l}}{\partial Net_{i,j,l}} &= (X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,\alpha_{i,l}
\end{aligned}\right\} \qquad (A.8)$$

Now, the derivatives of E with respect to the learning parameters can be calculated by chain rule as

$$\left.\begin{aligned}
\frac{\partial E}{\partial W_{i,j,l}} &= \frac{\partial E}{\partial Net_{i,l}}\,\frac{\partial Net_{i,l}}{\partial W_{i,j,l}} \\
&= \frac{\partial E}{\partial Net_{i,l}}\,X_{j,l-1} \\
\frac{\partial E}{\partial \theta_{i,l}} &= \frac{\partial E}{\partial Net_{i,l}}\,\frac{\partial Net_{i,l}}{\partial \theta_{i,l}} \\
&= \frac{\partial E}{\partial Net_{i,l}} \\
\frac{\partial E}{\partial \epsilon_{i,l}} &= \frac{\partial E}{\partial X_{i,l}}\,\frac{\partial X_{i,l}}{\partial \epsilon_{i,l}} \\
&= \frac{\partial E}{\partial X_{i,l}}\,\frac{X_{i,l}}{\epsilon_{i,l}} \\
\frac{\partial E}{\partial \alpha_{i,l}} &= \frac{\partial E}{\partial X_{i,l}}\,\frac{\partial X_{i,l}}{\partial \alpha_{i,l}} \\
&= \frac{\partial E}{\partial X_{i,l}}\,(X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,Net_{i,l}
\end{aligned}\right\} \qquad (A.9)$$

All quantities in Eq. (A.9) are obtained from the forward pass calculation, except $\frac{\partial E}{\partial Net_{i,l}}$ and $\frac{\partial E}{\partial X_{i,l}}$ which can be calculated by propagating the error backward through the network. This can be achieved by the following derivation.

For the output layer where index $l = n$, the derivative $\frac{\partial E}{\partial Net_{i,l}}$ can be calculated by :

$$\frac{\partial E}{\partial Net_{i,n}} = \frac{\partial E}{\partial X_{i,n}}\,\frac{\partial X_{i,n}}{\partial Net_{i,n}} \qquad (A.10)$$

From Eq. (A.2), we have

$$\frac{\partial E}{\partial X_{i,n}} = -(X_i^d - X_{i,n}) \tag{A.11}$$

Substituting the last equation of Eq. (A.8) and Eq. (A.11) into Eq. (A.10) yields

$$\frac{\partial E}{Net_{i,n}} = -(X_i^d - X_{i,n})\,(X_{i,n} + \frac{\epsilon_{i,n}}{2})\,(\frac{1}{2} - \frac{X_{i,n}}{\epsilon_{i,n}})\,\alpha_{i,n} \tag{A.12}$$

Since we do not have the equation like Eq. (A.2) for the lower layers, we propagate the error backward by computing $\frac{\partial E}{\partial X_{i,l}}$ using the values that have already calculated in the upper layer as

$$\frac{\partial E}{\partial X_{i,l}} = \sum_j (\frac{\partial E}{\partial Net_{j,l+1}} \frac{\partial Net_{j,l+1}}{\partial X_{i,l}}) \tag{A.13}$$

where the summation extends over all nodes in the $(l-1)^{th}$ layer.

From Eq. (A.6) we have

$$\frac{\partial Net_{j,l+1}}{\partial X_{i,l}} = W_{j,i,l+1} \tag{A.14}$$

Hence Eq. (A.13) become

$$\frac{\partial E}{\partial X_{i,l}} = \sum_j (\frac{\partial E}{\partial Net_{j,l+1}} W_{j,i,l+1}) \tag{A.15}$$

and then $\frac{\partial E}{\partial Net_{i,l}}$ for the lower layers can be determined by

$$\frac{\partial E}{\partial Net_{i,l}} = \frac{\partial E}{\partial X_{i,l}} \frac{\partial X_{i,l}}{\partial Net_{i,l}} \tag{A.16}$$

Substituting the last equation of Eq. (A.8) and Eq. (A.15) into Eq. (A.16) yields

$$\frac{\partial E}{\partial Net_{i,l}} = \{\sum_j (\frac{\partial E}{\partial Net_{j,l+1}} W_{j,i,l+1})\}\,\{(X_{i,l} + \frac{\epsilon_{i,l}}{2})\,(\frac{1}{2} - \frac{X_{i,l}}{\epsilon_{i,l}})\,\alpha_{i,l}\} \tag{A.17}$$

Notice that quantities $\frac{\partial E}{\partial Net_{j,l+1}}$ and $W_{j,i,l+1}$ are already known from upper layer (previous computation).

Using Eq. (A.9), (A.11), (A.12), (A.15), and (A.17), we can write Eq. (A.5) as

For output layer:

$$\left.\begin{aligned}
\Delta W_{i,j,n} &= -\eta \frac{\partial E}{\partial W_{i,j,n}} \\
&= -\eta \{-(X_i^d - X_{i,n})\,(X_{i,n} + \tfrac{\epsilon_{i,n}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,n}}{\epsilon_{i,n}})\,\alpha_{i,n}\}\,X_{j,n-1} \\
\Delta \theta_{i,n} &= -\eta \frac{\partial E}{\partial \theta_{i,n}} \\
&= -\eta \{-(X_i^d - X_{i,n})\,(X_{i,n} + \tfrac{\epsilon_{i,n}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,n}}{\epsilon_{i,n}})\,\alpha_{i,n}\} \\
\Delta \alpha_{i,n} &= -\eta \frac{\partial E}{\partial \alpha_{i,n}} \\
&= -\eta \{-(X_i^d - X_{i,n})\,(X_{i,n} + \tfrac{\epsilon_{i,n}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,n}}{\epsilon_{i,n}})\,Net_{i,n}\} \\
\Delta \epsilon_{i,n} &= -\eta \frac{\partial E}{\partial \epsilon_{i,n}} \\
&= -\eta \{-(X_i^d - X_{i,n})\,\tfrac{X_{i,n}}{\epsilon_{i,n}}\}
\end{aligned}\right\} \qquad (A.18)$$

For hidden layers:

$$\left.\begin{aligned}
\Delta W_{i,j,l} &= -\eta \frac{\partial E}{\partial W_{i,j,l}} \\
&= -\eta \{\textstyle\sum_j (\frac{\partial E}{\partial Net_{j,l+1}}\,W_{i,j,l+1})\}\,(X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,\alpha_{i,l}\,X_{j,l-1} \\
\Delta \theta_{i,l} &= -\eta \frac{\partial E}{\partial \theta_{i,l}} \\
&= -\eta \{\textstyle\sum_j (\frac{\partial E}{\partial Ne_{j,l+1}}\,W_{i,j,l+1})\}\,(X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,\alpha_{i,l} \\
\Delta \alpha_{i,l} &= -\eta \frac{\partial E}{\partial \alpha_{i,l}} \\
&= -\eta \{\textstyle\sum_j (\frac{\partial E}{\partial Net_{j,l+1}}\,W_{i,j,l+1})\}\,(X_{i,l} + \tfrac{\epsilon_{i,l}}{2})\,(\tfrac{1}{2} - \tfrac{X_{i,l}}{\epsilon_{i,l}})\,Net_{i,l} \\
\Delta \epsilon_{i,l} &= -\eta \frac{\partial E}{\partial \epsilon_{i,l}} \\
&= -\eta \{\textstyle\sum_j (\frac{\partial E}{\partial Net_{j,l+1}}\,W_{i,j,l+1})\}\,\tfrac{X_{i,l}}{\epsilon_{i,l}}\}
\end{aligned}\right\} \qquad (A.19)$$

The above derivation is for one pair of input and desired output pattern. For multiple patterns, we presented them cyclically using the same iteration formula as in Eq. (A.4) until all learning parameters stabilize. Eq. (A.4) , Eq. (A.18) and Eq. (A.19) constitute the learning algorithm which is the well known Back Error Propagation (BEP) training algorithm.

For fast convergence and smooth changes in all the learning parameters, we can add a momentum term κ into Eq. (A.4) [23]. Therefore

$$\left.\begin{aligned} W_{i,j,l}(t+1) &= W_{i,j,l}(t) + \Delta W_{i,j,l} + \kappa\{W_{i,j,l}(t) - W_{i,j,l}(t-1)\} \\ \theta_{i,l}(t+1) &= \theta_{i,l}(t) + \Delta\theta_{i,l} + \kappa\{\theta_{i,l}(t) - \theta_{i,l}(t-1)\} \\ \alpha_{i,l}(t+1) &= \alpha_{i,l}(t) + \Delta\alpha_{i,l} + \kappa\{\alpha_{i,l}(t) - \alpha_{i,l}(t-1)\} \\ \epsilon_{i,l}(t+1) &= \epsilon_{i,l}(t) + \Delta\epsilon_{i,l} + \kappa\{\epsilon_{i,l}(t) - \epsilon_{i,l}(t-1)\} \end{aligned}\right\} \qquad (A.20)$$

where $\kappa =$ momentum term $(0 < \kappa < 1)$.

Appendix B

Table B.1: Learning Parameters Before Training (for simulation A)

$\theta_{i,l}$

i\l	1	2	3
1	0.353	0.261	0.222
2	0.886	0.498	0.393
3	0.159	0.865	0.220
4	0.059	0.960	0.193
5	0.159	0.229	0.054
6	0.583	0.670	0.444
7	0.293	0.824	-----
8	0.076	0.844	-----
9	0.942	0.425	-----
10	0.003	0.914	-----
11	0.788	0.399	-----
12	0.307	0.405	-----
13	0.212	0.222	-----
14	0.152	0.001	-----
15	0.644	0.815	-----
16	0.532	0.966	-----
17	0.327	0.492	-----
18	0.944	0.289	-----
19	0.273	0.174	-----
20	0.205	0.241	-----

$\epsilon_{i,l}$

i\l	1	2	3
1	0.840	0.059	0.702
2	0.319	0.891	0.067
3	0.584	0.288	0.218
4	0.691	0.458	0.350
5	0.164	0.377	0.634
6	0.604	0.300	0.031
7	0.390	0.566	-----
8	0.299	0.819	-----
9	0.857	0.943	-----
10	0.846	0.066	-----
11	0.533	0.761	-----
12	0.983	0.762	-----
13	0.609	0.485	-----
14	0.305	0.529	-----
15	0.387	0.019	-----
16	0.604	0.315	-----
17	0.652	0.809	-----
18	0.368	0.576	-----
19	0.517	0.261	-----
20	0.592	0.045	-----

$\alpha_{i,l}$

i\l	1	2	3
1	0.396	0.878	0.415
2	0.447	0.303	0.504
3	0.016	0.710	0.478
4	0.384	0.676	0.916
5	0.900	0.775	0.211
6	0.533	0.355	0.783
7	0.270	0.719	-----
8	0.742	0.391	-----
9	0.405	0.180	-----
10	0.663	0.521	-----
11	0.462	0.883	-----
12	0.266	0.688	-----
13	0.601	0.125	-----
14	0.886	0.873	-----
15	0.338	0.861	-----
16	0.754	0.243	-----
17	0.459	0.936	-----
18	0.946	0.220	-----
19	0.007	0.321	-----
20	0.024	0.002	-----

$W_{i,j,l}$, $l = 1$

i\j	1	2
1	0.176	0.932
2	0.910	0.473
3	0.872	0.696
4	0.930	0.455
5	0.399	0.893
6	0.694	0.839
7	0.740	0.651
8	0.678	0.577
9	0.273	0.935
10	0.662	0.047
11	0.373	0.618
12	0.149	0.377
13	0.645	0.026
14	0.841	0.077
15	0.743	0.256
16	0.902	0.378
17	0.320	0.211
18	0.649	0.251
19	0.229	0.251
20	0.943	0.137

$W_{i,j,l}$, $l = 2$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.270	0.549	0.324	0.865	0.297	0.680	0.833	0.876	0.650	0.073
2	0.858	0.343	0.692	0.345	0.894	0.959	0.122	0.982	0.055	0.615
3	0.265	0.293	0.196	0.018	0.830	0.573	0.105	0.733	0.119	0.224
4	0.401	0.871	0.046	0.295	0.394	0.560	0.311	0.823	0.475	0.091
5	0.423	0.566	0.970	0.354	0.432	0.179	0.215	0.337	0.454	0.045
6	0.152	0.705	0.730	0.843	0.618	0.955	0.479	0.527	0.029	0.608
7	0.402	0.985	0.630	0.832	0.437	0.791	0.059	0.135	0.544	0.930
8	0.748	0.410	0.476	0.344	0.417	0.024	0.526	0.718	0.236	0.695
9	0.598	0.016	0.757	0.131	0.375	0.217	0.578	0.206	0.787	0.748
10	0.418	0.497	0.885	0.563	0.686	0.968	0.182	0.568	0.030	0.204
11	0.990	0.797	0.387	0.182	0.225	0.737	0.534	0.665	0.202	0.469
12	0.897	0.134	0.948	0.429	0.078	0.784	0.444	0.421	0.702	0.190
13	0.261	0.414	0.688	0.754	0.171	0.129	0.159	0.293	0.858	0.279
14	0.761	0.642	0.735	0.547	0.100	0.732	0.529	0.519	0.287	0.947
15	0.361	0.817	0.855	0.289	0.999	0.009	0.064	0.968	0.683	0.302
16	0.200	0.901	0.129	0.929	0.765	0.480	0.556	0.012	0.384	0.378
17	0.951	0.232	0.520	0.617	0.188	0.979	0.503	0.160	0.540	0.193
18	0.441	0.119	0.920	0.674	0.165	0.780	0.710	0.185	0.777	0.215
19	0.690	0.155	0.328	0.451	0.412	0.507	0.846	0.825	0.569	0.692
20	0.033	0.879	0.181	0.965	0.738	0.445	0.799	0.495	0.331	0.164

$W_{i,j,l}$, $l = 2$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.899	0.254	0.611	0.842	0.832	0.373	0.757	0.109	0.861	0.559
2	0.038	0.376	0.526	0.382	0.661	0.607	0.816	0.447	0.027	0.472
3	0.947	0.739	0.821	0.826	0.251	0.256	0.338	0.388	0.527	0.266
4	0.262	0.917	0.978	0.332	0.902	0.241	0.373	0.752	0.458	0.901
5	0.684	0.062	0.561	0.316	0.268	0.501	0.063	0.965	0.730	0.806
6	0.048	0.593	0.403	0.904	0.330	0.311	0.855	0.734	0.099	0.340
7	0.885	0.390	0.415	0.857	0.287	0.108	0.577	0.778	0.142	0.809
8	0.780	0.763	0.122	0.371	0.928	0.766	0.290	0.984	0.428	0.038
9	0.455	0.285	0.850	0.450	0.835	0.430	0.331	0.951	0.249	0.357
10	0.158	0.548	0.204	0.211	0.067	0.193	0.902	0.782	0.303	0.422
11	0.566	0.097	0.277	0.017	0.564	0.887	0.070	0.062	0.253	0.307
12	0.126	0.248	0.658	0.138	0.323	0.823	0.014	0.924	0.376	0.582
13	0.688	0.839	0.815	0.049	0.073	0.431	0.750	0.152	0.412	0.103
14	0.426	0.494	0.702	0.258	0.811	0.880	0.001	0.107	0.273	0.984
15	0.725	0.924	0.649	0.916	0.536	0.977	0.233	0.419	0.582	0.780
16	0.729	0.544	0.678	0.590	0.283	0.108	0.299	0.529	0.905	0.437
17	0.249	0.330	0.151	0.623	0.212	0.944	0.622	0.214	0.474	0.843
18	0.255	0.663	0.961	0.942	0.808	0.797	0.986	0.700	0.665	0.118
19	0.140	0.847	0.701	0.208	0.067	0.439	0.845	0.301	0.339	0.537
20	0.737	0.449	0.350	0.618	0.942	0.138	0.759	0.045	0.587	0.658

W_{ijl}, $l = 3$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.010	0.967	0.336	0.845	0.612	0.012	0.852	0.511	0.234	0.670
2	0.948	0.030	0.405	0.767	0.662	0.869	0.794	0.569	0.587	0.576
3	0.098	0.794	0.326	0.843	0.043	0.380	0.167	0.401	0.153	0.327
4	0.013	0.774	0.507	0.119	0.079	0.554	0.077	0.777	0.155	0.411
5	0.081	0.477	0.131	0.668	0.689	0.002	0.280	0.287	0.781	0.974
6	0.498	0.224	0.433	0.718	0.563	0.930	0.914	0.396	0.503	0.628

W_{ijl}, $l = 3$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.802	0.060	0.760	0.619	0.924	0.362	0.204	0.326	0.556	0.874
2	0.573	0.281	0.749	0.419	0.801	0.001	0.168	0.537	0.065	0.922
3	0.480	0.645	0.124	0.638	0.743	0.890	0.895	0.095	0.870	0.396
4	0.029	0.818	0.631	0.156	0.611	0.556	0.888	0.483	0.337	0.116
5	0.148	0.349	0.467	0.349	0.895	0.392	0.173	0.576	0.927	0.313
6	0.422	0.616	0.711	0.244	0.877	0.260	0.370	0.566	0.218	0.272

Table B.2: Learning parameters for simulation A (after training)

$\theta_{i,l}$

i\l	1	2	3
1	0.351	0.255	0.205
2	0.886	0.485	0.430
3	0.159	0.863	-1.035
4	0.058	0.973	0.962
5	0.157	0.249	-0.187
6	0.579	0.647	-2.353
7	0.293	0.837	-----
8	0.073	0.823	-----
9	0.941	-5.886	-----
10	0.002	0.881	-----
11	0.787	0.418	-----
12	0.306	0.378	-----
13	0.211	-7.051	-----
14	0.151	0.003	-----
15	0.643	0.784	-----
16	0.529	0.958	-----
17	0.326	0.468	-----
18	0.943	-0.404	-----
19	0.273	0.172	-----
20	0.205	0.241	-----

$\epsilon_{i,l}$

i\l	1	2	3
1	0.848	0.247	0.471
2	0.317	0.139	0.471
3	0.579	0.742	0.563
4	0.689	-1.277	1.773
5	0.185	-0.765	1.345
6	0.612	0.105	0.808
7	0.388	-0.287	-----
8	0.308	0.171	-----
9	0.856	4.475	-----
10	0.843	-0.165	-----
11	0.530	-1.029	-----
12	0.981	0.427	-----
13	0.603	3.150	-----
14	0.301	-0.973	-----
15	0.383	0.353	-----
16	0.604	-0.229	-----
17	0.648	0.611	-----
18	0.368	1.363	-----
19	0.514	-0.760	-----
20	0.587	-0.227	-----

$\alpha_{i,l}$

i\l	1	2	3
1	0.423	0.855	0.000
2	0.448	-0.436	0.000
3	0.028	0.712	-1.558
4	0.388	0.751	-0.630
5	0.903	0.822	0.069
6	0.545	-0.011	-1.678
7	0.371	0.748	-----
8	0.748	-0.048	-----
9	0.406	-3.045	-----
10	0.664	0.331	-----
11	0.463	0.927	-----
12	0.271	0.526	-----
13	0.601	-7.466	-----
14	0.886	0.884	-----
15	0.339	0.705	-----
16	0.765	-0.083	-----
17	0.463	0.844	-----
18	0.946	-1.105	-----
19	0.008	0.378	-----
20	0.026	-0.031	-----

W_{ijl}, $l = 1$

i\j	1	2
1	0.176	0.933
2	0.910	0.473
3	0.872	0.696
4	0.930	0.455
5	0.399	0.893
6	0.694	0.838
7	0.740	0.651
8	0.678	0.577
9	0.273	0.935
10	0.662	0.047
11	0.373	0.518
12	0.149	0.377
13	0.645	0.025
14	0.841	0.077
15	0.743	0.256
16	0.902	0.377
17	0.320	0.211
18	0.649	0.251
19	0.229	0.251
20	0.943	0.137

$H_{i,j,l}$, $l = 2$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.268	0.548	0.322	0.863	0.296	0.678	0.832	0.876	0.647	0.070
2	0.866	0.343	0.688	0.346	0.898	0.965	0.122	0.985	0.062	0.610
3	0.281	0.293	0.195	0.019	0.829	0.573	0.105	0.733	0.116	0.228
4	0.398	0.874	0.050	0.300	0.392	0.560	0.313	0.823	0.474	0.103
5	0.419	0.569	0.976	0.359	0.428	0.178	0.217	0.337	0.452	0.057
6	0.149	0.702	0.723	0.837	0.619	0.953	0.476	0.526	0.025	0.597
7	0.397	0.986	0.633	0.833	0.434	0.788	0.060	0.133	0.540	0.936
8	0.738	0.405	0.469	0.336	0.415	0.015	0.521	0.714	0.225	0.684
9	6.571	0.020	-0.861	0.281	3.163	4.110	0.907	2.061	5.826	-2.721
10	0.417	0.493	0.876	0.555	0.687	0.965	0.178	0.567	0.026	0.188
11	0.985	0.801	0.393	0.189	0.222	0.738	0.536	0.666	0.200	0.485
12	0.880	0.129	0.940	0.419	0.074	0.772	0.438	0.416	0.684	0.180
13	5.215	0.569	-1.157	1.363	2.661	3.773	0.463	2.274	4.857	-2.023
14	0.758	0.643	0.735	0.549	0.099	0.732	0.529	0.520	0.284	0.952
15	0.351	0.811	0.846	0.280	0.998	0.000	0.058	0.964	0.671	0.288
16	0.197	0.899	0.127	0.926	0.764	0.477	0.555	0.011	0.380	0.374
17	0.941	0.228	0.513	0.609	0.186	0.972	0.499	0.157	0.528	0.183
18	0.727	0.083	0.734	0.623	0.325	0.968	0.688	0.284	0.991	-0.087
19	0.684	0.155	0.327	0.451	0.410	0.505	0.846	0.824	0.564	0.695
20	0.033	0.879	0.181	0.965	0.728	0.445	0.799	0.495	0.331	0.153

$H_{i,j,l}$, $l = 2$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.807	0.251	0.609	0.841	0.831	0.371	0.755	0.108	0.849	0.557
2	0.040	0.374	0.522	0.281	0.559	0.609	0.815	0.446	0.034	0.468
3	0.946	0.737	0.824	0.829	0.253	0.259	0.338	0.389	0.537	0.260
4	0.263	0.921	0.987	0.338	0.906	0.247	0.378	0.757	0.461	0.905
5	0.685	0.069	0.500	0.321	0.272	0.506	0.068	0.970	0.735	0.812
6	0.044	0.584	0.395	0.900	0.338	0.305	0.848	0.729	0.093	0.334
7	0.884	0.393	0.419	0.859	0.288	0.108	0.579	0.779	0.145	0.812
8	0.773	0.752	0.114	0.366	0.923	0.757	0.282	0.977	0.423	0.032
9	2.145	-0.260	-1.877	-0.901	0.017	0.692	-0.488	0.296	-1.350	-1.499
10	0.153	0.535	0.192	0.205	0.061	0.184	0.892	0.775	0.294	0.413
11	0.568	0.104	0.288	0.024	0.569	0.896	0.076	0.068	0.258	0.313
12	0.116	0.233	0.650	0.134	0.317	0.812	0.004	0.917	0.369	0.574
13	2.039	-0.268	-1.227	-0.597	-0.617	1.369	-0.140	-0.302	-1.432	-1.980
14	0.426	0.494	0.705	0.260	0.812	0.883	0.002	0.108	0.274	0.985
15	0.717	0.909	0.639	0.911	0.530	0.966	0.223	0.411	0.574	0.771
16	0.728	0.540	0.675	0.588	0.281	0.104	0.298	0.526	0.903	0.435
17	0.242	0.318	0.144	0.619	0.207	0.936	0.614	0.208	0.467	0.836
18	0.296	0.496	0.719	0.833	0.708	0.757	0.851	0.603	0.488	-0.084
19	0.139	0.845	0.703	0.209	0.067	0.440	0.845	0.301	0.339	0.537
20	0.737	0.449	0.350	0.618	0.941	0.138	0.759	0.045	0.587	0.657

$H_{i,j,l}$, $l = 3$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.098	0.953	0.330	0.835	0.604	0.007	0.840	0.497	0.182	0.668
2	0.947	0.033	0.404	0.766	0.662	0.866	0.794	0.572	0.436	0.572
3	0.047	0.587	0.035	0.701	0.318	0.420	0.234	0.263	-3.218	0.404
4	0.223	0.493	0.571	0.004	-0.135	0.685	0.098	0.553	1.521	0.400
5	0.046	0.479	0.111	0.622	0.665	-0.026	0.231	0.282	0.567	0.944
6	-0.135	0.208	-0.420	0.843	0.503	0.022	0.203	-0.033	2.472	0.172

$H_{i,j,l}$, $l = 3$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.787	0.045	0.732	0.608	0.922	0.357	0.156	0.316	0.552	0.874
2	0.576	0.281	0.667	0.421	0.795	0.001	0.169	0.537	0.965	0.922
3	0.379	0.260	1.082	0.448	0.421	0.727	0.208	0.008	0.703	0.406
4	-0.224	0.640	-0.512	-0.101	0.775	0.452	0.670	0.292	0.149	0.130
5	0.134	0.326	0.513	0.336	0.831	0.392	0.166	0.563	0.920	0.305
6	0.649	-0.307	1.790	0.628	-0.126	0.195	-0.325	1.366	0.543	0.152

Table B.3: Learning parameters for simulation B and C (before training)

β_{il}

i\l	1	2	3
1	0.353	0.261	0.222
2	0.886	0.498	0.393
3	0.159	0.865	0.220
4	0.059	0.960	0.193
5	0.159	0.229	0.054
6	0.583	0.670	0.444
7	0.293	0.824	0.910
8	0.076	0.844	0.696
9	0.942	0.425	0.399
10	0.003	0.914	-----
11	0.788	0.399	-----
12	0.307	0.405	-----
13	0.212	0.222	-----
14	0.152	0.001	-----
15	0.644	0.815	-----
16	0.532	0.966	-----
17	0.327	0.492	-----
18	0.944	0.289	-----
19	0.273	0.174	-----
20	0.205	0.241	-----

ϵ_{il}

i\l	1	2	3
1	0.840	0.059	0.702
2	0.319	0.891	0.067
3	0.584	0.286	0.218
4	0.691	0.458	0.350
5	0.164	0.377	0.634
6	0.604	0.300	0.031
7	0.390	0.566	0.932
8	0.299	0.819	0.872
9	0.857	0.943	0.455
10	0.846	0.066	-----
11	0.533	0.761	-----
12	0.983	0.762	-----
13	0.609	0.485	-----
14	0.305	0.529	-----
15	0.387	0.019	-----
16	0.604	0.315	-----
17	0.652	0.809	-----
18	0.368	0.576	-----
19	0.517	0.261	-----
20	0.592	0.045	-----

α_{il}

i\l	1	2	3
1	0.396	0.878	0.415
2	0.447	0.303	0.504
3	0.016	0.710	0.479
4	0.384	0.676	0.916
5	0.900	0.776	0.211
6	0.533	0.356	0.783
7	0.270	0.719	0.176
8	0.742	0.391	0.473
9	0.405	0.190	0.930
10	0.663	0.521	-----
11	0.462	0.883	-----
12	0.266	0.699	-----
13	0.601	0.125	-----
14	0.886	0.873	-----
15	0.338	0.861	-----
16	0.754	0.243	-----
17	0.459	0.936	-----
18	0.946	0.220	-----
19	0.007	0.321	-----
20	0.024	0.002	-----

W_{ijl}, $l = 1$

i\j	1	2	3
1	0.893	0.694	0.839
2	0.740	0.651	0.678
3	0.577	0.273	0.935
4	0.662	0.047	0.373
5	0.618	0.149	0.377
6	0.645	0.026	0.841
7	0.077	0.743	0.256
8	0.902	0.378	0.320
9	0.211	0.649	0.251
10	0.229	0.251	0.943
11	0.137	0.270	0.549
12	0.324	0.865	0.297
13	0.680	0.833	0.876
14	0.650	0.073	0.899
15	0.264	0.611	0.842
16	0.832	0.373	0.757
17	0.109	0.851	0.559
18	0.858	0.343	0.692
19	0.345	0.894	0.959
20	0.122	0.982	0.055

W_{ijl}, $l = 2$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.615	0.038	0.376	0.526	0.282	0.561	0.607	0.816	0.447	0.027
2	0.224	0.947	0.739	0.821	0.626	0.251	0.256	0.338	0.388	0.527
3	0.091	0.262	0.917	0.978	0.332	0.902	0.241	0.373	0.752	0.458
4	0.045	0.684	0.062	0.551	0.316	0.268	0.501	0.063	0.965	0.730
5	0.608	0.048	0.593	0.403	0.904	0.330	0.311	0.855	0.734	0.099
6	0.930	0.885	0.390	0.415	0.857	0.287	0.103	0.577	0.778	0.142
7	0.695	0.780	0.763	0.122	0.371	0.928	0.765	0.290	0.984	0.428
8	0.748	0.455	0.285	0.850	0.450	0.835	0.430	0.331	0.951	0.249
9	0.204	0.158	0.548	0.204	0.211	0.067	0.193	0.902	0.782	0.303
10	0.469	0.566	0.097	0.277	0.017	0.564	0.857	0.070	0.062	0.253
11	0.190	0.126	0.248	0.658	0.138	0.323	0.823	0.014	0.924	0.376
12	0.279	0.688	0.839	0.815	0.049	0.073	0.431	0.750	0.152	0.412
13	0.947	0.426	0.494	0.702	0.258	0.811	0.880	0.001	0.107	0.273
14	0.302	0.725	0.924	0.649	0.916	0.536	0.977	0.233	0.419	0.582
15	0.378	0.729	0.544	0.678	0.599	0.283	0.109	0.289	0.529	0.905
16	0.193	0.249	0.330	0.151	0.623	0.212	0.944	0.622	0.214	0.474
17	0.215	0.255	0.663	0.961	0.942	0.808	0.797	0.986	0.700	0.655
18	0.692	0.140	0.847	0.701	0.208	0.067	0.439	0.845	0.301	0.339
19	0.154	0.737	0.449	0.350	0.618	0.942	0.138	0.759	0.045	0.587
20	0.670	0.802	0.060	0.780	0.619	0.924	0.362	0.204	0.326	0.556

W_{ijl}, $l = 2$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.473	0.285	0.293	0.196	0.018	0.830	0.573	0.106	0.733	0.119
2	0.266	0.401	0.871	0.046	0.295	0.394	0.560	0.311	0.833	0.475
3	0.901	0.423	0.566	0.970	0.354	0.432	0.179	0.215	0.337	0.454
4	0.806	0.152	0.705	0.730	0.843	0.618	0.965	0.479	0.537	0.029
5	0.340	0.402	0.985	0.630	0.832	0.437	0.791	0.059	0.135	0.544
6	0.809	0.748	0.410	0.475	0.344	0.417	0.024	0.526	0.718	0.236
7	0.038	0.598	0.016	0.757	0.131	0.375	0.217	0.578	0.206	0.787
8	0.357	0.418	0.497	0.885	0.563	0.686	0.959	0.182	0.568	0.030
9	0.422	0.990	0.797	0.387	0.182	0.225	0.737	0.534	0.665	0.202
10	0.307	0.897	0.134	0.949	0.429	0.078	0.784	0.444	0.421	0.702
11	0.682	0.261	0.414	0.688	0.754	0.171	0.129	0.159	0.293	0.858
12	0.103	0.761	0.642	0.735	0.547	0.100	0.732	0.529	0.519	0.287
13	0.984	0.361	0.817	0.855	0.289	0.999	0.069	0.064	0.998	0.683
14	0.780	0.200	0.901	0.129	0.929	0.765	0.490	0.556	0.012	0.384
15	0.437	0.951	0.232	0.530	0.617	0.188	0.979	0.503	0.160	0.540
16	0.843	0.441	0.119	0.930	0.674	0.165	0.790	0.710	0.185	0.777
17	0.118	0.690	0.155	0.328	0.451	0.412	0.507	0.846	0.825	0.569
18	0.537	0.033	0.879	0.181	0.965	0.728	0.445	0.799	0.495	0.331
19	0.658	0.010	0.967	0.336	0.845	0.612	0.012	0.852	0.511	0.234
20	0.874	0.948	0.030	0.405	0.767	0.662	0.969	0.794	0.569	0.587

W_{ijl}, $l = 3$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.576	0.573	0.281	0.749	0.419	0.801	0.001	0.168	0.537	0.965
2	0.327	0.480	0.645	0.124	0.638	0.743	0.890	0.895	0.095	0.879
3	0.411	0.029	0.818	0.631	0.156	0.611	0.566	0.888	0.483	0.337
4	0.974	0.148	0.349	0.467	0.349	0.895	0.392	0.173	0.576	0.927
5	0.628	0.422	0.616	0.711	0.244	0.877	0.260	0.370	0.566	0.218
6	0.260	0.843	0.758	0.235	0.715	0.551	0.775	0.694	0.482	0.838
7	0.988	0.280	0.459	0.460	0.970	0.532	0.777	0.706	0.438	0.432
8	0.803	0.332	0.575	0.665	0.048	0.614	0.705	0.799	0.919	0.900
9	0.004	0.589	0.997	0.586	0.715	0.077	0.578	0.684	0.458	0.438

W_{ijl}, $l = 3$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.922	0.098	0.794	0.326	0.843	0.043	0.380	0.167	0.401	0.153
2	0.396	0.013	0.774	0.507	0.119	0.079	0.554	0.077	0.777	0.155
3	0.116	0.081	0.477	0.131	0.608	0.689	0.002	0.280	0.287	0.781
4	0.313	0.498	0.224	0.433	0.718	0.563	0.930	0.914	0.398	0.503
5	0.272	0.019	0.415	0.633	0.654	0.270	0.476	0.061	0.306	0.274
6	0.111	0.803	0.865	0.803	0.800	0.518	0.334	0.878	0.631	0.995
7	0.020	0.190	0.593	0.131	0.040	0.593	0.300	0.857	0.548	0.350
8	0.160	0.075	0.533	0.718	0.968	0.461	0.394	0.742	0.970	0.075
9	0.757	0.268	0.342	0.391	0.443	0.409	0.252	0.113	0.978	0.820

Table B.4: Learning parameters for simulation B (after training)

θ_{il}

i\l	1	2	3
1	0.313	0.115	0.789
2	0.843	-.134	1.620
3	0.121	0.268	0.851
4	-.037	0.634	1.077
5	0.110	-.190	0.461
6	0.485	0.590	-1.17
7	0.281	0.439	0.953
8	0.037	0.215	0.054
9	0.871	-.275	-.138
10	-.363	0.800	-----
11	0.725	-1.66	-----
12	0.255	-.043	-----
13	0.031	0.060	-----
14	0.035	-.190	-----
15	0.621	0.437	-----
16	0.420	0.899	-----
17	0.252	0.349	-----
18	0.909	0.170	-----
19	0.247	-.328	-----
20	0.199	0.240	-----

ϵ_{il}

i\l	1	2	3
1	0.878	0.477	0.554
2	0.343	1.375	0.600
3	0.651	0.644	0.562
4	0.736	0.696	1.514
5	0.239	0.782	1.331
6	0.630	0.410	1.037
7	0.414	0.574	1.325
8	0.358	0.927	2.680
9	0.817	2.161	3.313
10	1.186	0.181	-----
11	0.528	2.413	-----
12	1.004	1.020	-----
13	0.767	0.655	-----
14	0.404	2.349	-----
15	0.428	0.418	-----
16	0.619	0.260	-----
17	0.732	1.174	-----
18	0.318	0.475	-----
19	0.575	1.226	-----
20	0.589	0.042	-----

α_{il}

i\l	1	2	3
1	0.431	0.851	0.107
2	0.398	0.973	1.838
3	0.292	0.515	0.912
4	0.459	0.618	0.722
5	0.908	0.750	2.893
6	0.497	0.397	1.763
7	0.299	0.407	0.339
8	0.762	0.595	1.055
9	0.323	1.726	0.378
10	0.863	0.283	-----
11	0.413	4.394	-----
12	0.341	0.686	-----
13	0.559	0.441	-----
14	0.873	3.024	-----
15	0.347	0.425	-----
16	0.666	0.211	-----
17	0.493	0.971	-----
18	0.889	0.100	-----
19	0.269	1.070	-----
20	0.064	0.025	-----

W_{ijl}, $l = 1$

i\j	1	2	3
1	0.893	0.692	0.827
2	0.740	0.650	0.666
3	0.577	0.272	0.924
4	0.662	0.045	0.348
5	0.618	0.147	0.364
6	0.645	0.023	0.813
7	0.077	0.742	0.252
8	0.902	0.377	0.308
9	0.211	0.647	0.231
10	0.229	0.240	0.838
11	0.137	0.268	0.531
12	0.324	0.864	0.282
13	0.680	0.828	0.825
14	0.650	0.070	0.864
15	0.254	0.611	0.836
16	0.832	0.369	0.724
17	0.109	0.849	0.538
18	0.858	0.342	0.682
19	0.345	0.893	0.951
20	0.122	0.991	0.053

$W_{i,j,l}$, $l = 2$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.624	0.038	0.366	0.530	0.285	0.533	0.614	0.836	0.451	-.050
2	0.319	0.951	0.786	0.849	0.840	0.324	0.276	0.350	0.372	0.909
3	0.101	0.245	0.927	0.986	0.332	0.911	0.242	0.369	0.710	0.590
4	0.066	0.680	0.064	0.559	0.320	0.255	0.509	0.078	0.958	0.739
5	0.630	0.035	0.614	0.412	0.906	0.370	0.313	0.845	0.697	0.314
6	0.935	0.884	0.384	0.417	0.868	0.267	0.112	0.589	0.782	0.088
7	0.701	0.768	0.758	0.127	0.371	0.915	0.769	0.296	0.962	0.436
8	0.782	0.446	0.288	0.863	0.455	0.828	0.441	0.347	0.928	0.319
9	0.615	0.272	0.562	0.311	0.296	0.007	0.329	1.168	1.008	0.179
10	0.468	0.563	0.094	0.278	0.017	0.555	0.888	0.073	0.056	0.238
11	1.011	0.313	0.699	0.867	0.283	0.893	0.998	0.176	1.103	3.321
12	0.292	0.673	0.858	0.823	0.049	0.112	0.431	0.737	0.110	0.631
13	0.971	0.428	0.494	0.710	0.262	0.809	0.888	0.015	0.111	0.286
14	-.318	0.456	1.160	0.490	0.738	1.783	0.629	-.706	-.407	4.216
15	0.375	0.716	0.537	0.681	0.590	0.252	0.111	0.309	0.509	0.864
16	0.197	0.248	0.330	0.153	0.623	0.211	0.945	0.624	0.312	0.481
17	0.250	0.255	0.666	0.973	0.950	0.784	0.813	1.017	0.711	0.622
18	0.693	0.137	0.842	0.703	0.208	0.057	0.441	0.849	0.296	0.323
19	0.357	0.779	0.508	0.403	0.664	1.011	0.192	0.835	0.104	0.948
20	0.670	0.802	0.060	0.760	0.619	0.924	0.362	0.203	0.325	0.557

$W_{i,j,l}$, $l = 2$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.457	0.312	0.305	0.171	0.010	0.824	0.580	0.095	0.733	0.123
2	0.298	0.440	0.976	0.149	0.339	0.441	0.641	0.281	0.859	0.474
3	0.893	0.415	0.509	1.000	0.368	0.411	0.184	0.161	0.339	0.451
4	0.796	0.177	0.718	0.726	0.843	0.609	0.971	0.451	0.533	0.031
5	0.352	0.390	1.000	0.686	0.849	0.436	0.808	0.019	0.143	0.539
6	0.798	0.764	0.405	0.455	0.339	0.407	0.025	0.513	0.716	0.237
7	0.024	0.605	0.009	0.752	0.127	0.352	0.218	0.535	0.200	0.786
8	0.344	0.444	0.523	0.897	0.567	0.674	0.992	0.131	0.572	0.030
9	0.428	1.460	1.243	0.351	0.254	0.430	1.084	0.585	0.776	0.252
10	0.300	0.899	0.129	0.941	0.426	0.069	0.781	0.431	0.418	0.702
11	0.901	0.728	1.404	1.437	1.135	0.752	0.869	0.279	0.721	0.891
12	0.112	0.743	0.654	0.791	0.562	0.093	0.742	0.485	0.525	0.282
13	0.982	0.384	0.837	0.857	0.294	1.004	0.027	0.051	0.969	0.684
14	1.298	-1.23	0.336	1.391	1.179	0.737	-.003	0.429	-.201	0.208
15	0.414	0.962	0.220	0.499	0.605	0.159	0.973	0.463	0.157	0.540
16	0.842	0.444	0.121	0.922	0.676	0.164	0.783	0.703	0.184	0.777
17	0.106	0.740	0.163	0.305	0.451	0.399	0.529	0.814	0.836	0.573
18	0.531	0.037	0.875	0.174	0.961	0.719	0.444	0.785	0.491	0.331
19	0.702	0.165	1.179	0.435	0.914	0.726	0.184	0.858	0.579	0.343
20	0.874	0.948	0.030	0.406	0.767	0.662	0.868	0.794	0.569	0.587

$W_{i,j,l}$, $l = 3$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.582	0.575	0.301	0.776	0.496	0.817	0.026	0.196	0.501	0.978
2	0.340	0.524	0.724	0.245	0.639	0.769	1.003	1.017	0.102	0.906
3	0.406	-.035	0.823	0.660	0.103	0.622	0.595	0.902	0.357	0.343
4	0.958	-.392	0.106	0.202	0.144	0.818	0.194	-.196	-.020	0.886
5	0.725	1.011	0.748	0.841	0.779	0.927	0.415	0.638	1.651	0.230
6	0.097	-.020	0.453	-.049	0.284	0.423	0.518	0.190	-.976	0.757
7	0.924	-.138	0.214	0.208	0.621	0.438	0.534	0.363	0.086	0.381
8	0.816	0.545	0.469	0.506	0.452	0.617	0.670	0.787	1.403	0.860
9	0.036	0.952	1.098	0.634	1.096	0.141	0.699	0.814	0.463	0.438

$W_{i,j,l}$, $l = 3$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.057	0.095	0.802	-.319	0.864	0.053	0.417	0.170	0.329	0.153
2	-.099	0.048	0.784	-.264	0.178	0.108	0.704	0.091	0.717	0.155
3	-.040	0.032	0.452	-.174	0.667	0.697	0.007	0.272	0.156	0.781
4	1.083	0.133	0.134	-.684	0.505	0.538	0.446	0.882	-.063	0.502
5	0.976	0.634	0.699	-1.37	0.749	0.303	0.879	0.210	0.975	0.277
6	-2.12	0.248	0.557	1.462	0.592	0.445	-.338	0.798	-.226	0.993
7	-.173	-.252	0.413	-1.23	-.154	0.538	-.192	0.732	0.088	0.348
8	0.133	0.472	0.674	-.321	0.807	0.450	0.470	0.835	1.063	0.077
9	0.422	0.824	0.037	0.886	0.522	0.447	0.603	0.313	1.460	0.833

Table B.5: Learning parameters for simulation C (after training)

θ_{il}

i\l	1	2	3
1	0.170	-.549	0.959
2	0.855	0.004	1.379
3	0.139	0.585	0.627
4	-.025	0.518	2.196
5	0.115	-.006	2.158
6	0.490	0.225	1.449
7	0.245	0.251	0.457
8	-.083	0.074	0.416
9	0.835	-2.09	0.066
10	-.235	0.689	-----
11	0.718	0.053	-----
12	0.199	-.241	-----
13	0.012	-.324	-----
14	0.094	0.155	-----
15	0.828	-.215	-----
16	0.288	0.839	-----
17	0.130	-.335	-----
18	0.916	-.651	-----
19	0.182	-.405	-----
20	0.196	0.240	-----

$c_{i,l}$

i\l	1	2	3
1	1.059	1.005	0.681
2	0.291	0.820	0.729
3	0.612	0.268	0.387
4	0.747	0.443	2.671
5	0.240	0.453	1.247
6	0.631	0.336	0.328
7	0.518	0.447	2.281
8	0.770	0.648	3.961
9	0.852	2.037	-1.96
10	0.968	0.004	-----
11	0.525	0.703	-----
12	1.079	0.846	-----
13	0.741	0.734	-----
14	0.364	3.434	-----
15	0.397	0.597	-----
16	0.798	0.079	-----
17	0.813	1.094	-----
18	0.304	0.860	-----
19	0.656	0.629	-----
20	0.608	0.028	-----

a_{il}

i\l	1	2	3
1	0.590	1.116	0.595
2	0.356	0.296	1.499
3	0.199	0.310	0.598
4	0.481	0.333	0.691
5	0.907	0.551	3.126
6	0.539	0.305	1.953
7	0.382	0.352	1.226
8	0.972	0.330	1.061
9	0.348	4.554	0.384
10	0.782	0.004	-----
11	0.398	0.648	-----
12	0.430	0.538	-----
13	0.434	0.487	-----
14	0.889	3.156	-----
15	0.313	0.429	-----
16	0.736	0.078	-----
17	0.390	0.833	-----
18	0.897	0.776	-----
19	0.280	0.570	-----
20	0.153	0.008	-----

W_{ijl}, $l = 1$

i\j	1	2	3
1	0.893	0.073	0.783
2	0.740	0.646	0.067
3	0.577	0.270	0.929
4	0.662	0.035	0.344
5	0.618	0.142	0.363
6	0.645	0.014	0.811
7	0.077	0.737	0.241
8	0.903	0.360	0.271
9	0.211	0.634	0.216
10	0.229	0.220	0.867
11	0.137	0.261	0.525
12	0.324	0.862	0.203
13	0.680	0.807	0.811
14	0.660	0.065	0.879
15	0.254	0.609	0.836
16	0.832	0.344	0.681
17	0.109	0.827	0.498
18	0.868	0.339	0.082
19	0.345	0.883	0.931
20	0.122	0.980	0.052

W_{ijl}, $l = 2$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.890	0.080	0.416	0.617	0.339	0.607	0.689	1.078	0.491	0.282
2	0.206	0.934	0.742	0.808	0.816	0.236	0.258	0.311	0.364	0.643
3	0.072	0.260	0.915	0.977	0.330	0.889	0.237	0.369	0.726	0.464
4	0.031	0.668	0.062	0.546	0.312	0.251	0.502	0.056	0.933	0.767
5	0.594	0.038	0.590	0.425	0.913	0.321	0.298	0.897	0.699	0.039
6	0.922	0.872	0.390	0.414	0.855	0.272	0.110	0.577	0.751	0.182
7	0.689	0.764	0.762	0.121	0.369	0.910	0.769	0.295	0.949	0.478
8	0.744	0.436	0.286	0.842	0.444	0.811	0.439	0.326	0.912	0.350
9	1.582	0.381	0.797	0.336	0.348	0.299	0.729	1.534	1.253	2.192
10	0.461	0.558	0.097	0.272	0.014	0.554	0.867	0.064	0.046	0.271
11	0.171	0.112	0.253	0.648	0.132	0.310	0.833	-.006	0.894	0.457
12	0.305	0.683	0.858	0.811	0.048	0.066	0.459	0.748	0.135	0.622
13	0.987	0.426	0.506	0.716	0.265	0.810	0.904	0.038	0.095	0.401
14	-1.34	0.607	0.879	-.928	0.139	0.305	1.294	-3.27	0.886	4.831
15	0.361	0.702	0.559	0.654	0.577	0.250	0.138	0.259	0.475	1.127
16	0.190	0.345	0.330	0.150	0.622	0.207	0.944	0.621	0.206	0.484
17	0.285	0.247	0.687	0.871	0.951	0.803	0.845	1.037	0.678	0.888
18	0.931	0.177	0.891	0.767	0.252	0.105	0.530	1.038	0.341	0.684
19	0.257	0.751	0.472	0.370	0.633	0.955	0.189	0.824	0.058	0.808
20	0.670	0.802	0.060	0.760	0.619	0.924	0.362	0.203	0.326	0.556

W_{ijl}, $l = 2$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.481	0.527	0.500	0.287	0.071	0.983	0.724	0.105	0.828	0.176
2	0.263	0.415	0.884	0.053	0.308	0.354	0.590	0.256	0.829	0.480
3	0.887	0.411	0.549	0.967	0.348	0.406	0.165	0.184	0.327	0.452
4	0.792	0.150	0.695	0.729	0.840	0.685	0.954	0.434	0.523	0.029
5	0.313	0.382	0.955	0.626	0.812	0.430	0.739	0.040	0.110	0.542
6	0.795	0.751	0.408	0.477	0.343	0.392	0.025	0.485	0.716	0.237
7	0.020	0.603	0.016	0.760	0.129	0.345	0.220	0.526	0.198	0.788
8	0.342	0.433	0.514	0.892	0.569	0.845	0.994	0.113	0.565	0.032
9	0.683	2.302	1.986	0.797	0.568	0.905	1.981	0.617	1.542	0.509
10	0.300	0.894	0.129	0.946	0.428	0.061	0.784	0.422	0.421	0.702
11	0.576	0.269	0.390	0.691	0.758	0.132	0.136	0.110	0.305	0.863
12	0.106	0.830	0.096	0.702	0.575	0.077	0.809	0.467	0.570	0.308
13	0.980	0.418	0.869	0.879	0.305	1.003	0.055	0.025	0.992	0.699
14	1.752	-.020	1.501	0.062	1.793	-.851	2.434	-.222	0.258	0.278
15	0.425	0.992	0.261	0.535	0.636	0.117	1.044	0.399	0.206	0.557
16	0.839	0.441	0.118	0.920	0.674	0.157	0.781	0.698	0.185	0.777
17	0.114	0.790	0.217	0.363	0.478	0.407	0.588	0.775	0.891	0.599
18	0.556	0.274	1.095	0.272	1.030	0.844	0.630	0.776	0.611	0.391
19	0.670	0.132	1.076	0.383	0.884	0.649	0.118	0.822	0.574	0.266
20	0.874	0.948	0.030	0.405	0.767	0.662	0.868	0.794	0.569	0.587

H_{ijl}, $l = 3$

i\j	1	2	3	4	5	6	7	8	9	10
1	0.493	0.557	0.299	0.772	0.407	0.807	0.022	0.173	-.040	0.969
2	0.287	0.495	0.695	0.204	0.646	0.772	0.968	0.961	-.048	0.895
3	0.385	0.033	0.832	0.649	0.159	0.619	0.573	0.902	-.026	0.339
4	0.409	0.068	0.103	0.054	0.320	0.578	0.100	-.116	1.930	0.823
5	0.562	0.601	0.764	0.980	0.306	0.980	0.494	0.645	0.614	0.263
6	-.448	0.744	0.744	0.250	0.548	0.503	0.725	0.616	0.538	0.843
7	1.276	0.231	0.422	0.319	1.017	0.511	0.731	0.646	-.013	0.410
8	1.122	0.750	0.309	0.382	0.467	0.340	0.465	0.718	0.184	0.753
9	-.380	0.198	0.836	0.249	0.510	-.086	0.287	0.256	-.267	0.385

H_{ijl}, $l = 3$, continued

i\j	11	12	13	14	15	16	17	18	19	20
1	0.921	0.035	0.747	-.114	0.793	0.048	0.311	0.059	0.357	0.153
2	0.444	0.005	0.749	-.117	0.114	0.096	0.604	0.031	0.754	0.155
3	0.127	0.064	0.465	-.084	0.652	0.693	-.012	0.241	0.272	0.781
4	0.119	0.339	0.238	-.954	0.510	0.431	0.260	0.498	0.368	0.503
5	0.517	0.162	0.420	1.742	0.731	0.323	0.774	0.009	0.198	0.274
6	0.065	0.481	0.550	2.222	0.587	0.521	-.300	0.200	0.369	0.995
7	-.106	0.223	0.609	-.504	-.064	0.570	0.365	1.099	0.611	0.350
8	0.600	0.823	1.016	0.117	1.054	0.342	0.824	1.233	1.322	0.075
9	0.350	-.250	0.044	-1.37	0.043	0.356	-.544	-.362	0.741	0.820

Appendix C

```
/***************************************************************
*        This cre.c program need input data such as            *
*                  - version number                            *
*                  - number of patterns used for training      *
*                  - number of nodes in each layer             *
*                  - learning rate kappa and momentum term eta *
*                  - criteria of stable condition for learning *
*                     parameters ,test                         *
*         The output is rnd numbors for learning parameters    *
*         and input and desired output patterns                *
*               LAST UPDATE FEBRUARY 27 1990                   *
***************************************************************/

#define PI   3.1415926538
#include <stdio.h>
#include <math.h>


main()
{
int np, version;
int nl;
int n[4];
int j,layer,node, pattern, dode;
double srand48();
double drand48();
double cos();
float theta, etta, kappa, test;
nl    = 3;


scanf("%d", &version);
scanf("%d", &np);
scanf("%d %d %d %d", &n[0], &n[1], &n[2] , &n[3]);
scanf("%f", &etta);
scanf("%f", &kappa);
scanf("%f", &test);

printf("%d\n", version);
printf("%d\n", np);
printf("%d %d %d %d\n", n[0],n[1],n[2],n[3]);
printf("%f %f\n", etta, kappa);
printf("%f\n", test);

/* printout th,np, and nl */
if (version == 1 || version == 2){
for (layer =1; layer<=nl; ++layer) {
printf("\n");
for (node = 1; node<=n[layer]; ++node) {
printf("%f %f %f\n" , drand48(), drand48(),drand48());

}
}
}
else if (version == 3){                       /* printout th only */
for (layer =1; layer<nl+1; ++layer) {
for (node = 1; node<n[layer]+1; ++node) {
printf("%f" , drand48());
printf("\n");

}
}
}

else {
printf("error in version number...");
goto end;
}

/* printout weights w */
printf("\n");
for (layer =1; layer<nl+1; ++layer) {
for (node = 1; node<=n[layer]; ++node) {
for (j = 1; j<=n[layer-1]; j++) {
printf("%f ", drand48());
       if (j%5 == 0 ){
            printf("\n");
                    }
}
            printf("\n");
}
            printf("\n");
}
            printf("\n");

/* create patterns-simple patterns */
theta = 0.0;
for (pattern = 1 ; pattern <= np ;++pattern) {
printf("%f %f %f\n", (1.0/3.0) * cos(theta),(2.0/3.0) * cos(theta)
                       ,cos(theta));
printf("%f\n",1.0 * cos(theta));
theta + = (PI/2.0)/(np);
}

end:
printf("\n");
}
```

```
        /*********************************************/
        /*      WIBAWA UTAMA                         */
        /*      Dept. of Electrical & Comp Eng.      */
        /*      NJIT Newark, New Jersey 07102        */
        /*-------------------------------------------*/
        /*      Last update February 26, 1990        */
        /*-------------------------------------------*/
        /*      TRANSFORMING 2-D IMAGES INTO 3-D     */
        /*********************************************/

#define PI   3.1415926536
#include <stdio.h>
#include <math.h>


main()
{                               /*start of main()*/

double sqrt(),cos(),sin();
double vers();              /* return absolut value of 1-cos(q) */

double  x2d[5][50], y2d[5][50];                  /* orig. 2-d vectors */
double  x3d[5][50], y3d[5][50], z3d[5][50];    /* vectors in 3-d */
/*          |   |
            |   +---> # of patterns
            +-------> # of joints                           */

int np,nsyn,nj,joint,pattern;
double r1,r2,r3,rx,ry,rz,rot,phi,x,y,z;
double a11, a12, a13,
       a21, a22, a23,   /* transformation matrix */
       a31, a32, a33;

   /* input original 2-d vectors */

 scanf("%d", &np);                      /* number of vectors or patterns */
  scanf("%d", &nsyn);                   /* number of synthesis pattaerns x,y */
  scanf("%d", &nj);                     /* number of joints in one arm */
   scanf("%f %f %f", &r1, &r2, &r3); /*read orientation vector r */
    scanf("%f", &rot);                  /* rotation angle in degree */
      phi = rot * (PI/180.0);           /* convert into radian */

   /* read input vectors */
   for (pattern =1; pattern <=  np; pattern++){
     for (joint = 1 ; joint <= nj; joint ++){
       scanf("%f", &y2d[joint][pattern]);
    }
    }

   /* read synthesis input vectors */
   for (pattern =np+1; pattern <= np+nsyn; pattern++){
      joint = 3;                            /* end-effector */
       scanf("%f", &x2d[joint][pattern]);
       scanf("%f", &y2d[joint][pattern]);
    }
   /* calculate the others ordinates (x) */
   for (pattern =1; pattern <=  np; pattern++){
     x= 0.0;
```



```
      y2d[0][0]=0.0;
        for (joint = 1 ; joint <= nj; joint ++){
             x += sqrt( (1/3.0)*(1/3.0)
                  -(y2d[joint][pattern]-y2d[joint-1][pattern])*
                   (y2d[joint][pattern]-y2d[joint-1][pattern]) );
             x2d[joint][pattern] = x;
      }
      }


    /* calculate coordinate of unit vector r , (rx,ry,rz) */
      rx = r1/sqrt(r1*r1 + r2*r2 + r3*r3);
      ry = r2/sqrt(r1*r1 + r2*r2 + r3*r3);
      rz = r3/sqrt(r1*r1 + r2*r2 + r3*r3);

 /* calculate transformation matrix */

     a11 = rx*rx*vers(phi) + cos(phi);
     a12 = rx*ry*vers(phi) - rz*sin(phi);
     a13 = rx*rz*vers(phi) + ry*sin(phi);

     a21 = rx*ry*vers(phi) + rz*sin(phi);
     a22 = ry*ry*vers(phi) + cos(phi);
     a23 = ry*rz*vers(phi) - rx*sin(phi);

     a31 = rx*rz*vers(phi) - ry*sin(phi);
     a32 = ry*rz*vers(phi) + rx*sin(phi);
     a33 = rz*rz*vers(phi) + cos(phi);

 /* calculate transformed input vectors in 3d */
   for (pattern =1; pattern <=  np; pattern++){
      for (joint = 1 ; joint <= nj; joint ++){
         x3d[joint][pattern] = a11* x2d[joint][pattern]
                             + a12* y2d[joint][pattern];
           y3d[joint][pattern] = a21* x2d[joint][pattern]
                             + a22* y2d[joint][pattern];
             z3d[joint][pattern] = a31* x2d[joint][pattern]
                             + a32* y2d[joint][pattern];
      }
      }
 /* calculate transformed synthesis input vectors in 3d */
   for (pattern =np+1; pattern <= np+nsyn; pattern++){
      joint = 3;                                /* end-effector */
         x3d[joint][pattern] = a11* x2d[joint][pattern]
                             + a12* y2d[joint][pattern];
           y3d[joint][pattern] = a21* x2d[joint][pattern]
                             + a22* y2d[joint][pattern];
             z3d[joint][pattern] = a31* x2d[joint][pattern]
                             + a32* y2d[joint][pattern];
      }
#ifdef DEBUG
  /* print out result for basic-graphic program only*/
for (pattern =1; pattern <=  np; pattern++){
    for (joint = 1 ; joint <= nj; joint ++){
       printf("DATA %f %f %f\n",
              x2d[joint][pattern],y2d[joint][pattern],0.0);
     }
     }
for (pattern =1; pattern <=  np; pattern++){
```

```
    for (joint = 1 ; joint <= nj; joint ++){
       printf("DATA# %f %f %f\n",
               x3d[joint][pattern], y3d[joint][pattern],
                  z3d[joint][pattern]);
     }
     }
#endif

for (pattern =1; pattern <=  np; pattern++){
    printf("\n");
    for (joint = 1 ; joint <= nj; joint ++){
       printf("%f %f %f\n",
               x3d[joint][pattern],y3d[joint][pattern]
                ,z3d[joint][pattern]);
      }
          printf("%f %f %f\n",
               x3d[joint-1][pattern],y3d[joint-1][pattern],
                 z3d[joint-1][pattern]);
      }

   for (pattern =np+1; pattern <= np+ncyn; pattern++){
    printf("\n");
    for (joint = 1 ; joint <= nj; joint ++){
      if (joint == 3){
       printf("%f %f %f\n",
               x3d[joint][pattern],y3d[joint][pattern]
                ,z3d[joint][pattern]);
      }
         else {
          printf("%f %f %f\n", 0.0,0.0,0.0);       /* fake data */
         }
      }
          printf("%f %f %f\n",
               x3d[joint-1][pattern],y3d[joint-1][pattern],
                 z3d[joint-1][pattern]);
      }


} /* end of main() */

double vers(s)
double s;
{
 return(1- cos(s));
}
```



```
        /*********************************************/
        /*      WIBAWA UTAMA                         */
        /*      Dept. of Electrical and Computer Eng. */
        /*      NJIT Newark, New Jersey 07102        */
        /*-------------------------------------------*/
        /*      Last update March 12, 1990           */
        /*-------------------------------------------*/
        /*               BPNV. C                     */
        /*      THREE LAYER PERCEPTRON WITH          */
        /*      BACK PROPAGATION ALGORITHM           */
        /*********************************************/

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

main()
{                               /*start of main()*/

double exp();
double fabs();                /* return absolut value of a-b */
double quad();                /* return x*x */
/*          +------> max # of nodes in output/input (xd/xi) layer
            |   +--> max # of patterns
            |   |                              */
double  xd[50][100],          /* desired output patterns/supervised vectors */
        xi[50][100],          /* input  patterns or command input vectors   */
        xt[50][100];          /* record for checking when to stop */

/*          +------> max # of nodes in each layer
            |  +---> # of layers - three layer perceptron,
            |  |          input layer is not counted as layer.
            |  |                                          */
double   x[50][3],            /* output nodes in hidden layers */
       net[50][3],            /* (sum of W*X)+ th for each node */
   thoenet[50][3];            /* derivatives of error E with respect to net */

/*         +------------> max # of nodes in l th layer-subscript i
           |    +--------> max # of nodes in (l-1)th layer-subscript j
           |    |    +----> # of layers-subscript l
           |    |    |                              */
double   w[50][50][3],        /* weigtht to be trained */
        fw[50][50][3],        /* father of w */
        gw[50][50][3],        /* grand father of w */
        dw[50][50][3];        /* delta or gradient  w */

/*         +-------------> max # of nodes in each layer
           |    +---------> max # of layers
           |    |                                      */
double  th[50][3],            /* threshold theta to be trained */
       fth[50][3],            /* father of th */
       gth[50][3],            /* grand father of th */
       dth[50][3];            /* delta or gradient th */

double  up[50][3],            /* upsilon to be trained */
       fup[50][3],            /* father of up */
       gup[50][3],            /* grand father of up */
       dup[50][3];            /* delta or gradient of alpha */
```

```c
double  al[50][3],          /* alpha to be trained */
        fal[50][3],         /* father of al */
        gal[50][3],         /* grand father of al */
        dal[50][3];         /* delta or gradient of alpha */

double etta, kappa;         /* learning rate & momentum term */
double test;                /* stoping criteria for this program */
double sigma, sum;          /* least square error */
double a, b, c, z, s, f;

int version;            /* version 1. alpha and upsilon, together with
                           w and th, are included as learning parameters
                           output nodes always positive values
                           version 2. idem version 1 except nodes output
                           can be negative values
                           version 3. alpha and upsilon are not included
                           output nodes always positive as version 1.  */
int np, nop;                /* number of pattern */
int nl=3;               /*  three layer perceptron (feed-forward nn) */
int pattern, layer, node, j, i;   /* looping counters */
long int itcount;                 /* iteration counter */
long int nc=32768;                /* iteration limit */
int n[4];                         /* store # of nodes in each layer */


/**** read data from backprop.dat and initialize parameters ****/
   scanf("%d",&version);
     scanf("%d",&np);
       scanf("%d %d %d %d", &n[0],&n[1],&n[2],&n[3]);
         scanf("%f %f", &etta, &kappa);
            scanf("%f", &test);
  if ( (version == 1) || (version == 2) ){
    for (layer = 1 ; layer <= nl ; ++layer){
      for (node = 1 ; node <= n[layer] ; ++node){
          scanf("%f %f %f", &th[node][layer],
                          &up[node][layer],&al[node][layer]);
              net[node][layer] = 0;
                thonet[node][layer] = 0;
                  gth[node][layer] = fth[node][layer] = th[node][layer];
                  gup[node][layer] = fup[node][layer] = up[node][layer];
                  gal[node][layer] = fal[node][layer] = al[node][layer];
                    dth[node][layer] = 0;
                      dup[node][layer] = 0;
                        dal[node][layer] = 0;
   }
   }
   }   /* end of if version 1 or version 2 */

  else if (version == 3){
    for (layer = 1 ; layer <= nl ; ++layer){
      for (node = 1 ; node <= n[layer] ; ++node){
        scanf("%f", &th[node][layer]);
          x[node][layer] = 0;
            net[node][layer] = 0;
              thonet[node][layer] = 0;
                gth[node][layer] = fth[node][layer] = th[node][layer];
                  dth[node][layer] = 0;
   }
```



```c
   }
   }   /* end of if version 3 */

  else {
    printf("error in version number, please check it. bye..\n");
     goto end;
   }

  for (layer = 1 ; layer <= nl ; ++layer){
      for (node  = 1 ; node  <= n[layer] ; ++node){
         for (j = 1; j <= n[layer-1] ; ++j){
   scanf("%f ", &w[node][j][layer]);
             gw[node][j][layer]=fw[node][j][layer]= w[node][j][layer];
      dw[node][j][layer] = 0;
    }
    }
    }

   /* input pair of pattern */

   for (pattern =1; pattern <= np; pattern++){
     for (node = 1 ; node <= n[nl]; node++){
       scanf("%f", &xd[node][pattern]);
    }
           for (node =1 ; node <= n[0]; node++){
              scanf("%f", &xi[node][pattern]);
    }
    }
/********* end of read data and intialized parameters **********/

    if (version == 1){
     do{
        itcount += 1;
        if (itcount == nc) goto stp;
        /* beginning of forward pass */
        for (pattern = 1 ; pattern <= np; ++pattern){
          for (layer = 1 ; layer <= nl ; ++ layer){
            if (layer== 1){                           /* input layer */
              for (node = 1 ; node <= n[layer]; ++node){
                z = 0.0;
                  for (j = 1 ; j <= n[layer-1]; ++j){
                    z+=w[node][j][layer] * xi[j][pattern];
     }

                     net[node][layer] = z + th[node][layer];
                       f=exp(-al[node][layer]*net[node][layer]);
                         x[node][layer]=up[node][layer]/(1+f);
     }
     }                                          /* end of input layer */


              else {                 /* hidden layer to output layer */
                for (node = 1 ; node <= n[layer] ; ++node){
                  z = 0.0;
                    for (j = 1 ; j <= n[layer-1] ; ++j){
                      z+=w[node][j][layer] * x[j][layer-1];
     }

                        net[node][layer] = z + th[node][layer];
                          f=exp(-al[node][layer]*net[node][layer]);
                            x[node][layer]=up[node][layer]/(1+f);
```

```
    }
    }                                             /* end of else */
    }                                             /* end of layer */

    /* end of forward pass */


    /* beginning of backward pass */
    for (layer= nl; layer >0; --layer){
      if (layer==nl){ /* output layer */
        for (node=1; node <= n[nl] ; ++node){
/*****************************************************************/
          xt[node][pattern] = x[node][nl];
/*****************************************************************/
          a = - (xd[node][pattern] - x[node][nl]);
           b = x[node][nl]/up[node][nl];
            c = x[node][nl] * (1.0 - b);
             thoenet[node][nl] = a * c * al[node][nl];
              for (j =1 ; j <= n[nl-1] ; ++j){
                dw[node][j][nl] = - etta * thoenet[node][nl] * x[j][nl-1];
                 w[node][j][nl]  = fw[node][j][nl] + dw[node][j][nl] +
                           kappa * (fw[node][j][nl] - gw[node][j][nl]);
                   gw[node][j][nl] = fw[node][j][nl];
                    fw[node][j][nl] = w[node][j][nl];
     }                                                /* next j */
                      dth[node][nl] = - etta * a * c * al[node][nl];
                      dup[node][nl] = - etta * a * b;
                      dal[node][nl] = - etta * a * c * net[node][nl];
                       th[node][nl]  = fth[node][nl] + dth[node][nl] +
                              kappa * (fth[node][nl] - gth[node][nl]);
                       up[node][nl]  = fup[node][nl] + dup[node][nl] +
                              kappa * (fup[node][nl] - gup[node][nl]);
                       al[node][nl]  = fal[node][nl] + dal[node][nl] +
                              kappa * (fal[node][nl] - gal[node][nl]);
                        gth[node][nl] = fth[node][nl];
                         fth[node][nl] = th[node][nl];
                          gup[node][nl] = fup[node][nl];
                           fup[node][nl] = up[node][nl];
                             gal[node][nl] = fal[node][nl];
                              fal[node][nl] = al[node][nl];
     }                                                /* next node */
     }                                        /* end of output layer */
      else {                    /*hidden layer to lower/input layer */
       for (node=1; node <= n[layer] ; ++node){
         a = 0.0;
          for (i =1 ; i <= n[layer-1] ; ++i){
            a += thoenet[i][layer+1] * w[i][node][layer+1];
     }
           b = x[node][layer]/up[node][layer];
           c = x[node][layer] * (1.0 - b);
           thoenet[node][layer] = a * c * al[node][layer];
            for (j=1; j <= n[layer-1]; ++j){
              dw[node][j][layer] =
                      - etta * thoenet[node][layer] * x[j][layer-1];
              w[node][j][layer]  =
                      fw[node][j][layer] + dw[node][j][layer] +
                      kappa * (fw[node][j][layer] - gw[node][j][layer]);
              gw[node][j][layer] = fw[node][j][layer];
              fw[node][j][layer] = w[node][j][layer];
```



```
          }                                                  /* next j */
                   dth[node][layer] = - etta * a * c * al[node][layer];
                   dup[node][layer] = - etta * a * b;
                   dal[node][layer] = - etta * a * c * net[node][layer];
                    th[node][layer]  =
                            fth[node][layer] + dth[node][layer] +
                            kappa * (fth[node][layer] - gth[node][layer]);
                     up[node][layer]  =
                              fup[node][layer] + dup[node][layer] +
                              kappa * (fup[node][layer] - gup[node][layer]);
                       al[node][layer]  =
                          fal[node][layer] + dal[node][layer] +
                          kappa * (fal[node][layer] - gal[node][layer]);
                              gth[node][layer] = fth[node][layer];
                                fth[node][layer] = th[node][layer];
                                  gup[node][layer] = fup[node][layer];
                                    fup[node][layer] = up[node][layer];
                                      gal[node][layer] = fal[node][layer];
                                        fal[node][layer] = al[node][layer];
          }                                          /* next node */
          }                                          /*end of else*/
          }                                          /* next layer */
          /* end of backward pass */
          }                                          /* next pattern */

   /* check when to stop */
 if(test ==1){
  goto skip1;   /****** skip this part of program, save time... *****/
   }
    else{
    sigma = 0.0;
    sum = 0.0;
    for (pattern = 1 ; pattern <= np ; ++pattern)      {
     for (node= 1 ; node <= n[nl]; ++node)          {
            a = quad(xd[node][pattern]- xt[node][pattern]);
              sum = sum + a ;
      }
      }
                     sigma = 0.5 * sum;
   }
 skip1:
     nop = 1; /** nop = no operation **/
 } /* end of do */
      while((sigma > test) || (test == 1));  /* end of do while */
     } /* end of version 1 */



 /* begining of version 2 */

     if (version == 2){
      do{
         itcount += 1;
         if (itcount ==  nc) goto stp;
         /* beginning of forward pass */
         for (pattern = 1 ; pattern <= np; ++pattern){
           for (layer = 1 ; layer <= nl ; ++ layer){
             if (layer== 1){                           /* input layer */
               for (node = 1 ; node <= n[layer]; ++node){
```

```
                z = 0.0;
                 for (j = 1 ; j <= n[layer-1]; ++j){
                   z+=w[node][j][layer] * xi[j][pattern];
    }
                        net[node][layer] = z + th[node][layer];
                           f=exp(-al[node][layer]*net[node][layer]);
                             x[node][layer]=up[node][layer]*
                                             ( 1/(1+f) - 0.5 ) ;
    }
    }                                             /* end of input layer */


           else {                   /* hidden layer to output layer */
             for (node = 1 ; node <= n[layer] ; ++node){
                z = 0.0;
                  for (j = 1 ; j <= n[layer-1] ; ++j){
                     z+=w[node][j][layer] * x[j][layer-1];
    }
                          net[node][layer] = z + th[node][layer];
                            f=exp(-al[node][layer]*net[node][layer]);
                               x[node][layer]=up[node][layer]*
                                         ( 1/(1+f) - 0.5 );
    }
    }                                                    /* end of else */
    }                                                   /* end of layer */

    /* end of forward pass */

    /* beginning of backward pass */
    for (layer= nl; layer >0; --layer){
      if (layer==nl){ /* output layer */
        for (node=1; node <= n[nl] ; ++node){
/***********************************************************************/
           xt[node][pattern] = x[node][nl];
/***********************************************************************/
           a = - (xd[node][pattern] - x[node][nl]);
             b = x[node][nl]/up[node][nl];
               c = (x[node][nl]+ 0.5*up[node][nl]) * (0.5 - b);
                 thoenet[node][nl] = a * c * al[node][nl];
                   for (j =1 ; j <= n[nl-1] ; ++j){
                     dw[node][j][nl] = - etta * thoenet[node][nl] * x[j][nl-1];
                       w[node][j][nl]  = fw[node][j][nl] + dw[node][j][nl] +
                                   kappa * (fw[node][j][nl] - gw[node][j][nl]);
                          gw[node][j][nl] = fw[node][j][nl];
                            fw[node][j][nl] = w[node][j][nl];
     }                                                        /* next j */
                              dth[node][nl] = - etta * a * c * al[node][nl];
                              dup[node][nl] = - etta * a * b;
                              dal[node][nl] = - etta * a * c * net[node][nl];
                               th[node][nl]  = fth[node][nl] + dth[node][nl] +
                                       kappa * (fth[node][nl] - gth[node][nl]);
                               up[node][nl]  = fup[node][nl] + dup[node][nl] +
                                       kappa * (fup[node][nl] - gup[node][nl]);
                               al[node][nl]  = fal[node][nl] + dal[node][nl] +
                                       kappa * (fal[node][nl] - gal[node][nl]);
                                 gth[node][nl] = fth[node][nl];
                                   fth[node][nl] = th[node][nl];
                                     gup[node][nl] = fup[node][nl];
                                       fup[node][nl] = up[node][nl];
```



```
                                                 gal[node][nl] = fal[node][nl];
                                                   fal[node][nl] = al[node][nl];
        }                                                       /* next node */
        }                                               /* end of output layer */
         else {                             /*hidden layer to lower/input layer */

          for (node=1; node <= n[layer] ; ++node){
            a = 0.0;
             for (i =1 ; i <= n[layer-1] ; ++i){
               a += thoenet[i][layer+1] * w[i][node][layer+1];
        }
               b = x[node][layer]/up[node][layer];
               c = (x[node][layer]+ 0.5*up[node][layer]) * (0.5 - b);
               thoenet[node][layer] = a * c * al[node][layer];
                for (j=1; j <= n[layer-1]; ++j){
                  dw[node][j][layer] =
                         - etta * thoenet[node][layer] * x[j][layer-1];
                  w[node][j][layer]  =
                         fw[node][j][layer] + dw[node][j][layer] +
                         kappa * (fw[node][j][layer] - gw[node][j][layer]);
                  gw[node][j][layer] = fw[node][j][layer];
                  fw[node][j][layer] = w[node][j][layer];
         }                                                         /* next j */
                    dth[node][layer] = - etta * a * c * al[node][layer];
                    dup[node][layer] = - etta * a * b;
                    dal[node][layer] = - etta * a * c * net[node][layer];
                     th[node][layer]  =
                             fth[node][layer] + dth[node][layer] +
                             kappa * (fth[node][layer] - gth[node][layer]);
                      up[node][layer]  =
                               fup[node][layer] + dup[node][layer] +
                               kappa * (fup[node][layer] - gup[node][layer]);
                        al[node][layer]  =
                           fal[node][layer] + dal[node][layer] +
                           kappa * (fal[node][layer] - gal[node][layer]);
                               gth[node][layer] = fth[node][layer];
                                 fth[node][layer] = th[node][layer];
                                    gup[node][layer] = fup[node][layer];
                                      fup[node][layer] = up[node][layer];
                                        gal[node][layer] = fal[node][layer];
                                          fal[node][layer] = al[node][layer];
         }                                                      /* next node */
         }                                                      /*end of else*/
         }                                                      /* next layer */
         /* end of backward pass */
         }                                                      /* next pattern */

   /* check when to stop */
  if(test ==1){
   goto skip2;   /****** skip this part of program, save time... *****/
    }
     else{
     sigma =0.0;
     sum = 0.0;
     for (pattern = 1 ; pattern <= np ; ++pattern)      {
      for (node= 1 ; node <= n[nl]; ++node)          {
              a = quad(xd[node][pattern] - xt[node][pattern]);
                sum = sum + a ;
      }
```

```
        }
                        sigma = 0.5 * sum;
    }
  skip2:
      nop = 1; /** nop = no operation **/
  } /* end of do */
      while((sigma > test) || (test == 1));  /* end of do while */
    } /* end of version 2*/



   if (version == 3) {
    do{
      itcount += 1;
      if (itcount == nc) goto stp;
     /* beginning of forward pass */
      for (pattern = 1 ; pattern < np+1; ++pattern){
       for (layer = 1 ; layer < nl+1 ; ++ layer){
        if (layer== 1){                                /* input layer */
         for (node = 1 ; node < n[layer]+1; ++node){
          z = 0.0;
           for (j = 1 ; j < n[layer-1]+1; ++j){
              z+=w[node][j][layer] * xi[j][pattern];
                                                     }
                net[node][layer] = z + th[node][layer];
                  f=exp(-net[node][layer]);
                    x[node][layer]=1/(1+f);
                   }
                                                     } /* end of input layer */


         else { /* hidden-output layer*/
          for (node = 1 ; node < n[layer]+1 ; ++node){
            z = 0.0;
              for (j = 1 ; j < n[layer-1]+1 ; ++j){
                z+=w[node][j][layer] * x[j][layer-1];
                              }
                  net[node][layer] = z + th[node][layer];
                    f=exp(-net[node][layer]);
                      x[node][layer]=1/(1+f);
                                                     }
                                                     }  /* end of else */
                                                     }  /* end of layer */

 /* end of forward pass */

 /* beginning of backward pass */
   for (layer= nl; layer >0; --layer){
    if (layer==nl){                                   /* output layer */
     for (node=1; node < n[nl]+1 ; ++node){
/*********************************************************************/
          xt[node][pattern] = x[node][nl];
/*********************************************************************/
      a = - (xd[node][pattern] - x[node][nl]);
      b = x[node][nl];
      c = x[node][nl] * (1.0 - b);
      thoenet[node][nl] = a * c ;
        for (j =1 ; j < n[nl-1]+1 ; ++j){
          dw[node][j][nl] = - etta * thoenet[node][nl] * x[j][nl-1];
```



```
                w[node][j][nl]  = fw[node][j][nl] + dw[node][j][nl] +
                                  kappa * (fw[node][j][nl] - gw[node][j][nl]);
                  gw[node][j][nl] = fw[node][j][nl];
                    fw[node][j][nl] = w[node][j][nl];
                                                     }  /* next j */
                       dth[node][nl] = - etta * a * c ;
                         th[node][nl]  = fth[node][nl] + dth[node][nl] +
                                       kappa * (fth[node][nl] - gth[node][nl]);
                           gth[node][nl] = fth[node][nl];
                             fth[node][nl] = th[node][nl];
                                                     } /* next node */
                                                     } /* end of output layer */
 else {                                          /* hidden-input layer */
        for (node=1; node < n[layer]+1 ; ++node){
         a = 0.0;
          for (i =1 ; i < n[layer-1]+1 ; ++i){
            a += thoenet[i][layer+1] * w[i][node][layer+1];
                                                     }
            b = x[node][layer];
            c = x[node][layer] * (1.0 - b);
            thoenet[node][layer] = a * c ;
              for (j=1; j < n[layer-1]+1; ++j){
               dw[node][j][layer] =
                                  - etta * thoenet[node][layer] * x[j][layer-1];
                w[node][j][layer]  =
                               fw[node][j][layer] + dw[node][j][layer] +
                               kappa * (fw[node][j][layer] - gw[node][j][layer]);
                  gw[node][j][layer] = fw[node][j][layer];
                   fw[node][j][layer] = w[node][j][layer];
                                                     } /* next j */
                     dth[node][layer] = - etta * a * c ;
                      th[node][layer]  = fth[node][layer] + dth[node][layer] +
                                   kappa * (fth[node][layer] - gth[node][layer]);
                        gth[node][layer] = fth[node][layer];
                         fth[node][layer] = th[node][layer];
                                                     }    /* next node */
                                                     }    /*end of else*/
                                                     }    /* next layer */
  /* end of backward pass */
                                                    } /* next pattern */

   /* check when to stop */
 if(test ==1){
  goto skip3;   /****** skip this part of program, save time... *****/
   }
    else{
    sigma = 0.0;
    sum = 0.0;
    for (pattern = 1 ; pattern <= np ; ++pattern)      {
     for (node= 1 ; node <= n[nl]; ++node)          {
            a = quad(xd[node][pattern] - xt[node][pattern]);
              sum = sum + a ;
      }
      }
                      sigma = 0.5 * sum;
   }
  skip3:
      nop = 1; /** nop = no operation **/
  } /* end of do */
```

```
      while((sigma > test) || (test == 1));  /* end of do while */
    } /* end of version 3 */


/* printout learned parameters and related data */
stp:
    printf("%d\n", version);
      printf("%d\n", np);
        printf("%d %d %d %d\n", n[0],n[1],n[2],n[3]);
        printf ("%f %f\n", etta, kappa);
        printf ("%f %f\n", test, sigma);
        printf ("%d\n", itcount);
        printf ("%d\n", 2*np);

           for (layer = 1 ; layer <= nl ; ++layer)           {
             for (node = 1 ; node <= n[layer] ; ++node)      {
               if ((version == 1) || (version == 2)){
                  printf("%f %f %f", th[node][layer],
                                     up[node][layer],al[node][layer]);
                      printf("\n");
                  }
               if (version == 3){
                    printf("%f", th[node][layer]);
                       printf("\n");
                  }
                                                  }
                                                  }

          for (layer = 1 ; layer <= nl ; ++layer)            {
           for (node = 1 ; node <= n[layer] ; ++node)        {
             for (j = 1; j <= n[layer-1] ; ++j)              {
               printf("%f ", w[node][j][layer]);
                 if (j%5 == 0){
                  printf("\n");
                               }
                                                  }
                  printf("\n");
                                                  }
                  printf("\n");
                                                  }
                  printf("\n");
#ifdef DEBUG
/* input pattern */
           for (pattern =1; pattern <=  np; ++pattern)      {
              for (node =1 ; node <= n[3]; ++node)          {
                 printf("%f ", xd[node][pattern]);
                     if(node%3==0){
                   printf("\n");
                               }
                                                }
                      for (node =1 ; node <= n[0]; ++node)  {
                          printf("%f ", xi[node][pattern]);
                                                }
                               printf("\n");
                                                }
#endif
/* end of printout */
```



```
end:
printf("\n");
}                /* end of main() program */


double fabs(a,b)
double a,b;
{
if (a-b >=0.0)
   return (a-b);
else
   return (-(a-b));
  }


        double quad(x)
        double x;
        {
          return( x * x);
        }
```

```
/****************************************************************
 *                     3 - D  TRAJECTORY                         *
 * Program name :fr.c. This program read data that had already been    *
 * trained by bp.c and  calculate the presented input data through     *
 * forward pass procedure and print-out the comparison between desired *
 * or actual output and NN output.                                     *
 ****************************************************************/

#include <stdio.h>
#include <math.h>
#define PI   3.1415926536



main()
{                               /*start of main()*/

double exp(), sqrt();
        double fabs(),quad();

double   xi[50][50],       /* input  patterns */
                xd[50][50],     /* desired output,just for output comparison */
          x[50][50],      /* output nodes in hidden layers */
        net[50][3];     /* (sum of W*X) + threshold th */

double   w[50][50][3];       /* trained weights */

double  th[50][3];            /* trained threshold */

double  up[50][3];            /* trained upsilon */

double  al[50][3];            /* trained alpha */

double z, f ;
        double xdes[4], ydes[4], zdes[4];
        double xact[4], yact[4], zact[4];
        double zphy[4]; /*reqnired to make the system physically realizable*/
        double err[4];
        double etta, kappa, test, sigma, sum, errtot;
int version,np,nin, nl, pattern, layer, node,dode, j,i ;
        long int itcount;
int n[4];
        int flag;
nl   = 3;

 /* read data from backprop.dat and initialize parameters */

/* read important parameters */
        scanf("%d",&version);
scanf("%d",&np);
        scanf("%d %d %d %d", &n[0],&n[1],&n[2],&n[3]);
        scanf("%f", &etta);
        scanf("%f", &kappa);
        scanf("%f %f", &test, &sigma);
        scanf("%d", &itcount);
        scanf("%d", &nin);
/* read learning paramters */
      if ((version == 1) || (version == 2)){
```



```
         for (layer = 1 ; layer <= nl ; ++layer){
           for (node = 1 ; node <= n[layer] ; ++node){
             scanf("%f %f %f", &th[node][layer],
                             &up[node][layer],&al[node][layer]);
                  x[node][layer] = 0;
        }
        }
        }
          else if (version == 3){
             for (layer = 1 ; layer <= nl ; ++layer){
                for (node = 1 ; node <= n[layer] ; ++node){
                    scanf("%f", &th[node][layer]);
                      up[node][layer]=1.0;
                         al[node][layer]=1.0;
                            x[node][layer] =0.0;
        }
        }
        }
            else {
                 printf("error in version number, please check it. bye..\n");
                  goto end;
        }

        for (layer = 1 ; layer <=nl ; ++layer)                    {
                for (node = 1 ; node  <= n[layer] ; ++node)        {
             for (j = 1; j <= n[layer-1] ; ++j)           {
   scanf("%f ", &w[node][j][layer]);
                }
         }
                      }

 /* end of read data and intialized parameters */

/* initialize input matrix, tobe calculated by NN */
   /****************************************************************
    * NOTE all desired values, xd, must be physically realizable ! *
    ****************************************************************/
          for (pattern=1; pattern <= nin; ++pattern){
          for (node =1 ; node <= n[nl];  node = node +1)
             scanf("%f", &xd[node][pattern]);
           for (node = 1; node <= n[0]; ++node)
           scanf ("%f", &xi[node][pattern]);
}

/* print header */
printf("\n");
printf("\n");
printf("\n");
printf (" This 3-layer NN with %d %d %d %d configuration",n[0],n[1],n[2],n[3]);
printf (" has been trained with\n");
if(test== 1){
printf (" %d patterns, etta= %f, kappa= %f\n", np,etta, kappa);
}
else{
printf (" %d patterns, etta= %f, kappa= %f,and stoping criteria",np,etta,kappa);
printf (" test=%f\n",test);
}
printf (" version = %d  and training time = %d iteration\n",version,itcount);
printf("\n");
```

```
printf("   DESIRED VALUES        ** OUTPUTS \n");
printf("  X       Y       Z     X      Y      Z    Zphy   ");
printf("XD-XN  YD-YN  ZD-ZP  LS. ERR.\n");
printf("==================================================");
printf("===========================\n");
printf("\n");

  /* begining of forward pass */
    pattern = 0.0;
    while (++pattern <=nin  ){
      for (layer = 1 ; layer <= nl ; ++ layer){
        if (layer== 1){                              /* input layer */
          for (node = 1 ; node <= n[layer]; ++node){
             z = 0.0;
                for (j = 1 ; j <= n[layer-1]; ++j){
                   z+=w[node][j][layer] * xi[j][pattern];
    }
                     net[node][layer] = z + th[node][layer];
                         f=exp(-al[node][layer]*net[node][layer]);
                        if ((version == 1) || (version == 3)){
                           x[node][layer]=up[node][layer]/(1+f);
                        }
                     if (version == 2 ) {
                   x[node][layer] = up[node][layer] *
                                  (1/(1+f) - 0.5);
                     }
    }
    }                                       /* end of input layer */

       else {                          /* hidden layer to output layer */
        for (node = 1 ; node <= n[layer] ; ++node){
           z = 0.0;
              for (j = 1 ; j <= n[layer-1] ; ++j){
                 z+=w[node][j][layer] * x[j][layer-1];
    }
                       net[node][layer] = z + th[node][layer];
                         f=exp(-al[node][layer]*net[node][layer]);
                       if ((version == 1) || (version == 3))
                          x[node][layer]=up[node][layer]/(1+f);

                     if (version == 2 )
                   x[node][layer] = up[node][layer] *
                                  (1/(1+f) - 0.5);

    }
    }                                          /* end of else */
    }                                          /* end of layer */

 /* end of forward pass */

    /* printout result */

    /* initialization */
      xdes[0]=xdes[1]=xdes[2]=xdes[3] = 0.0;
      ydes[0]=ydes[1]=ydes[2]=ydes[3] = 0.0;
```



```
      zdes[0]=zdes[1]=zdes[2]=zdes[3] = 0.0;
      xact[0]=xact[1]=xact[2]=xact[3] = 0.0;
      yact[0]=yact[1]=yact[2]=yact[3] = 0.0;
      zact[0]=zact[1]=zact[2]=zact[3] = 0.0;
      zphy[0]=zphy[1]=zphy[2]=zphy[3] = 0.0;

      for (node = 1 ; node <= n[3] ; node= node + 3){
/*********************************************************
 * NOTE all desired values must be physically realizable ! *
 *********************************************************/
        if (node == 1){
        xdes[1]= xd[node][pattern];
        ydes[1]= xd[node+1][pattern];
        zdes[1]= xd[node+2][pattern];
        xact[1]= x[node][nl];
        yact[1]= x[node+1][nl];
        zact[1]= x[node+2][nl];
        if (zact[1]-zact[0] < 0){
        zphy[1] = zphy[0] - sqrt( (1/3.0)*(1/3.0)
                                 -quad(xact[1]-xact[0])
                                 -quad(yact[1]-yact[0]) );
        }
        else {
        zphy[1] = zphy[0] + sqrt( (1/3.0)*(1/3.0)
                                 -quad(xact[1]-xact[0])
                                 -quad(yact[1]-yact[0]) );
        }

        if (xdes[1] == 9.0){
        printf ("-----  -----  -----  ");
        }
        else {
        printf ("%.3f  %.3f  %.3f  ", xdes[1],ydes[1],zdes[1]);
        }

        printf ("%.3f  %.3f  %.3f  %.3f  ", xact[1],yact[1],zact[1],zphy[1]);

        /* calculating and printing error */
        if (xdes[1] == 9.0){
        printf ("-----  -----  -----  -----\n");
        }
        else {
        err[1] = quad(xdes[1]-xact[1]) + quad(ydes[1]-yact[1])
              + quad(zdes[1]-zphy[1]);
        printf ("%.3f  %.3f  %.3f  %.3f\n", xdes[1]-xact[1],
                ydes[1]-yact[1], zdes[1]-zphy[1], sqrt(err[1]));
        }

        } /*end of node = 1 */

        if (node == 4){
        xdes[2]= xd[node][pattern];
        ydes[2]= xd[node+1][pattern];
        zdes[2]= xd[node+2][pattern];
        xact[2]= x[node][nl];
        yact[2]= x[node+1][nl];
        zact[2]= x[node+2][nl];
        if (zact[2] - zact[1] < 0){
        zphy[2] = zphy[1] - sqrt( (1/3.0)*(1/3.0)
```

```
                              -quad(xact[2]-xact[1])
                              -quad(yact[2]-yact[1]) );
}
else {
zphy[2] = zphy[1] + sqrt( (1/3.0)*(1/3.0)
                              -quad(xact[2]-xact[1])
                              -quad(yact[2]-yact[1]) );
}
if (xdes[2] == 9.0){
printf ("-----  -----  -----  ");
}
else {
printf ("%.3f  %.3f  %.3f  ", xdes[2],ydes[2],zdes[2]);
}
printf ("%.3f  %.3f  %.3f  %.3f  ", xact[2],yact[2],zact[2],zphy[2]);

/* calculating and printing error  */
if (xdes[2] == 9.0){
printf ("-----  -----  -----  -----\n");
}
else {
err[2] = quad(xdes[2]-xact[2]) + quad(ydes[2]-yact[2])
      + quad(zdes[2]-zphy[2]);
printf ("%.3f  %.3f  %.3f  %.3f\n", xdes[2]-xact[2],
        ydes[2]-yact[2], zdes[2]-zphy[2], sqrt(err[2]));
}
} /* end of node = 4 */


if (node == 7){
xdes[3]= xd[node][pattern];
ydes[3]= xd[node+1][pattern];
zdes[3]= xd[node+2][pattern];
xact[3]= x[node][n1];
yact[3]= x[node+1][n1];
zact[3]= x[node+2][n1];
if (zact[3]  - zact[2] < 0){
zphy[3] = zphy[2] - sqrt( (1/3.0)*(1/3.0)
                              -quad(xact[3]-xact[2])
                              -quad(yact[3]-yact[2]) );
}
else {
zphy[3] = zphy[2] + sqrt( (1/3.0)*(1/3.0)
                              -quad(xact[3]-xact[2])
                              -quad(yact[3]-yact[2]) );
}
if (xdes[3] == 9.0){
printf ("-----  -----  -----  ");
}
else {
printf ("%.3f  %.3f  %.3f  ", xdes[3],ydes[3],zdes[3]);
}
printf ("%.3f  %.3f  %.3f  %.3f  ", xact[3],yact[3],zact[3],zphy[3]);

/* calculating and printing error */
if (xdes[3] == 9.0){
printf ("-----  -----  -----  -----\n");
}
else {
```



```
          err[3] = quad(xdes[3]-xact[3]) + quad(ydes[3]-yact[3])
                + quad(zdes[3]-zphy[3]);
          printf ("%.3f  %.3f  %.3f  %.3f\n", xdes[3]-xact[3],
                  ydes[3]-yact[3], zdes[3]-zphy[3], sqrt(err[3]));
          }
          } /* end of node = 7 */
         }                                            /* end of node */

              sum = sqrt(err[1]) + sqrt(err[2]) + sqrt(err[3]);
            errtot = errtot + sum;
          printf ("\n");
         }                                            /* end of while */
end:
         printf ("\n");
         printf ("E = %f, total error = %f\n", sigma, errtot);
         }                                  /* end of main() program */



double fabs(a,b)
double a,b;
{
if (a-b >=0.0)
   return (a-b);
else
   return (-(a-b));
  }

          double quad(h)
          double h;
          {
            return (h * h);
          }
```