1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Trajectory Control of Robotic Manipulators By Using a Feedback-Error-Learning Neural Network

by

ZARYAB HAMAVAND, B.Eng.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Engineering

Ottawa-Carleton Institute for
Mechanical and Aerospace Engineering

Department of
Mechanical and Aerospace Engineering
Carleton University
Ottawa, Ontario
August 16th, 1994

© Copyright
1994, Zaryab Hamavand

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

**THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.**

**L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.**

**THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.**

**L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.**

ISBN 0-315-98577-1

Canada

Name ZARYAB HAMAVAND

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

ENGINEERING, MECHANICAL

SUBJECT TERM

0548

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
 General 0289
 Ancient 0290
 Linguistics 0291
 Modern 0401
Literature 0294
 General 0295
 Classical 0297
 Comparative 0298
 Medieval 0316
 Modern 0591
 African 0305
 American 0352
 Asian 0355
 Canadian (English) 0593
 Canadian (French) 0311
 English 0312
 Germanic 0315
 Latin American 0313
 Middle Eastern 0314
 Romance 0314
 Slavic and East European 0314

PHILOSOPHY, RELIGION AND

THEOLOGY

Philosophy 0422
Religion 0318
 General 0321
 Biblical Studies 0319
 Clergy 0320
 History of 0322
 Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
 Archaeology 0326
 Cultural 0327
 Physical 0310
Business Administration 0272
 General 0770
 Accounting 0454
 Banking 0338
 Management 0385
 Marketing 0501
Canadian Studies 0503
Economics 0505
 General 0508
 Agricultural 0509
 Commerce-Business 0510
 Finance 0511
 History 0358
 Labor 0366
 Theory 0351
Folklore 0358
Geography 0366
Gerontology 0351
History 0578
 General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
 General 0616
 International Law and Relations 0617
 Public Administration 0814
Recreation 0452
Social Work 0626
Sociology 0627
 General 0938
 Criminology and Penology 0631
 Demography 0628
 Ethnic and Racial Studies 0629
 Individual and Family Studies 0630
 Industrial and Labor Relations 0700
 Public and Social Welfare 0344
 Social Structure and Development 0709
 Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
 General 0285
 Agronomy 0475
 Animal Culture and Nutrition 0476
 Animal Pathology 0359
 Food Science and Technology 0478
 Forestry and Wildlife 0479
 Plant Culture 0480
 Plant Pathology 0817
 Plant Physiology 0777
 Range Management 0746
 Wood Technology 0306
Biology 0287
 General 0308
 Anatomy 0309
 Biostatistics 0379
 Botany 0329
 Cell 0353
 Ecology 0369
 Entomology 0793
 Genetics 0410
 Limnology 0307
 Microbiology 0317
 Molecular 0416
 Neuroscience 0433
 Oceanography 0821
 Physiology 0778
 Radiation 0472
 Veterinary Science 0786
 Zoology 0760
Biophysics 0425
 General 0996
 Medical 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
 General 0300
 Audiology 0992
 Chemotherapy 0567
 Dentistry 0350
 Education 0769
 Hospital Management 0758
 Human Development 0982
 Immunology 0564
 Medicine and Surgery 0347
 Mental Health 0569
 Nursing 0570
 Nutrition 0380
 Obstetrics and Gynecology 0354
 Occupational Health and Therapy 0381
 Ophthalmology 0571
 Pathology 0419
 Pharmacology 0572
 Pharmacy 0382
 Physical Therapy 0573
 Public Health 0574
 Radiology 0575
 Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry 0485
 General 0749
 Agricultural 0486
 Analytical 0487
 Biochemistry 0488
 Inorganic 0738
 Nuclear 0490
 Organic 0491
 Pharmaceutical 0494
 Physical 0495
 Polymer 0754
 Radiation 0405
Mathematics 0605
Physics 0986
 General 0606
 Acoustics 0608
 Astronomy and Astrophysics 0748
 Atmospheric Science 0607
 Atomic 0798
 Electronics and Electricity 0759
 Elementary Particles and High Energy 0609
 Fluid and Plasma 0610
 Molecular 0752
 Nuclear 0756
 Optics 0611
 Radiation 0463
 Solid State 0346
Statistics 0984

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering 0537
 General 0538
 Aerospace 0539
 Agricultural 0540
 Automotive 0541
 Biomedical 0542
 Chemical 0543
 Civil 0544
 Electronics and Electrical 0348
 Heat and Thermodynamics 0546
 Hydraulic 0547
 Industrial 0794
 Marine 0548
 Materials Science 0743
 Mechanical 0551
 Metallurgy 0552
 Mining 0549
 Nuclear 0765
 Packaging 0554
 Petroleum 0790
 Sanitary and Municipal 0428
 System Science 0796
 Geotechnology 0796
 Operations Research 0795
 Plastics Technology 0994
 Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis
**Trajectory Control of Robotic Manipulators By Using a
Feedback-Error-Learning Neural Network**
submitted by
Zaryab Hamavand, B.Eng.
in partial fulfilment of the requirements for
the degree of Master of Engineering

Professor H.M. Schwartz
Thesis Co-Supervisor

Professor D.L. Russell
Thesis Co-Supervisor

Professor R. Bell
Chairman, Department of Mechanical and Aerospace Engineering

Carleton University
August 16, 1994

Abstract

This thesis presents a neural network based control strategy for trajectory control of robot manipulators. The neural network learns the inverse dynamics of a robot manipulator without any *a priori* knowledge of the manipulator's inertial parameters nor any *a priori* knowledge of the equation of dynamics. A two step feedback-error-learning process is proposed.

Strategies for selection of the training trajectories is discussed. The methods of finding the training trajectories for the simulation and the experiment have been presented. The difficulties with on-line training are discussed and simulation results show these difficulties. A simulation of a two degree of freedom serial link manipulator illustrates the effectiveness of the proposed method. The output of the neural network was compared with the actual inverse dynamic function of the manipulator. The neural network learned the inverse dynamic of the manipulator and was able to follow any arbitrary trajectory in the robot workspace with a high degree of accuracy. Experiments were performed on a two degree of freedom, direct drive manipulator. The experimental results are very good.

Acknowledgements

For offering numerous comments and suggestions on the entire project, I am especially indebted to my supervisor, Professor H.M. Schwartz. His constant guidance and support in my graduate studies, as well as my undergraduate studies, have been a source of encouragement for me.

I also appreciate the help and guidance provided by my Co-supervisor, Professor D.L. Russell.

I would also like to thank Danny Lemay and Dave Sword for their assistance with the care of the robot hardware and Naren Mehta for his help in computer software set up. My appreciation extends to my fellow research colleagues, particularly Gabriel Warshaw, for helping me to perform the experiment.

Dedication

To my parents, Kokab Zaryab and Rostam Hamavand,
who have always had faith in me and showed it
through their encouragement and support.

Contents

Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
Table of Contents	vi
List of Tables	ix
List of Figures	x
1 INTRODUCTION	1
1.1 Trajectory Control	1
1.2 Neural Network Control	2
1.3 Trajectory Control of Robotic Manipulators By Using a Feedback- Error-Learning Neural Network	4
1.4 Thesis Outline	5
2 ARTIFICIAL NEURAL NETWORKS	6
2.1 Neural Networks Basic Ideas	6

2.2	Multilayer Neural Networks	7
2.3	Neural Networks Training Algorithm	11
3	THE NEURAL NETWORK CONTROLLER DESIGN	17
3.1	The Neural Network Controller Structure	17
3.2	Selection Of The Training Trajectory.	19
3.3	Sequential Training	21
3.4	Off line training	23
3.4.1	Stage One training: Learning The PD Controller Output . . .	23
3.4.2	Stage Two Training: Learning The Inverse Dynamics	29
4	SIMULATION	32
4.1	Robot Model	32
4.2	Mapping The PD Controller	35
4.3	Mapping The Inverse Dynamics	37
5	EXPERIMENT	51
5.1	Manipulator Description	51
5.2	Mapping The PD Controller	54
5.3	Mapping The Inverse Dynamics	61
5.4	Trained for the Predefined Trajectory	71
5.5	Discussion	73
6	CONCLUSION	83
A	Prefilter	88
A.1	Prefilter Matrices	89
A.2	Matlab Code	90

B	Simulation Software Listing	91
B.1	First Stage Of Training	92
B.2	Second Stage Of The Training	103
B.3	Neural Network Test	115
C	Experiment Software Listing	122
C.1	Define The Trajectory	123
C.2	Robot Control	125
C.2.1	Main Program	126
C.2.2	Filter Program Listing	143
C.2.3	Datacard Program Listing	150
C.2.4	Marith Program Listing	155
C.3	Training The Neural Network	158

List of Tables

4.1 Link Parameters of the 2 DOF Manipulator.	34
---	----

List of Figures

2.1	physical structure of Neuron cells.	7
2.2	Structure of basic artificial neuron cells (Perceptron).	8
2.3	Structure of multilayer neural network.	9
2.4	Block diagram of multilayer network.	9
2.5	Multilayer neural network with bias term.	10
2.6	Simple example of multilayer network.	13
3.1	The controller model for each manipulator joint.	18
3.2	Nonlinear function of two variable.	20
3.3	Online methods of training (Sequential).	22
3.4	PD controller torque (solid line) and neural network output (dashed line) during online training.	24
3.5	First weight in First node of output layer during online training.	25
3.6	Torque input (solid line) and neural network output (dashed line) when online training has been turned off. The network was trained online.	26
3.7	First stage of training. The solid line shows the data collection part, and the dotted line shows the training part of this procedure. The data collection and training were executed separately.	28

3.8	Stage two method of training. The solid line shows the data collection part, and the dotted line shows the training part of this procedure. The data collection and training were executed separately.	31
4.1	Schematic of 2 Degree of freedom robot, which was used for the simulation.	34
4.2	Output of the neural network (solid line) and the PD controller (dashed line) at the completion of stage one training	38
4.3	Trajectory tracking performance at the beginning of stage two training. Desired trajectory for joint 1 (solid line) and measured trajectory for joint 1 (dashed line).	39
4.4	Trajectory tracking performance at the beginning of stage two training. Desired trajectory for joint 2 (solid line) and measured trajectory for joint 2 (dashed line).	40
4.5	Trajectory tracking performance at the beginning of stage two training. Tracking error of joint 1 (solid line) and tracking error of joint 2 (dashed line)	41
4.6	Desired and actual position of joint 1 at the completion of the second stage of training. Desired position is the solid line and actual position is the dashed line.	43
4.7	Desired and actual position of joint 2 at the completion of the second stage of training. Desired position is the solid line and actual position is the dashed line.	44
4.8	Position error of the manipulator. The solid line is the joint 1 tracking error and the dashed line is the joint 2 tracking error. The neural network has completed stage two training.	45

4.9	Torque outputs from the PD controller and the neural network for joint one. The solid line shows the PD controller output torque and the dashed line shows the Neural Network output torque. The neural network is at the end of the second stage of training.	47
4.10	Torque outputs from the PD controller and the neural network for joint two. The solid line shows the PD controller output torque and the dashed line shows the Neural Network output torque. The neural network is at the end of the second stage of training.	48
4.11	Comparison of the neural network output torque (solid line) and the exact torque required for perfect inverse dynamics control (dashed line) for joint one	49
4.12	Comparison of the neural network output torque (solid line) and the exact torque required for perfect inverse dynamics control (dashed line) for joint two	50
5.1	Schematic of 2 Degree of freedom robot, which was used for the experiment.	52
5.2	Connection between the robot and computer.	53
5.3	Comparison of the neural network output (solid line) and the PD controller output (dashed line) at the end of stage one training. The last 200 training iterations are shown.	57
5.4	Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.	58
5.5	Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.	59

5.6	Position error of the manipulator, where the Neural Network has completed the first stage of training for that trajectory. The solid line is joint one; the dashed line is joint two.	60
5.7	Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.	63
5.8	Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.	64
5.9	Position error of the manipulator, where the Neural Network has completed the second stage of training for that trajectory. The solid line is joint one the dashed line is joint two.	65
5.10	Comparison of the neural network and PD controller together (dashed line) and the PD controller alone (solid line). For the first joint. . . .	66
5.11	Comparison of the neural network and PD controller together (dashed line) and the PD controller alone (solid line). For the second joint. . .	67
5.12	Desired and actual position of the first joint of the manipulator. The dashed line is the desired position and the solid line is the actual position. The Neural Network was trained with a different trajectory.	68
5.13	Desired and actual position of the second joint of the manipulator. The dashed line is the desired position and the solid line is the actual position. The Neural Network was trained with a different trajectory.	69
5.14	Position error of the manipulator. The solid line is the error for joint 1 and the dashed line is the error for joint 2. The neural network was trained with a different trajectory.	70
5.15	Comparison of the neural network output (solid line) and the PD controller output (dashed line) at the end of stage one training. The last 200 training iterations are shown.	72

5.16	Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory. The neural network completed the second stage of training.	75
5.17	Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory. The neural network completed the second stage of training.	76
5.18	Position error of the manipulator, where the Neural Network has completed the first stage of training for that trajectory. The solid line is joint one the dashed line is joint two.	77
5.19	Desired and actual position of the manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for a different trajectory.	78
5.20	Desired and actual position of the manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for a different trajectory.	79
5.21	Position error of the manipulator, where the Neural Network has completed second stage of training for a different trajectory. The solid line is joint 1 tracking error and the dashed line is the joint 2 tracking error.	80
5.22	Comparison of the neural network and PD controller together and the PD controller alone. The solid line is joint 1 tracking error using PD controller alone and the dashed line is joint 1 tracking error by using the PD controller and Neural Network together	81

5.23 Comparison of the neural network and PD controller together and the PD controller alone. The solid line is joint 2 tracking error using PD controller alone and dashed line is joint 2 tracking error by using the PD controller and Neural Network together	82
---	----

Chapter 1

INTRODUCTION

This thesis investigates the use of neural network based controllers for use in robot trajectory control. Several recent publications have proposed that neural networks be used for general control applications [9,11,21]. Furthermore, several papers have proposed the use of these neuro-controllers for the robotics application [7,10,13-15,23]. This thesis examines the issues involved in implementing such neural network based controllers.

1.1 Trajectory Control

Several control methods have been proposed for controlling a robotic manipulator. Each control method has an advantage and a disadvantage with respect to the other methods. The first controller design is the Proportional Derivative Control (PD Controller) method [20]. The PD controller is one of the most simple feedback control designs. The advantage of this method is the simplicity of the design. This is good enough for most industrial applications, but for a high speed and high bandwidth manipulator, Inverse Dynamic and Computed Torque methods have been proposed. Inverse Dynamic Control and the Computed Torque method of control are two other well known methods of robot control [20]. In these two methods, the

dynamic structure and inertial parameters of the manipulator have to be known. The performance of these methods degrades due to modelling errors. To overcome the difficulties associated with modelling errors and parameter uncertainties, Adaptive Control Methods have been suggested by several researchers such as Craig et al [5] and Slotine and Li[18-19]. In these methods, the dynamic structure of the manipulator has to be known. However, the inertial parameters are estimated. These methods allow the manipulator controller to adapt to changes in the environment or the manipulator's inertial parameters. The proposed algorithm by Craig et al [5] requires the actual manipulator joint acceleration during the manipulator motion. This acceleration should either be measured or calculated. Due to noise in the system, the acceleration measurement is not precise. Another difficulty with this method is that the mass matrix has to be inverted. In addition, if the desired trajectory is not persistently excited, problems in estimation of the parameters could result [17]. The Slotine and Li algorithm [18] does not require the actual manipulator joint acceleration. A disadvantage of the Slotine and Li [18-19] algorithm is that the stability of the system in discrete time is not guaranteed [22]. Usually the manipulator is controlled by a computer, which is controlling the manipulator in discrete time, therefore the stability of the manipulator is not guaranteed.

In general, the above control systems for the robotic manipulator require some information about the dynamic structure of the manipulator. In addition, some of these algorithms require information about the inertial parameters of the manipulator.

1.2 Neural Network Control

Increasingly, Artificial Neural Networks are being used for solving problems such as pattern recognition or estimation of non-linear functions. Stinchcombe (1989) [21] illustrated the mathematical proof that a neural network, with at least one hidden layer, is capable of approximating a measurable function. Also, if a neural network is not able to perform the approximation, it is due to the following:

- Inadequate learning
- Insufficient numbers of hidden units
- The lack of deterministic relationship between input and target.

Miyamoto et al [10] proposed the idea of using an artificial neural network for trajectory control of a robotic manipulator. This method is called a Feedback Error Learning method. Narendra and Parthasarathy [11] showed that a neural network can be used in system identification.

Researchers have continued using artificial neural networks for controlling robotic manipulators. For example, Khemaissia and Morris [7] used a neural network in addition to a PD controller. They implemented a separate neural network for each manipulator joint, and the neural networks learned the required actuator torques along the desired trajectory. They used an on-line feedback error learning method [10] to train the network. Newton and Xu [13] proposed a neural network based controller for a space manipulator application. They too used separate neural networks for each joint of the manipulator and an on-line feedback error learning method to train the network. Newton and Xu also discussed the difficulties in choosing appropriate trajectories, learning strategies, and network architectures such that on-line training will be successful in teaching the manipulator inverse dynamics. Ozaki et. al. [14] used two Neural Networks in addition to a PD controller. They used the computed torque method for trajectory control of the manipulator. In this procedure, one neural network estimated the inertial matrix, and the other one estimated the centrifugal, coriolis and friction forces. In this design, the output of each neural network estimated a term in the inertial matrix or a term estimating centrifugal, coriolis, and frictional effects. For training the neural network, the dynamics equations of the manipulator had to be known. Wilhelmsen and Cotter [23] used different artificial neural networks to control a one degree of freedom manipulator. They illustrated that the Backpropagation network is a good choice.

Some researchers have suggested *The Cerebellar Model Articulation Controller* (CMAC) for controlling the manipulator. This method of manipulator control was

proposed by J.S. Albus [1]. This method is a reference table method. The input to the neural network depends on the state of the manipulator. The table shows which input node will be activated at which instant, and the value of that node will be input to the neural network. Miller et. al.[9] shows that the CMAC method can be an alternative for the backpropagation method, and it can be used in any design where a backpropagation network is used.

1.3 Trajectory Control of Robotic Manipulators By Using a Feedback-Error-Learning Neural Network

The objective of this thesis is to develop a neural network based controller for a robot manipulator. The idea is to use a single neural network to learn the inverse dynamics of the manipulator. The output of the neural network would then be used as a feedforward controller for the manipulator. It is assumed that the inertial parameters and the equation of motion for the manipulator are unknown and this information is not available for training the neural network. A two step feedback error learning method is implemented. This approach is originally and well presented in Miyamoto et. al. [10]. In the first stage of training, the neural network learns the output of the PD controller. In the second stage of training, the PD controller and the neural network operate in parallel. The neural network is then further trained by the output of the combined controller. Stage two training is repeated until the neural network has achieved the desired accuracy. Ideally, if one could drive the PD component of the combined controller to zero, then the feedforward signal, as given by the neural network, would be exactly the inverse dynamics of the manipulator. Using this two step method, the neural network learns the inverse dynamics of the manipulator.

1.4 Thesis Outline

In chapter 2, an artificial neural network has been presented. In addition, the back-propagation method of training a multi-layer neural network has been described in detail. Chapter 3 presents the structure of the neural network controller and the methods of training the network. Methods of choosing appropriate training trajectories and difficulties with on-line training are also discussed. Chapter 4 presents simulation results of the proposed method for a two degree of freedom serial link manipulator. Chapter 5 shows experimental results performed on a two degree of freedom direct drive manipulator. Appendix A contains the procedure for digitizing a prefilter. In addition, the code which is used in matlab for calculating the required matrix has been presented. The software listing of the simulation and experiment have been presented in appendices B and C respectively.

Chapter 2

ARTIFICIAL NEURAL NETWORKS

In this section, the basic idea of Neural Networks will be introduced. First, the basic concept of a neuron (or perceptron) will be introduced. Second, some widely used network designs will be shown, and finally one of the most widely used methods of training multilayer neural networks, Backpropagation (or delta rule), will be described in detail.

2.1 Neural Networks Basic Ideas

Human bodies contain millions and millions of neuron cells, and these cells are connected to each other. The signals from sensory perception pass back and forth through these cells to the brain. Figure 1 shows the physical structure of these cells. This is the basic idea behind the Artificial Neural Network. The idea is to model neurons, combine them by connection, and then pass a signal between them. Figure 2.2 shows the structure of a basic artificial neuron (Perceptron)[6].

Each perceptron has several input signals. These signals can be from another perceptron or from sensors, and each signal is multiplied by a constant number called a weight. These weights are measures of connections of synapses. If the weight is positive, the connection is EXCITATORY; and if the connection is negative, the connection is INHIBITORY. After all the signals are multiplied by the connection weights, the results are summed together. This summation will be used in the Threshold Procedure or as input into a nonlinear function such as a sigmoidal

Figure 2.1: physical structure of Neuron cells.

function. The result will be the activation of the perceptron (See Figure 2.2).

The perceptron shown in figure 2.2 is the basis of Neural Network architecture. By using the output of one or several perceptrons as the input to the next perceptron, a two level neural network will be designed. In general, by combining several perceptrons or neurons, different neural network designs can be created. Choosing a specific neural network architecture depends on the application for the neural network.

2.2 Multilayer Neural Networks

By using the output of one or several perceptrons as input to the next perceptron, two levels of neural networks will be designed. If the input to the perceptron layer is from sensors (from outside of the network), the layer is called the *Input layer*. If the output of the perceptron layer is not input to another layer in the network, the layer is called the *Output layer*. The layers between the input layer and output layers are called *Hidden layers*.

Figure 2.2: Structure of basic artificial neuron cells (Perceptron).

Multilayer neural networks consist of at least three layers: the input layer, the hidden layer, and the output layer. However, the hidden layer can contain several layers.

In Figure 2.3, a multilayer neural network has been shown. In this architecture, there are input and output layers with two additional hidden layers, and the output of each layer is the input for the next layer. This structure can be shown as a block diagram, where "w" consists of the weight matrices for that layer, and "r" is the nonlinear function—or the threshold function (See Figure 2.4).

A multilayer neural network can be modified by adding a *bias* term to one or all layers. The bias term for each layer can be counted as a perceptron which always has an activation of one. To calculate the activation of the second layer nodes, the procedure is as described previously: all the inputs will be multiplied by their corresponding weights, they will be summed together, and a bias term added. This result will be used in a nonlinear function. Figure 2.5 shows the presence of a bias term in the multilayer neural network architecture.

From a system theoretic point of view, multilayer neural networks can be

Figure 2.3: Structure of multilayer neural network.

Figure 2.4: Block diagram of multilayer network.

Figure 2.5: Multilayer neural network with bias term.

considered as versatile nonlinear maps with the elements of the weight matrices as parameters (Narendra, 1990)[11]

The multilayer neural network can be used in system identification[9]. A multilayer network is considered to be a system with the weight matrix as a parameter (the same as the parameters of the system which has to be identified). Adjusting the weight matrix will allow the system to perform exactly the same as the system which have to be identified. Therefore, the neural network weights are an estimation of system's parameters which are identified [9]. In addition, the multilayer neural network is a nonlinear system, which is well suited for the identification of nonlinear systems with unknown characteristics.

Moreover, the multilayer network has been successfully used for pattern recognition, and it is well documented in the literature since a large amount of research is currently being performed in this area [6,8].

There are also two practical uses of this network. First, the network is used for paint quality inspections in the automobile industry. A laser beam is sent to the paint surface, the reflected beam from the surface comes back and is checked, and the shape of the beam will be determined by the goodness of the surface. The

multilayer neural network is used to identify whether the reflected beam is from a good or bad surface. The second usage is data compression. This means it is used to reduce the needed data for transmitting a visual pattern from one station to another station.[6]

2.3 Neural Networks Training Algorithm

One of the common methods of training multilayer neural networks is Backpropagation (or Delta rule). The procedure for finding the best possible value for the weights of a neural network is called the training or learning method.

The literature indicates that a multilayer neural network is a good choice for trajectory control of the manipulator[7,10,11,13-15,23]. Static Backpropagation has been used for training the multilayer neural network in this project, and it will be described in detail.

The Backpropagation or Delta rule is a method for updating weights of a Multilayer Neural Network. The network is a simple network in which all of the nodes are connected. It consists of at least three layers which are input, hidden, and output layers; but the hidden layer can contain several layers. The name Backpropagation is used because there is a two way flow: one way is from the input layer to the output layer (this is used to obtain the results), and another way is from the output layer to the input layer (this is used to update weights- learning algorithm).

To train the network using the Backpropagation method, the input pattern is propagated to the output layer, and the result is checked with the expected result. From this, the error is calculated and used to update the weights. The methods of updating the output layer's weights and the hidden layer's weights are different.

To activate each perceptron on this network, two functions can be used. The first function is called the linear output unit and is a summation of all the inputs multiplied by the weights for that perceptron. The second function is a sigmoid (logistic) function which is

$$f(x) = \frac{1}{1 + \exp^{-x}} \quad (2.1)$$

To activate each perceptron on this network by using the sigmoidal function, x will be a summation of all the inputs multiplied by the weights for that perceptron.

$$x = \sum_{i=1}^n w_i X_i + \theta, i = 1, 2, \dots, n \quad (2.2)$$

where w_i is the weight of each input to that perceptron, X_i is the input to that perceptron, θ is a bias term, and n is the number of inputs to that perceptron.

In this section, the mathematical procedure for updating the weights of the output layer and the hidden layer of the network are presented. In addition, it is assumed that the network has been using the sigmoidal function for defining its activation. Figure 2.6 shows one example of multilayer network with three nodes in the input layer, two nodes in the hidden layer and one node in the output layer.

In the backpropagation learning method, the total error will be allocated to all of the nodes in all of the layers, depending on their contribution toward output. There is a function $y = \Phi(x), x \in R^n, y \in R^m$ and a set of P vector pairs of data $(x_1, y_1), (x_2, y_2), \dots, (x_p, y_p)$. One trains the network to estimate the function $\Phi(x)$ through the input/output data. To accomplish this task, the steepest descent technique will be used.

If the input vector $X_p = (X_{p1}, X_{p2}, \dots, X_{pn})^t$ has been used, the net input to the j^{th} node in the first hidden layer is

$$net_{pj}^h = \sum_{i=1}^n w_{ji}^h X_{pi} + \theta_j^h \quad (2.3)$$

where w_{ji}^h is the weight of each i^{th} input to the j^{th} perceptron in the hidden layer, X_{pi} is the input to the i^{th} node from the p^{th} set of input vectors to the perceptrons in the hidden layer, n is the number of nodes in the input layer, and θ_j^h is a bias term. For the network defined in figure 2.6, the input to the nonlinear function γ on the node one in the hidden layer (i.e could be sigmoidal) is,

$$net_{p1}^h = W_{11}^h X_{p1} + W_{12}^h X_{p2} + W_{13}^h X_{p3} \quad (2.4)$$

Figure 2.6: Simple example of multilayer network.

The output of j^{th} hidden layer node is

$$i_{pj} = f(net_{pj}^h) \quad (2.5)$$

Where h refers to the hidden layer, and the function f is a sigmoidal function.

The total input to the k^{th} node of output layer will be

$$net_{pk}^o = \sum_{j=1}^m w_{kj}^o i_{pj} + \theta_k^o \quad (2.6)$$

Where m is the number of nodes in hidden layer. For the network defined in figure 2.6, the input to the nonlinear function γ on the node one in the output layer (i.e could be sigmoidal) is,

$$net_{p1}^o = W_{11}^o i_{p1} + W_{12}^o i_{p2} \quad (2.7)$$

The output of the k^{th} node in the output layer will be

$$O_{pk} = f(net_{pk}^o) \quad (2.8)$$

(o refers to the output layers) The total error will be the addition of the error of each node in the output layer. If the error on each output layer node is

$$\delta_{pk} = (y_{pk} - O_{pk}) \quad (2.9)$$

where p is referred to as the p^{th} training vector, k is referred to as k^{th} node of the output layer, O_{pk} is the actual output at the node, and y_{pk} is the desired output at the node, then the total error will be

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 \quad (2.10)$$

Where r is the number of nodes in the output layer. To find the direction in which the weight has to change, the gradient of E_p with respect to the weight has to be calculated.

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 = \frac{1}{2} \sum_{k=1}^r (y_{pk} - O_{pk})^2 \quad (2.11)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(net_{pk}^o))^2 \quad (2.12)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(\sum_{j=1}^m w_{kj}^o i_{pj} + \theta_k^o))^2 \quad (2.13)$$

The partial derivative of this function with respect to the weights can be calculated using the chain rule from calculus, and the partial derivative will be:

$$-\frac{\partial E_p}{\partial w_{kj}^o} = (y_{pk} - O_{pk}) f'_k(\text{net}_{pk}^o) i_{pj} \quad (2.14)$$

Therefore, to update the weights on the output layer, this formula can be used:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - O_{pk}) f'_k(\text{net}_{pk}^o) i_{pj} \quad (2.15)$$

The factor η is called a learning rate parameter. The activation function is assumed to be a sigmoidal function, and the derivative of a sigmoidal function is:

$$f' = f(1 - f) \quad (2.16)$$

where f is a sigmoidal function. By using this assumption, the update of the output layer weight will be:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta(y_{pk} - O_{pk}) O_{pk} (1 - O_{pk}) i_{pj} \quad (2.17)$$

To reduce the size of the equation, a new variable has been introduced:

$$\delta_{pk}^o = (y_{pk} - O_{pk}) f_k(\text{net}_{pk}^o) \quad (2.18)$$

$$\delta_{pk}^o = \delta_{pk} f_k(\text{net}_{pk}^o) \quad (2.19)$$

The resulting weight update for the output layer will be:

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \eta \delta_{pk}^o i_{pj} \quad (2.20)$$

To update the weights of the hidden layer, the error of the hidden layer output and the gradient of total error with respect to each hidden layer weight has to be calculated.

$$E_p = \frac{1}{2} \sum_{k=1}^r \delta_{pk}^2 = \frac{1}{2} \sum_{k=1}^r (y_{pk} - O_{pk})^2 \quad (2.21)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(net_{pk}^o))^2 \quad (2.22)$$

$$E_p = \frac{1}{2} \sum_{k=1}^r (y_{pk} - f(\sum_{j=1}^m w_{kj}^o i_{pj} + \theta_k^o))^2 \quad (2.23)$$

in addition

$$i_{pj} = f(net_{pj}^h) = f(\sum_{i=1}^n w_{ji}^h X_{pi} + \theta_j^h) \quad (2.24)$$

By taking the partial derivative of E_p with respect to the hidden layer weights and substituting the equivalent partial derivative of different terms, the gradient will be:

$$-\frac{\partial E_p}{\partial w_{ji}^h} = -\sum_{k=1}^r (y_{pk} - O_{pk}) f_k(net_{pk}^o) w_{kj}^o f_j(net_{pj}^h) X_{pi} \quad (2.25)$$

The portion for updating the hidden layer weights will be

$$\Delta P w_{ji}^h = \sum_{k=1}^r (y_{pk} - O_{pk}) f_k(net_{pk}^o) w_{kj}^o \eta f_j(net_{pj}^h) X_{pi} \quad (2.26)$$

$$\Delta P w_{ji}^h = \eta f_j(net_{pj}^h) X_{pi} \sum_{k=1}^r \delta_{pk}^o w_{kj}^o \quad (2.27)$$

The hidden layer error term is defined as,

$$\delta_{pj}^h = f_j(net_{pj}^h) \sum_{k=1}^r \delta_{pk}^o w_{kj}^o \quad (2.28)$$

The weights on the hidden layer are updated by equation 2.29 as,

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \eta \delta_{pj}^h X_{pi} \quad (2.29)$$

One can use a similar approach to update the weights for the bias terms[6]. In this thesis the network used will not have a bias term. After all of the weights in the hidden layers and the output layer are updated, a new training vector will be propagated again, and the weights will be updated according to the new error. This procedure should be continued until the error is acceptable.

Chapter 3

THE NEURAL NETWORK CONTROLLER DESIGN

In this chapter, the structure of the neural network controller will be presented. In addition, the two stage method of training neural networks will be discussed. Methods of choosing appropriate training trajectories and difficulties with online training are presented.

3.1 The Neural Network Controller Structure

The general equation of motion of the n -link robotic manipulator is [20]

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) = \mathbf{T} \quad (3.1)$$

where $\mathbf{q} \in \mathbb{R}^n$ denotes a vector of joint positions, $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{n \times n}$ denotes the manipulator mass matrix, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \in \mathbb{R}^n$ denotes the vector of centrifugal and coriolis terms, $\mathbf{G}(\mathbf{q}) \in \mathbb{R}^n$ denotes the vector of gravity terms, and $\mathbf{T} \in \mathbb{R}^n$ denotes the input of each actuator. Ideally, if one could compute a control signal given by

$$\mathbf{T} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}_d + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) \quad (3.2)$$

then the manipulator would follow the desired trajectory perfectly. The objective is to estimate the function on the right hand side of equation (3.2). Where $\ddot{\mathbf{q}}_d$ denotes a vector of desired joint accelerations of the manipulator. Define the function,

$$\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d) = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}_d + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) \quad (3.3)$$

Figure 3.1: The controller model for each manipulator joint.

The control law is given as,

$$\mathbf{T} = \hat{\mathbf{g}}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d) + \mathbf{K}_p(\mathbf{q}_d - \mathbf{q}) + \mathbf{K}_d(\dot{\mathbf{q}}_d - \dot{\mathbf{q}}) \quad (3.4)$$

where $\hat{\mathbf{g}}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ is the estimate of the function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$. A multi-layer neural network (as described in chapter 2) has the capability of estimating a nonlinear function. The objective is to train the neural network to estimate the smooth function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ over the operating region of the robot manipulator.

$$\mathbf{NN}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d) = \hat{\mathbf{g}}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d) \quad (3.5)$$

This neural network is located in the feedforward path, with the PD controller in the feedback path to control the manipulator. Then the control law is given by,

$$\mathbf{T} = \mathbf{NN}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d) + \mathbf{K}_p(\mathbf{q}_d - \mathbf{q}) + \mathbf{K}_d(\dot{\mathbf{q}}_d - \dot{\mathbf{q}}) \quad (3.6)$$

where $\mathbf{NN}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ is the output of the neural network, $\mathbf{K}_p$ is the proportional gain, and $\mathbf{K}_d$ is the derivative gain. Figure 3.1 shows a block diagram of this method of control. The inputs to the neural network are desired position, desired velocity, desired acceleration, measured position, and measured velocity of each joint of manipulator. The outputs from the neural network are the actuator torques and forces.

3.2 Selection Of The Training Trajectory.

The neural network requires some information about the function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ for training. To collect this kind of information, the manipulator has to be in motion.

The trajectory chosen to train the neural network must have sufficient information such that the function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ can be estimated over the operating region of the robot. Therefore, a wide set of combinations of joint positions, velocities, desired accelerations, and input torques must be available.

If one used a single repetitive trajectory to train the neural network, then the neural network would not learn the inverse dynamics of the manipulator, but only the necessary torques and forces to move along that particular trajectory.

Figure 3.2: Nonlinear function of two variable.

The function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ is a function of several variables and produces a surface. To train the neural network to estimate this function, the information all along the surface is needed. Figure 3.2 shows one arbitrary surface to demonstrate the importance of selecting a trajectory. If a single repetitive trajectory is used to train the neural network, it will be like collecting information about one particular line in the surface shown in figure 3.2, for training the neural network.

As such, the trajectories chosen to train the neural network are generated by a set of stochastically generated inputs and time intervals. This method is similar to that proposed in Newton and Xu [13]. Commanded joint positions are generated by a random uniform selection over the operating range of each joint. The commanded joint positions are then driven through a prefilter whose bandwidth is set at the desired operating bandwidth of the manipulator. The output of the prefilter is the desired position, desired velocity, and the desired acceleration of each joint. Furthermore, the time interval for which each commanded joint position is held is also a uniform random variable. The time interval for which each commanded input is held is selected from a uniform random distribution as either 1, 2, 3, or 4 times the prefilter time constant. As such, relatively wide fluctuations in position, velocity, acceleration, and torques can be experienced by the manipulator.

By choosing randomly the commanded joint positions and the time intervals for which each commanded input is held, the system is excited by a wide combination of joint variables.

3.3 Sequential Training

The PD controller output is used as an initial estimate of inverse dynamics. Therefore, the output of the PD controller can be used for the initial training of the neural network. One method for training the neural network to map the PD controller is online (or Sequential) training. Figure 3.3 shows the sequential method for training the neural network. In the method of online training, the neural network is trained as the manipulator is in motion and under only PD control.

At each time step, the information which is used to control the manipulator is

Figure 3.3: Online methods of training (Sequential).

input to the neural network, and the output of the neural network is compared with the output of the PD controller. This information is used to update the neural network weights.

Initially, an attempt was made to train the neural network online, and the neural network was trained on each consecutive time step. Figure 3.3 shows this method of training. The neural network is trained as the manipulator is in motion and under only PD control.

For each time step that the manipulator was in the motion, desired position, desired velocity, desired acceleration, measured position, and measured velocity for each of the two joints were input to the neural network. Then the output of the neural network was compared with the output of the PD controller. Using this error and the Backpropagation algorithm, the weights of the neural network were updated. In this method, the neural network was trained to learn the output of the PD controller. The detailed description of the manipulator and the neural network used for the simulation has been presented in chapter 4. However, the result of the

simulation for the online training has been shown in this section. This simulation illustrates the difficulties one may encounter when the neural network is trained online. Figure 3.4 shows the output of the PD controller and the neural network, but the neural network is being trained as the manipulator follows the test trajectory. The results of figure 3.4 appear to be very good. The neural network output closely follows the PD controller output, and it appears that the neural network learned to map the PD controller. However, closer examination shows that the weights have not converged to a steady state value. They are oscillating as depicted in figure 3.5. A comparison of figure 3.4 and figure 3.5 shows that the neural network output layer weights were oscillating in phase with the desired output. This could have resulted because the bandwidth of the adaptation of the weights was higher than the bandwidth of the system's output. However, this has not been proven. The online training was stopped, and the neural network was tested. The neural network output could no longer follow the output of the PD controller. This effect is depicted in figure 3.6.

3.4 Off line training

As mentioned above, using online training was not successful in training the neural network to map the PD controller. To train the neural network a two stage off-line training is proposed.

3.4.1 Stage One training: Learning The PD Controller Output

In stage one of the feedback-error-learning method, the neural network is trained to learn the output of the PD controller. The PD controller output is used as an initial estimate of the inverse dynamics of the manipulator.

Figure 3.7 shows the first stage of the training process. In this stage, the manipulator is controlled by only the PD controller. The manipulator is moved in the robot work space in a path generated by a set of stochastically generated inputs and

Figure 3.4: PD controller torque (solid line) and neural network output (dashed line) during online training.

Figure 3.5: First weight in First node of output layer during online training.

Figure 3.6: Torque input (solid line) and neural network output (dashed line) when online training has been turned off. The network was trained online.

time intervals. This robot motion produces a wide range of joint variable combinations. This method of trajectory selection has been discussed in section 3.2. At each time step, desired position, desired velocity, desired acceleration, measured position, measured velocity, and joint torques and forces for each actuator are stored to an array indexed by time step i . At completion of the training trajectory, the array is written to disc.

To train the neural network to map a function such as $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$, it is required that all the variables in the function $\mathbf{g}(\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}_d)$ be input to the neural network. The PD controller is a function of desired position, desired velocity, measured position, and measured velocity. Therefore, desired position, desired velocity, measured position, and measured velocity are required inputs to the neural network. Desired acceleration is one of the variables in the manipulator inverse dynamics function. Therefore, desired acceleration should be one of the inputs to the neural network.

The neural network structure used in this thesis cannot be trained by processing data sequentially in time, one cannot process in sequential order data points $i = 1, i = 2, \dots, i = N$. This would be the same as online training (discussed in section 3.3) To obviate the difficulties of online (or sequential) training, the training set of data is selected randomly. A uniform random number generator selects a time step from $i = 1$ to $i = N$. Where N is the number of collected sets of data points during the motion of the manipulator under the control of the PD controller. The data collected at this randomly selected time step is used to train the neural network. As such, any correlation between subsequent training points is lost, and the weights no longer oscillate.

The inputs to the neural network are desired position, desired velocity, desired acceleration, measured position, and measured velocity for each joint. When the random time step is chosen, the data for that time step are input to the neural network. The outputs of the neural network are compared with the PD controller output torques for that time step. The error between the PD controller output torque and the output of the neural network are used to update the weights on the neural network by equation 2.20 and 2.29. This procedure is continued until the neural network output is close to the PD output torques. Figure 3.7 illustrates the

Figure 3.7: First stage of training. The solid line shows the data collection part, and the dotted line shows the training part of this procedure. The data collection and training were executed separately.

stage one phase of training.

3.4.2 Stage Two Training: Learning The Inverse Dynamics

In stage one training, the neural network is trained to map the PD controller. Therefore, the neural network output is an initial estimate for the required input actuator torques, and this neural network can be used as the inverse dynamics part of the control design.

Stage two training begins by implementing a inverse dynamics controller (Neural Network) in combination with a PD controller. The inverse dynamics component of the controller, as depicted in figure 3.8, is the output of the neural network. The neural network at the beginning of stage two training is the neural network which had been trained in stage one.

The manipulator is controlled by the neural network in the inverse dynamics path and the PD controller in feedback path. The manipulator is moved again in the robot workspace as a filtered step function, in a random fashion, and with a random time interval. This robot motion produces a wide range of joint variable combinations. This method of trajectory selection has been discussed in section 3.2.

At each time step, desired position, desired velocity, desired acceleration, measured position, measured velocity, and the joint torques and forces for each actuator are stored to an array indexed by time step i . At this stage, the actuator torque is the addition of the PD controller output and Neural Network output. When the manipulator completes the motion on the random trajectory, then the array is saved to disc.

A uniform random number generator selects a time step from $i = 1$ to $i = N$. The data collected at this randomly selected time step is used to train the neural network. The inputs to the neural network are the desired position, the desired velocity, the desired acceleration, the measured position, and the measured velocity for each joint. The outputs are the actuator input torques or forces

The input torques and forces to the actuator in this stage are the addition of the

neural network output and PD controller output torques.

$$\tau_{in} = \tau_{pd} + \tau_n \quad (3.7)$$

Where the τ_{in} is the actuator input torques and forces, τ_{pd} is the PD controller output, and τ_n is the neural network output which is used in the feed forward path during the manipulator motion.

The error used to train the neural network is the difference between the output of the neural network and the actuator forces and torques that were stored as the manipulator followed the test trajectory. This error is given by

$$e_\tau = \tau_{in} - \tau_{nn} \quad (3.8)$$

where τ_{in} is the actuator torques and forces, and τ_{nn} is the output of the neural network which is in training. The network is trained until the weights converge to a steady state value. At the end of this initial phase of stage two training, the neural network is the best available estimate of the inverse dynamics.

The training is then repeated as follows:

- a) Select a desired trajectory as described in section 3.2.
- b) Implement a inverse dynamics controller in which the feedforward signal is the output of the most recently trained neural network.
- c) Collect data as described in section 3.4.2
- d) Retrain the neural network used in step b) as described above in section 3.4.2.
- e) Repeat steps a) - d) until the weights reach a steady state value or the desired accuracy is achieved.

Figure 3.8: Stage two method of training. The solid line shows the data collection part, and the dotted line shows the training part of this procedure. The data collection and training were executed separately.

Chapter 4

SIMULATION

To illustrate the neural network control design, a robot with two degrees of freedom is simulated. This chapter will begin by presenting the manipulator and its specifications. This manipulator design was used to test for on-line training, and the result of this simulation has been presented in section 3.3. The simulation results of the first and second stages of off-line training have been presented separately. After completing the second stage of training, the actual inverse dynamics function of the manipulator was compared with the neural network to verify the accuracy of the neural network. This result has been presented at the end of the second stage training section.

4.1 Robot Model

The manipulator model which was used for the simulation is a 2 Degree of Freedom Robot. This robot is made of two revolute joints. The masses for each link are assumed to be a concentrated mass at the end of each link. Figure 4.1 shows the manipulator used.

The position of the end effector with respect to the base coordinate system for the different joint variables is as follows:

$$P_x = l_1 \cos q_1 + l_2 \cos(q_2 + q_1 - \pi) \quad (4.1)$$

$$P_y = l_1 \sin q_1 + l_2 \sin(q_2 + q_1 - \pi) \quad (4.2)$$

Where P_x and P_y are the cartesian coordinates of the end effector; q_1 and q_2 are the angular positions of the joints; and l_1 and l_2 are link lengths. The coordinate systems, joint variables, and link lengths have been shown in figure 4.1.

The general equation of motion of the n -link robotic manipulator is

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{G}(\mathbf{q}) = \mathbf{T} \quad (4.3)$$

where $\mathbf{q} \in \mathbb{R}^n$ denotes a vector of joint positions, $\mathbf{M}(\mathbf{q}) \in \mathbb{R}^{n \times n}$ denotes the manipulator mass matrix, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \in \mathbb{R}^n$ denotes the vector of centrifugal and coriolis terms, $\mathbf{G}(\mathbf{q}) \in \mathbb{R}^n$ denotes the vector of gravity terms, and $\mathbf{T} \in \mathbb{R}^n$ denotes the input of each actuator. In this simulation, the effect of gravity was ignored, therefore the equation of the motion of the manipulator is as follows:

$$\mathbf{T} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \quad (4.4)$$

where,

$$\dot{\mathbf{M}}(\mathbf{q}) = \begin{bmatrix} m_1 l^2 + 2m_2 l^2 + 2m_2 l^2 \cos q_2 & m_2 l^2 + m_2 l^2 \cos q_2 \\ m_2 l^2 + m_2 l^2 \cos q_2 & m_2 l^2 \end{bmatrix} \quad (4.5)$$

$$\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) = \begin{bmatrix} -2m_2 l^2 \dot{q}_1 \dot{q}_2 \sin q_2 - m_2 l^2 \dot{q}_2^2 \sin q_2 \\ m_2 l^2 \dot{q}_1^2 \sin q_2 \end{bmatrix} \quad (4.6)$$

In these equations, m_i corresponds to the mass of each link which is concentrated, l_i corresponds to the lengths of each link, and q corresponds to the joint variable. The chosen values of each parameter for this simulation have been shown in table 4.1.

The acceleration of the manipulator joints can be computed as,

$$\ddot{\mathbf{q}} = \mathbf{M}^{-1}(\mathbf{q})\mathbf{T} - \mathbf{M}^{-1}(\mathbf{q})\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) - \mathbf{M}^{-1}(\mathbf{q})\mathbf{G}(\mathbf{q}) \quad (4.7)$$

Equation 4.7 is numerically solved (simulated) by using a fourth order Runge-Kutta algorithm.

Figure 4.1: Schematic of 2 Degree of freedom robot, which was used for the simulation.

<i>Lengths (meters)</i>	<i>Link masses(kg)</i>
$l_1 = 1$	$m_1 = 1$
$l_2 = 1$	$m_2 = 2$

Table 4.1: Link Parameters of the 2 DOF Manipulator.

4.2 Mapping The PD Controller

According to the design that was described in chapter 3, in stage one of the feedback-error-learning method, the neural network is trained to learn the output of the PD controller. The PD controller output is an initial estimate of the inverse dynamics of the manipulator.

The neural network consisted of two hidden layers with 35 nodes in each layer (10 nodes in the input layer and 2 nodes in the output layer)(10,35,35,2). The literature shows that in mapping a function, such as the inverse dynamic of the manipulator, two hidden layers is a good choice [6,11,13,14]. Due to memory limitation and speed of calculation of the algorithm, 35 nodes were chosen for the hidden layer. The inputs to the neural network are the desired position, the desired velocity, the desired acceleration, the measured position, and the measured velocity for each of the two joints. The two outputs of the neural network are the inverse dynamics torque inputs for each joint.

The desired position, the desired velocity, and the desired acceleration are given as the output of a second order prefilter. The prefilter is specified as a critically damped second order system with a bandwidth of $\omega_n = 2.0$ rad/sec. The prefilter transfer function is given as,

$$W_m(s) = \frac{4}{s^2 + 4s + 4} \quad (4.8)$$

In state space form one can specify the prefilter as,

$$\begin{bmatrix} \dot{q}_{mi} \\ \ddot{q}_{mi} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -4 & -4 \end{bmatrix} \begin{bmatrix} q_{mi} \\ \dot{q}_{mi} \end{bmatrix} + \begin{bmatrix} 0 \\ 4 \end{bmatrix} q_{ci} \quad (4.9)$$

This prefilter was digitized for the sampling frequency of 100 Hz which in discrete form is as follows:

$$\begin{bmatrix} q_{mi(k+1)} \\ \dot{q}_{mi(k+1)} \end{bmatrix} = \begin{bmatrix} 0.9998 & 0.0098 \\ -0.0392 & 0.9606 \end{bmatrix} \begin{bmatrix} q_{mi(k)} \\ \dot{q}_{mi(k)} \end{bmatrix} + \begin{bmatrix} 0.0002 \\ 0.0392 \end{bmatrix} q_{ci} \quad (4.10)$$

The derivation of equation 4.10 from equation 4.9 is given in appendix B. The step response of this model was used to generate the trajectory for the manipulator.

Commanded joint positions were generated by a random uniform selection over the operating range of each joint. The commanded joint positions were then driven through the prefilter defined in equation 4.8 and 4.10. Furthermore, the time interval for which each commanded joint position was held is also a uniform random variable and was selected from a uniform random distribution as either 1, 2, 3, or 4 times the prefilter time constant. This generation of trajectories was continued for 10 second intervals.

The robot was controlled by a PD controller and moved on the above trajectory. The data collected consisted of desired position, desired velocity, desired acceleration, measured position and measured velocity for the two joints, and torque input to each joint. At each time step, the above mentioned data were stored to an array indexed by time step i . The simulation sampling rate was set at 100 Hz. The result was 1000 test points for training the neural network.

The neural network consisted of two hidden layers with 35 nodes in each layer (10 nodes in the input layer and 2 nodes in the output layer). The non-linear function used for this neural network was a sigmoidal function. To train the neural network, backpropagation was used.

A neural network cannot be trained by processing the data sequentially in time. One cannot process in sequential order data points $i = 1, i = 2, \dots, i = N$. This would be the same as online training (as discussed in section 3.3). To obviate the difficulties of online or sequential training, the data was processed randomly. A uniform random number generator was used to select a time step from $i = 1$ to $i = 1000$. The data collected at this randomly selected time step was used to train the neural network. As such, any correlation between subsequent training points was lost, and the weights no longer oscillated. When the time step was chosen, then the corresponding desired position, desired velocity, desired acceleration, measured position, and measured velocity of each joint were input to the neural network. The stored torque for each joint during data collection in this stage (stage one of training) was $\tau = \tau_{pd}$, where τ is the actuator torques and forces, and τ_{pd} is the output of the PD controller during data collection.

The error between the PD controller output torque and the output of the neural

network was used to update the weights on the neural network. This training error is given by

$$e_\tau = \tau - \tau_{nn} \quad (4.11)$$

where τ is the actuator torques and forces, τ_{nn} is the output of the neural network during training, and e_τ is the training error. The Backpropagation method was used to update the neural network weights. The learning rate, η , was set at 0.2 at the begining of training, then after some iterations it was changed to 0.1 and then to 0.05. The above training was continued for 1000 iterations. The above procedure was continued for 481 randomly generated trajectories. Therefore, the neural network was trained for 481,000 training iteration points. Figure 4.2 illustrates the output of the neural network in comparison to the PD controller at the completion of stage one training. The neural network closely matches the PD controller output. These data are from the last trajectory which was used for training.

To test the performance of this design at the end of stage one training, the neural network was added to the system. The neural network provided the inverse dynamics signal, as depicted in figure 3.1, and the PD controller added in the feedback signal.

A new trajectory with a duration of 26 seconds was used to test the system. Figures 4.3 and 4.4 show the performance of the neural network controller design at the end of stage one. Figures 4.3 and 4.4 are the desired position and measured position of the manipulator joints and figure 4.5 shows the position errors of both joints during the motion of the manipulator.

4.3 Mapping The Inverse Dynamics

According to the design described in chapter 3, in stage two of the feedback-error-learning method, the neural network was trained to learn the inverse dynamics of the manipulator.

Stage two training began by implementing a inverse dynamics controller in combination with the PD controller. The inverse dynamics component of the controller, as depicted in figure 3.8, was the output of the neural network. The neural net-

Figure 4.2: Output of the neural network (solid line) and the PD controller (dashed line) at the completion of stage one training

Figure 4.3: Trajectory tracking performance at the beginning of stage two training. Desired trajectory for joint 1 (solid line) and measured trajectory for joint 1 (dashed line).

Figure 4.4: Trajectory tracking performance at the beginning of stage two training. Desired trajectory for joint 2 (solid line) and measured trajectory for joint 2 (dashed line).

Figure 4.5: Trajectory tracking performance at the beginning of stage two training. Tracking error of joint 1 (solid line) and tracking error of joint 2 (dashed line)

work at the beginning of stage two training was the neural network which had been trained in stage one

To begin this part of the simulation, the commanded joint positions were generated by a random uniform selection over the operating range of each joint. The commanded joint positions were then driven through a prefilter with a bandwidth set at the desired operating bandwidth of the manipulator ($\omega_n = 2.0$ rad/sec). Furthermore, a ten second test trajectory was selected.(see section 3.2)

The robot was controlled by the neural network in the inverse dynamics path together with the PD controller in the feedback path and moved on the above trajectory. The data collected consisted of the desired position, the desired velocity, the desired acceleration, the measured position, the measured velocity and the torque input for each joint. At each time step, the above mentioned data were stored to an array indexed by time step i . The simulation sampling rate was set at 100 Hz. The result was 1000 test points for training the neural network. A uniform random number generator was used to select a time step from $i = 1$ to $i = 1000$. The data collected at this randomly selected time step was used to train the neural network. When the time step was chosen, then the corresponding desired position, desired velocity, desired acceleration, measured position, and measured velocity of each joint were input to the neural network. The error used to train the network was the difference between the output of the neural network as it is being trained and the actuator forces and torques that were stored as the manipulator followed the test trajectory. This torque was the addition of the neural network output and the PD controller output during the time that the manipulator followed the test trajectory. Therefore, the stored torque for each joint during data collection was

$$\tau = \tau_{pd} + \tau_n \quad (4.12)$$

where τ is the actuator torques and forces, τ_n is the output of the neural network during data collection, and τ_{pd} is the output of the PD controller during data collection. This training error is given by

$$e_\tau = \tau - \tau_{nn} \quad (4.13)$$

Figure 4.6: Desired and actual position of joint 1 at the completion of the second stage of training. Desired position is the solid line and actual position is the dashed line.

The above training continued for 1000 iterations, and then a new trajectory was generated. The above procedure was continued for 531 randomly generated trajectories. Therefore, the neural network was trained for 531,000 training iteration points. A new trajectory with a duration of 27.5 seconds was used to test the system. Figures 4.6, 4.7, and 4.8 show the performance of the neural network controller design at the end of stage two. Figures 4.6 and 4.7 are the desired position and measured position of the first and second joints of the manipulator. The error is too small to be visible on this scale. Figure 4.8, shows the trajectory tracking error, it has a maximum value of approximately 0.003 rad.

Figure 4.7: Desired and actual position of joint 2 at the completion of the second stage of training. Desired position is the solid line and actual position is the dashed line.

Figure 4.8: Position error of the manipulator. The solid line is the joint 1 tracking error and the dashed line is the joint 2 tracking error. The neural network has completed stage two training.

Figures 4.9 and 4.10 show the neural network output signal and the PD controller signal. One can see that the magnitude of the neural network output signal was much larger than the PD controller signal. This implies that the neural network was a good estimator of the inverse dynamics of the manipulator. Figures 4.11 and 4.12 show both the neural network output signal and the exact signal as given by the inverse dynamics (equation 4.3). Here one can see that the neural network was a good estimator of the inverse dynamics.

The trajectory simulated for the results shown in figures 4.6 and 4.7 was different from the trajectory on which the neural network was trained. Since the the neural network was a good estimator of the inverse dynamics, the neural network could be used as a inverse dynamics controller for controlling the manipulator on any arbitrary trajectory.

Figure 4.9: Torque outputs from the PD controller and the neural network for joint one. The solid line shows the PD controller output torque and the dashed line shows the Neural Network output torque. The neural network is at the end of the second stage of training.

Figure 4.10: Torque outputs from the PD controller and the neural network for joint two. The solid line shows the PD controller output torque and the dashed line shows the Neural Network output torque. The neural network is at the end of the second stage of training.

Figure 4.11: Comparison of the neural network output torque (solid line) and the exact torque required for perfect inverse dynamics control (dashed line) for joint one

Figure 4.12: Comparison of the neural network output torque (solid line) and the exact torque required for perfect inverse dynamics control (dashed line) for joint two

Chapter 5

EXPERIMENT

To verify the Neural Network controller design, the controller has been implemented on the Carleton University Direct Drive Manipulator. The results of several experiments will be presented.

5.1 Manipulator Description

The Carleton University Direct Drive Manipulator is a two Degree of Freedom (DOF) *Selective Compliance Assembly Robot Arm* (SCARA) type which operates in the horizontal plane. The Direct Drive robot is defined as a manipulator with no reducer on its actuator. Therefore, the robot joint will move the same amount as the actuator connected to that joint.

An advantage of using a direct drive robot is that it ensures high speed, accurate tracking of fast trajectories, while keeping the problem of backlash and friction to a minimum, without compromising high stiffness (this can be achieved by using a good controller).

A disadvantage of using direct drive robots is that they are very sensitive to the effects of an external load upon them. The residual forces must be rejected as a disturbance by a PD controller. Therefore, during the operation of the direct drive robot, the PD controller has to be used to reduce any unwanted movement of the robot.

The direct drive manipulator, which has been used in this project, is a direct drive

Figure 5.1: Schematic of 2 Degree of freedom robot, which was used for the experiment.

robot made at Carleton University. This manipulator uses a five bar linkage. This manipulator has two direct drive brushless DC servo motors as actuators. These motors are made by Motion Control Systems (MCS) Inc. The two motors each have a twelve bit digital resolver to sense the position, and a tachometer to measure the velocity. Figure 5.1 shows the schematic of the Carleton University direct drive robotic manipulator which is used in this experiment.

The computer is interfaced to the robot motors and servo amplifiers via interface cards. Figure 5.2 shows a diagram of the connection between the computer and the direct drive robot. The two main actuators of the direct drive robot, which are the two direct drive DC servo motors, each consist of a twelve bit digital resolver / encoder to sense the position of the robot which sends a digital signal to the computer. These resolvers / encoders are connected to the Digital / Digital data translation card in the computer. In addition, the DC servo motor's tachometer measures the velocity of the motor (joint) and will send an analog signal to the computer. These analog signals will pass through an analog / digital data translation card in the computer. The digital control signal will pass through a digital / analog data translation card in the computer, and the analog control voltages will pass to

Figure 5.2: Connection between the robot and computer.

the servo amplifiers.

These Analog / Digital and Digital / Analog cards are manufactured by Data Translation Inc and have eight differential input Analog / Digital channels. Six of these input channels are used by a force sensor; and the other two, by the tachometers.

The Digital / Analog data translation card has eight channels. Two of these channels are used by the computer to control the DC servo motors. The other two channels are used to control the gripper's finger motor and the motor which is used to move the gripper in a vertical motion. For this project, the gripper and linear lead screw extension of the manipulator was disconnected.

A current will be sent to the two direct drive brushless DC servo motors by two

independent MCS servo amplifiers. These amplifiers are powered by two MCS high voltage power supplies. The commutation of the motors is performed electronically by the servo amplifiers using the resolver position signals.

The computer is a 486 PC computer with a clock speed of 33 MHz. This computer was used for control and data acquisition.

5.2 Mapping The PD Controller

According to the design that was described in chapter 3, in stage one of the feedback-error-learning method, the neural network is trained to learn the output of the PD controller. The PD controller output is an initial estimate of the inverse dynamics of the manipulator.

To begin the experiment, the trajectory has to be defined for the manipulator. The procedure for choosing the trajectory has been discussed in detail in chapter 3. To generate the trajectory which is described in chapter 3, a third order filter with a bandwidth of $\omega_n = 5.0$ rad/sec was used. The prefilter has three poles at -5, and its transfer function is given as,

$$W_m(s) = \frac{125}{s^3 + 15s^2 + 75s + 125} \quad (5.1)$$

In state space form one can specify the model as,

$$\begin{bmatrix} \dot{q}_{mi} \\ \ddot{q}_{mi} \\ q_{mi}^{(3)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -125 & -75 & -15 \end{bmatrix} \begin{bmatrix} q_{mi} \\ \dot{q}_{mi} \\ \ddot{q}_{mi} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 125 \end{bmatrix} q_{ci} \quad (5.2)$$

This prefilter was digitized for the sampling time of 50 Hz which in discrete form is as follows:

$$\begin{bmatrix} q_{mi(k+1)} \\ \dot{q}_{mi(k+1)} \\ \ddot{q}_{mi(k+1)} \end{bmatrix} = \begin{bmatrix} 0.9998 & 0.0199 & 0.0002 \\ -0.0226 & 0.9863 & 0.0172 \\ -2.1490 & -1.3120 & 0.7284 \end{bmatrix} \begin{bmatrix} q_{mi(k)} \\ \dot{q}_{mi(k)} \\ \ddot{q}_{mi(k)} \end{bmatrix} + \begin{bmatrix} 0.0002 \\ 0.0226 \\ 2.1490 \end{bmatrix} q_{ck} \quad (5.3)$$

Commanded joint positions were generated by a random uniform selection over the operating range of each joint. The commanded joint positions were then driven

through a prefilter with a bandwidth set at the desired operating bandwidth of the manipulator ($\omega_n = 5.0$ rad/sec). Furthermore, the time interval for which each commanded joint position was held is also a uniform random variable and was selected from a uniform random distribution as either 1, 2, 3 or 4 times the prefilter time constant. This generation of trajectories was continued for 15 second intervals and was saved as the input file for the motion of the manipulator. Thirty input files (as described above) were generated.

The robot was moved on the above saved trajectory. While the robot was in motion, a PD Controller was used and data collected. The data consisted of desired position, desired velocity, desired acceleration, measured position, measured velocity and torque input for each joint (voltage input to the motor). At each time step, the above mentioned data were stored to an array indexed by time step i . At completion of the training trajectory, the array was written to a disc. The computer sampling rate was set at 50 Hz. The result was 30 files which consisted of 750 test points for each trajectory and a total of 22500 test points for training the neural network. See figure 3.3 and the first stage of training (learning the PD controller) section for a description of this procedure.

The neural network consisted of two hidden layers with 35 nodes in each layer (10 nodes in the input layer and 2 nodes in the output layer were used) (10,35,35,2). The literature shows that in mapping a function, such as the inverse dynamic of the manipulator, two hidden layers is a good choice [6,11,13,14]. Due to memory limitation and speed of calculation of the algorithm, 35 nodes were chosen for the hidden layer. The non-linear function used for this neural network was a sigmoidal function. To train the neural network, backpropagation was used.

Initially, each of the 30 data files are selected one at a time and used to train the neural network. Subsequently, a uniform random number generator selects one of the thirty data files. Another uniform random number generator selects a time step from $i = 1$ to $i = 750$. The data collected at this randomly selected time step is used to train the neural network. Each time that a file is loaded to the program, one thousand training iterations are performed. Then a new data file is chosen randomly, and the procedure is continued.

The inputs to the neural network were the desired position, the desired velocity, the desired acceleration, the measured position, and the measured velocity for each joint. These data were input to the neural network from the time step that had been chosen randomly. The outputs of the neural network were compared with the torque input to the actuator which, in this stage, were the PD controller output voltages for that time step. The learning rate, η , was set at 0.2 at the beginning of training, then after some iterations it was changed to 0.1 and then to 0.05.

The stored voltage for each joint during the data collection in this stage (stage one of training) were $\tau = \tau_{pd}$, where τ is the actuator inputs, and τ_{pd} is the output of the PD controller during the data collection. The error between the PD controller output voltage and the output of the neural network is used to update the weights on the neural network. This training error is given by

$$\epsilon_{\tau} = \tau - \tau_{nn} \quad (5.4)$$

where τ is the actuator inputs, τ_{nn} is the output of the neural network during the training, and ϵ_{τ} is the training error.

This procedure is continued until the neural network output is close to the PD output voltages. The neural network underwent 492,000 training iterations. Figure 5.3 illustrates the results of this stage one training. The last 200 training iterations are depicted. It was felt that the accuracy of the neural network was sufficient for continuation to stage two training.

After the first stage of training, the trained network was added to the PD Controller to control the manipulator. Figures 5.4 and 5.5 show the measured position and desired position for joints one and two of the manipulator when using the neural network as a inverse dynamics controller at the end of stage one training. The position error for both joints of the manipulator have been plotted and shown in figure 5.6.

Figure 5.3: Comparison of the neural network output (solid line) and the PD controller output (dashed line) at the end of stage one training. The last 200 training iterations are shown.

Figure 5.4: Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.

Figure 5.5: Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.

Figure 5.6: Position error of the manipulator, where the Neural Network has completed the first stage of training for that trajectory. The solid line is joint one; the dashed line is joint two.

5.3 Mapping The Inverse Dynamics

According to the design described in chapter 3, in stage two of the feedback-error-learning method, the neural network is trained to learn the inverse dynamics of the manipulator.

Stage two training begins by implementing a inverse dynamics controller in combination with the PD controller. The inverse dynamics component of the controller, as depicted in figure 3.8, is the output of the neural network. The neural network at the beginning of stage two training is the neural network which had been trained in stage one. Since this neural network was trained to map the PD controller, it should perform in a way that is similar to a PD controller. However, the addition of a PD controller in this design will reduce the error.

To begin the second stage of the experiment, new sets of trajectories have been defined for the manipulator. The procedure for choosing the trajectories has been discussed in chapter 3. To generate the trajectories described in chapter 3, a third order filter with a bandwidth of $\omega_n = 5.0$ rad/sec was used. This prefilter was described in section 5.1. The same procedure was used to generate 30 new trajectory files which each contained 15 second motions for the manipulator.

The manipulator was moved along the 30 training trajectories using a combination of the neural network as a inverse dynamics controller and the PD controller in the feedback path. The data consisted of desired position, desired velocity, desired acceleration, measured position, measured velocity, and voltage input for each joint (voltage input to the motor) which were collected for each trajectory. At this time, the voltage input to each joint was the addition of neural network output and the PD controller output. At each time step, the above mentioned data were stored to an array indexed by time step i . At completion of the training trajectory, the array is written to disc, and the computer sampling rate is set at 50 Hz. Therefore, there were 30 files which consisted of 750 test points for each trajectory and a total of 22500 test points for training the neural network. See figure 3.8 and the second stage training section for a description of this procedure.

To ensure that all of the data files were loaded to the program, at the beginning

of training, all of the data files were selected one by one. Subsequently, the data files were selected randomly. This was exactly the same procedure which was used for the first stage of training.

The inputs to the neural network are the desired position, the desired velocity, the desired acceleration, the measured position, and the measured velocity for each joint. These data were input to the neural network from the time step that was chosen randomly. The error used to train the network was the difference between the output of the neural network during training and the actuator voltage that were stored as the manipulator followed the test trajectory. This voltage was the addition of the neural network output and the PD controller output during the time that the manipulator followed the test trajectory. Therefore, the stored voltage for each joint during data collection was

$$\tau = \tau_{pd} + \tau_n \quad (5.5)$$

where τ is the actuator input voltages, τ_n is the output of the neural network during data collection, as the manipulator follows the desired trajectory under neural network control and τ_{pd} is the output of the PD controller during data collection. This training error is given by

$$e_\tau = \tau - \tau_{nn} \quad (5.6)$$

where τ is the actuator torques and forces, τ_{nn} is the output of the neural network during training, and e_τ is the training error.

The network was trained until the weights converged to a steady state value. At the end of this initial phase of stage two training, the neural network was the best available estimate of the inverse dynamics. The neural network underwent 862,000 training iterations.

Figures 5.7 and 5.8 shows the desired trajectory and the actual position of the manipulator by using the neural network as a inverse dynamics controller at the end of stage two training. The last 15 seconds of the trajectory is depicted in the figures. The desired trajectory is the test trajectory on which the neural network training was done. The position error was plotted and shown in figure 5.9. Figures 5.10 and 5.11 compare the performance of the PD controller alone to the combination of

Figure 5.7: Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.

neural network controller and PD control for joint one and two.

An arbitrary test trajectory is then selected. This is a trajectory for which the neural network had never been trained. The combined neural network controller and PD controller is used to control the robot. Figure 5.12 and 5.13 show the desired position and the actual position of the robot for this test. Figure 5.14 is a plot of position error. This shows that the robot can move along any desired trajectory with a maximum of 0.8 and 0.5 degree error for joint one and joint two respectively.

Figure 5.8: Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory.

Figure 5.9: Position error of the manipulator, where the Neural Network has completed the second stage of training for that trajectory. The solid line is joint one the dashed line is joint two.

Figure 5.10: Comparison of the neural network and PD controller together (dashed line) and the PD controller alone (solid line). For the first joint.

Figure 5.11: Comparison of the neural network and PD controller together (dashed line) and the PD controller alone (solid line). For the second joint.

Figure 5.12: Desired and actual position of the first joint of the manipulator. The dashed line is the desired position and the solid line is the actual position. The Neural Network was trained with a different trajectory.

Figure 5.13: Desired and actual position of the second joint of the manipulator. The dashed line is the desired position and the solid line is the actual position. The Neural Network was trained with a different trajectory.

Figure 5.14: Position error of the manipulator. The solid line is the error for joint 1 and the dashed line is the error for joint 2. The neural network was trained with a different trajectory.

5.4 Trained for the Predefined Trajectory

In addition to the procedure that was described in the simulation and experiment sections, the neural network controller design was tested by using a predefined trajectory. In the previous section, the trajectory was randomly selected. In this section, the trajectory was a combination of sinusoidal functions. To begin this experiment, the robot was moved on a predefined sinusoidal trajectory. The trajectory equations of each joint are,

$$Pos1 = 205 - (25\cos(2\pi ft) + 25\cos(2\pi(2f)t)) \quad (5.7)$$

$$Pos2 = 90 - (20\cos(2\pi(0.5f)t) + 20\cos(2\pi(2.5f)t)) \quad (5.8)$$

Where $Pos1$ is the position of joint one, $Pos2$ is the position of joint two, and $f = 0.4Hz$ is the frequency. While the robot was in motion, a PD Controller was used and the data collected. See figure 3.7 and section 3.4.1 for a description of this procedure. By using the method described in section 5.2, the neural network was trained to map the PD controller. The last 200 training iterations have been depicted and presented in figure 5.15.

After training the neural network, it was added to the PD Controller to control the manipulator. For further training of the neural network, the manipulator was moved in the same trajectory (given by equations 5.7 and 5.8) as in initial training. Figures 5.16 and 5.17 show the desired position and actual position of the manipulator joints by using the neural network after completion of the second stage of training. The position error has been plotted and shown in figure 5.18.

This network was then used to move the robot on a different trajectory from the one which was used to train the neural network. Figures 5.19 and 5.20 show the desired and actual position of the robot for this test. Figure 5.21 is a plot of position error. In this experiment the maximum position error was approximately 4 degrees and 1 degree for joints one and two respectively. Figures 5.22 and 5.23 compare the performance of the PD controller alone to the combination of neural network controller and PD control for joints one and two.

Figure 5.15: Comparison of the neural network output (solid line) and the PD controller output (dashed line) at the end of stage one training. The last 200 training iterations are shown.

5.5 Discussion

In section 5.2 and 5.3, the training trajectories were chosen randomly. The trained neural network was used as a inverse dynamics controller combined with a PD controller in the feedback path to control the manipulator. An arbitrary test trajectory was selected. The neural network had never been trained for this trajectory. The manipulator was moved on this trajectory, and the results of this experiment have been shown in figures 5.12, 5.13, and 5.14. Figure 5.14 shows that the robot can move along the arbitrarily selected desired trajectory with maximum errors of 0.8 and 0.5 degrees for joints 1 and 2, respectively.

In section 5.4, only one predefined test trajectory was used. Figures 5.16 to 5.21 show the performance of this neural network when the manipulator moved on the test trajectory and on another predefined trajectory. Figures 5.22 and 5.23 show the performance of the second neural network in comparison to the PD controller. The results indicate that the performance of the neural network controller combined with the PD controller is much better than the performance of the PD controller alone. Figure 5.18 shows that the robot can move along the training trajectory with maximum tracking errors of 2.5 and 0.8 degrees for joints 1 and 2, respectively. However, when the robot moved along the trajectory that it was not trained for, the tracking errors were 4 and 1.4 degrees for joints 1 and 2, respectively (see figure 5.21).

A comparison of figures 5.14 and 5.21 indicates that the neural network which was trained by using an arbitrary test trajectory performed better than the neural network trained by using only one test trajectory. The superior performance was shown when the manipulator moved along a trajectory other than the one it was trained for. Figures 5.19 and 5.21, where the neural network was trained for a specific trajectory, show that a jump in the manipulator's joint one of 60 degrees corresponds to a tracking error of 3.2 degrees. In addition, when the robot moved on the small sinusoidal portion of the trajectory, the tracking error was 3 degrees (see figure 5.21). However, figures 5.12 and 5.14, where the neural network was trained for sets of arbitrary trajectories, show that the manipulator joint one jumped 80

degrees when the tracking error was only 0.6 degrees.

In summary, the above results indicate that the neural network trained by using sets of arbitrary trajectories learns the inverse dynamics of the manipulator better than one trained by using only one predefined trajectory. Also, the neural network, trained by using one specific trajectory, is capable of learning the required torque for that trajectory with a higher degree of accuracy. However, the performance of this network is not as good when it is used to control the manipulator along another trajectory.

Figure 5.16: Desired and actual position of manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory. The neural network completed the second stage of training.

Figure 5.17: Desired and actual position of manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for this trajectory. The neural network completed the second stage of training.

Figure 5.18: Position error of the manipulator, where the Neural Network has completed the first stage of training for that trajectory. The solid line is joint one the dashed line is joint two.

Figure 5.19: Desired and actual position of the manipulator joint one. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for a different trajectory.

Figure 5.20: Desired and actual position of the manipulator joint two. The solid line is the desired position and the dotted line is the actual position. The Neural Network was trained for a different trajectory.

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Figure 5.21: Position error of the manipulator, where the Neural Network has completed second stage of training for a different trajectory. The solid line is joint 1 tracking error and the dashed line is the joint 2 tracking error.

Figure 5.22: Comparison of the neural network and PD controller together and the PD controller alone. The solid line is joint 1 tracking error using PD controller alone and the dashed line is joint 1 tracking error by using the PD controller and Neural Network together

Figure 5.23: Comparison of the neural network and PD controller together and the PD controller alone. The solid line is joint 2 tracking error using PD controller alone and dashed line is joint 2 tracking error by using the PD controller and Neural Network together

Chapter 6

CONCLUSION

This thesis presented a feedback error approach to neural network training for robot trajectory control. In this approach, one does not need to know the inertial parameters nor the form of the equation of dynamics for the manipulator. Using a two stage off-line training process, the neural network is capable of learning the inverse dynamics of the manipulator. The neural network is then used in a feedforward control algorithm. A randomized training trajectory is proposed as well as random training point selection and processing.

Chapter 4 shows the simulation results of this approach. These results indicate that the neural network learns the manipulator inverse dynamics. Figures 4.11 and 4.12 show that the output of the neural network is following the inverse dynamic manipulator function with maximum error of 1 N.m. Since the neural network learned the inverse dynamics of the manipulator, it can be used to control the manipulator on any trajectory within the manipulator workspace. In addition, figure 4.8 shows that the manipulator followed an arbitrary trajectory with a tracking error of plus or minus 0.03 degrees.

Experiments were performed on a two degree of freedom direct drive manipulator, and the experimental results supported the simulation results. The manipulator was followed an arbitrary trajectory with a tracking error of plus or minus 0.8 degrees for the first joint and plus or minus 0.6 degrees for the second joint.

The neural network successfully learned the inverse dynamics and was able to track an arbitrary trajectory with a high degree of accuracy.

In chapter 2, the on-line training of the standard backpropagation neural network for trajectory control of the manipulator was presented. As the simulation showed, this method was not successful. Since the neural network weights were oscillating, the neural network could not map the PD controller output

Bibliography

- [1] Albus, J.S. 'A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)', *Transactions of the ASME, Journal of Dynamic Systems, Measurement and Control*, Vol.98, September 1975, pp.220-227.
- [2] An, Chae H., C.G. Atkeson, and J.M. Hollerbach, *Model-Based Control of a Robot Manipulator*, The MIT Press, Massachusetts, 1988.
- [3] Asada, H., T. Kanade, and I. Takeyama, 'Control of a Direct-Drive Arm', *Transactions of the ASME, Journal of Dynamic Systems, Measurement and Control*, Vol.105, September 1983, pp.136-142.
- [4] Craig, J.J. *Adaptive Control Of Mechanical Manipulators*, Addison-Wesley, Reading, Massachusetts, 1986.
- [5] Craig, J.J., P. Hsu, and S. Sastry, 'Adaptive Control Of Mechanical Manipulators', *International Journal of Robotics research*, Vol.6, No.2, 1987, pp.16-28.
- [6] Freeman, James A. and David M. Skapura, *Neural Networks Algorithms, Applications, and Programming Techniques*, Addison-Wesley Pub: California. 1991.
- [7] Khemaissia, S. and A.S. Morris, 'Neuro-adaptive control of robotic manipulators', *Robotica* Vol 11, Part 5, 465-473 (1993).
- [8] Kosko, Bart, *Neural Networks and Fuzzy Systems A Dynamical Systems Approach to Machine Intelligence*. Prentice Hall: New Jersey. 1992.

- [9] Miller, W. T., F.H. Glanz, L.G. Kraft. 'CMAC: An Associative Neural Network Alternative to Backpropagation', *Proceedings of the IEEE*, Vol.78, No.10, October 1990, pp.1561- 1567.
- [10] Miyamoto, H., M. Kawato, T. Setoyama and R. Suzuki, 'Feedback-error-learning neural networks for trajectory control of a robotic manipulator', *Neural Networks* **1**, 251-265. (1988).
- [11] Narendra, K.S. and Kannan Parthasarathy, 'Identification and Control of Dynamical Systems Using Neural Networks', *IEEE Trans. Neural Networks* **1**, No. 1, 4-26 (1990).
- [12] Narendra, K.S. and A.M. Annaswamy, *Stable Adaptive Systems* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
- [13] Newton, R.T. and Y. Xu, 'Neural Network Control of a Space Manipulator', *IEEE Control Systems* **13**, No. 6, 14-22 (1993).
- [14] Ozaki, T. T. Suzuki, T. Furuhashi, S. Okuma and Y. Uchikawa 'Trajectory control of Robotic Manipulators Using Neural Networks' *IEEE Trans. on Industrial Electronics* **38** No. 3, 195-202 (June 1991).
- [15] Poo, A.N., M.H. Ang Jr, C.L. Teo, and Qing Li, 'Performance of a Neuro-Model-Based Robot Controller: Adaptability and Noise Rejection', *Intelligent Systems Engineering* Vol.1, No.1, 50-62 (1992).
- [16] Rajguru, S., *Analysis and Design of a Direct Drive Robot*, Master's Thesis, Carleton University, Ottawa, Ontario, 1989.
- [17] Schwartz, H.M., G. Warshaw, and T. Janabi, 'Issues in Robot Adaptive Control', *Proceedings of the American Control Conference*, San Diego, California, May 1990, pp. 2797-2805.
- [18] Slotine, J-J.E. and W. Li, 'On the Adaptive Control of Robot Manipulators', *International Journal of Robotics Research*, Vol.6, No.3, Fall 1987, pp.49-59.

- [19] Slotine, J.-J.E. and W. Li, 'Adaptive Manipulator Control: A Case Study', *IEEE Transactions on Automatic Control* Vol.33, No.11, November 1988, pp.995-1003.
- [20] Spong, M.W. and M. Vidyasagar, *Robot Dynamics and Control*, John Wiley and Sons, New York, 1989.
- [21] Stinchcombe, M. and H. White, 'Multilayer Feedforward Networks are Universal Approximators', *Neural Networks*, Vol.2, pp. 359-366. 1989.
- [22] Warshaw, G.D., *Sampled-Data Robot Adaptive Control*, PHD Thesis, Carleton University, Ottawa, Ontario, January 1994.
- [23] Wilhelmsen, K. and N. Cotter, 'Neural Network Based Controllers for a Single-Degree-of-Freedom Robotic Arm', *Int. J. Conf. Neural Net.*, 1990.

Appendix A

Prefilter

In this section the method of digitizing the prefilter has been presented. This is an IRR filter which can be found in literature. In this section, the method of finding each matrix has been presented. Then the commands which can be used in *matlab* to calculate these matrices have been shown.

A.1 Prefilter Matrices

The transfer function of second or higher order systems can be written in state space form. For example the transfer function of a second order system is as follows:

$$Y(s) = \frac{\omega_n^2}{s^2 + \omega_n \eta s + \omega_n^2} X(s) \quad (\text{A.1})$$

Then this transfer function can be written in state space form as

$$\begin{bmatrix} \dot{y}_{mi} \\ \ddot{y}_{mi} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -\omega_n \eta \end{bmatrix} \begin{bmatrix} y_{mi} \\ \dot{y}_{mi} \end{bmatrix} + \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix} x_{ci} \quad (\text{A.2})$$

In state space the matrices called A and B respectively are $A = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -\omega_n \eta \end{bmatrix}$ and $B = \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix}$. To digitize this transfer function, it can be written in the form

$$\begin{bmatrix} y_{k+1} \\ \dot{y}_{k+1} \end{bmatrix} = \begin{bmatrix} a_{d11} & a_{d12} \\ a_{d21} & a_{d22} \end{bmatrix} \begin{bmatrix} y_k \\ \dot{y}_k \end{bmatrix} + \begin{bmatrix} b_{d1} \\ b_{d2} \end{bmatrix} x_k \quad (\text{A.3})$$

The matrix A_d is simple, and it is equal to $\exp^{At}$ where A is the matrix in the transfer function in state space form, and t is the sampling time. In addition, the matrix B_d is equal to

$$B_d = B \int \exp^{A(t-\tau)} d\tau \quad (\text{A.4})$$

then

$$B_d = \exp^{At} \int_0^t \exp^{-A\tau} d\tau \quad (\text{A.5})$$

The $\exp^{At}$ can be written as

$$\exp^{At} = I + At + \frac{A^2 t^2}{2!} + \frac{A^3 t^3}{3!} + \dots \quad (\text{A.6})$$

and

$$\exp^{-A\tau} = I - A\tau + \frac{A^2\tau^2}{2!} - \frac{A^3\tau^3}{3!} + \dots \quad (\text{A.7})$$

$$\int_0^t \exp^{-A\tau} d\tau = \tau - \frac{A\tau^2}{2!} + \frac{A^2\tau^3}{3!} - \frac{A^3\tau^4}{4!} + \dots \quad (\text{A.8})$$

By comparing equation B.6 and B.8 then

$$\int_0^t \exp^{-A\tau} d\tau = A^{-1}(I - \exp^{-At}) \quad (\text{A.9})$$

Therefore the B_d can be calculated as

$$B_d = \exp^{At}(A^{-1}(I - \exp^{-At}))B \quad (\text{A.10})$$

The above equation shows how the matrices A_d and B_d can be calculated.

A.2 Matlab Code

Matlab can be used to calculate the matrices A_d and B_d . The code which was used to calculate the matrices for simulation has been presented.

```
a= [0 1: -4 -4]           {Identify Matrix A}
ae = expm(a*0.01)         {Calculatin Ad}
ain = inv(a)               {Inverse of matrix A}
aen = expm(a*(-0.01))
I = [1 0;0 1]             {Identity matrix}
c = I - aen
d = ain * c
e = ae * d
B = [ 0 ; 4 ]              {Identify Matrix B}
Bd = e * B
```

Appendix B

Simulation Software Listing

In this appendix the listing of the software which was used for the simulation has been shown. First the listing of the two stage of the training has been shown. In addition the program which was used to test the Neural Network has been shown.

B.1 First Stage Of Training

```

program sim1pd;
{*****}
* Name : Zaryab Hamavand      *
* Date : Nov 1993             *
*                               *
* This program is the first stage of *
* the training. The input torque to *
* each actuator is the PD controller *
* output.                       *
{*****}
{$N+}
{$R+} {Enable range checking}

uses CRT;

Const
    num_of_input    = 10;
    num_of_output    = 2;
    first_node_num   = 35;
    second_node_num  = 35;
    etha             = 0.2;

    A11              = 0.9998;
    A12              = 0.0098;
    A21              = -0.0392;
    A22              = 0.9606;

    B1               = 0.0002;
    B2               = 0.0392;

    st1              = 1000;

    dt               = 0.01;
    kd               = 35.0;
    kp               = 60.0;

    m112             = 1.0;
    m212             = 2.0;
    wtof             = -1;

    addt1            = 20.0;
    divt1            = 50.0;
    addt2            = 10.0;
    divt2            = 25.0;

Type
    sarr = array [1..4] of Double;
    inarr = array [1..2] of Double;

Var

```



```

f1,f2,f3,f4,f5,f6      : text;
y1k1,y1k2,rout         : sarr;
xinp,tin               : inarr;
ii,jj,kk               : Integer;
f                      : sarr;
intwei                 : Double;
m,l,ss,ds,count1       : Integer;
Count2,st2             : Integer;

first_hid_weight : array[1..num_of_input, 1..first_node_num] of Double;
sec_hid_weight  : array[1..first_node_num, 1..second_node_num] of Double;
output_weight   : array[1..second_node_num, 1..num_of_output] of Double;

first_hid_out    : array[1..first_node_num] of Double;
first_hid_err    : array[1..first_node_num] of Double;

sec_hid_out      : array[1..second_node_num] of Double;
sec_hid_err      : array[1..second_node_num] of Double;

output_out       : array[1..num_of_output] of Double;
output_err       : array[1..num_of_output] of Double;
target           : array[1..num_of_output] of Double;

input_lay        : array[1..num_of_input] of Double;
data_set         : array[1..500, 1..12] of double;

{*****
* This procedure will propagate the input vector through the *
* network and will produce output. This network consists of  *
* two hidden layers.                                         *
*****}

procedure prop_forward;

Var
  i, j : Integer;
  sum : Double;

Begin
  For i := 1 To first_node_num Do
    Begin
      sum := 0.0;
      For j := 1 to num_of_input Do
        Begin
          sum := sum + (first_hid_weight[j,i] * input_lay[j]);
        End;
      first_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
    End;

    For i := 1 To second_node_num Do
      Begin
        sum := 0.0;
        For j := 1 to first_node_num Do
          Begin
            sum := sum + (first_hid_out[j] * sec_hid_weight[j,i]);
          End;
        sec_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
      End;
    End;
  End;

```

```

End;

For i := 1 To num_of_output Do
Begin
    sum := 0.0;
    For j := 1 to second_node_num Do
    Begin
        sum := sum + (sec_hid_out[j] * output_weight[j,i]);
    End;
    output_out[i] := 1.0 / (1.0 + exp(-1 * sum));
End;

End;

{*****
* This procedure will calculate the error on each layer's nodes, *
* according to the error on the output. *
*****}

Procedure compute_errors;

Var
    i, j : Integer;

Begin
    For i := 1 to num_of_output do
    Begin
        output_err[i] := output_out[i] * (1 - output_out[i]) *
            (target[i] - output_out[i]);
    End;

    For i := 1 to second_node_num do
    Begin
        sec_hid_err[i] := 0.0;
        For j := 1 to num_of_output do
        Begin
            sec_hid_err[i] := sec_hid_err[i] + ( output_err[j] *
                output_weight[i,j]);
        End;

        sec_hid_err[i] := sec_hid_err[i] * sec_hid_out[i] *
            (1 - sec_hid_out[i]);
    End;

    For i := 1 to first_node_num do
    Begin
        first_hid_err[i] := 0.0;
        For j := 1 to second_node_num do
        Begin
            first_hid_err[i] := first_hid_err[i] + ( sec_hid_err[j] *
                sec_hid_weight[i,j]);
        End;

        first_hid_err[i] := first_hid_err[i] * first_hid_out[i] *
            (1 - first_hid_out[i]);
    End;

```

```

End;

End;

{*****
*This procedure will update the weights of each layer according to*
*the error on each node.                                           *
*The 1st hidden layer will be updated, then the 2nd hidden layer, *
*and finally the output layer.                                     *
*****}

Procedure update_weights;

Var
  i, j : Integer;

Begin
  For i := 1 to first_node_num Do
    Begin
      For j := 1 to num_of_input Do
        Begin
          first_hid_weight[j,i] := first_hid_weight[j,i] + ( etha *
                                                                first_hid_err[i] * input_lay[j]);
        End;
      End;
    End;

    For i := 1 to second_node_num Do
      Begin
        For j := 1 to first_node_num Do
          Begin
            sec_hid_weight[j,i] := sec_hid_weight[j,i] + ( etha *
                                                                sec_hid_err[i] * sec_hid_out[j]);
          End;
        End;
      End;

      For i := 1 to num_of_output Do
        Begin
          For j := 1 to second_node_num Do
            Begin
              output_weight[j,i] := output_weight[j,i] + ( etha *
                                                                output_err[i] * sec_hid_out[j]);
            End;
          End;
        End;
      End;

    End;

  End;

{*****
* This procedure will calculate the desired position and velocity*
* by using a second order filter (IIR filter)                      *
*****}

Procedure sfiltin(xin :inarr; yk1:sarr; var yk2:sarr);

```

Begin

```

    yk2[1] := (A11 * yk1[1]) + (A12 * yk1[2]) + (B1 * xin[1]);
    yk2[2] := (A21 * yk1[1]) + (A22 * yk1[2]) + (B2 * xin[1]);

    yk2[3] := (A11 * yk1[3]) + (A12 * yk1[4]) + (B1 * xin[2]);
    yk2[4] := (A21 * yk1[3]) + (A22 * yk1[4]) + (B2 * xin[2]);

```

End;

```

{*****
* This procedure will put the curent position and velocity in the *
* previous position and velocity                                     *
*****}

```

Procedure update(var yk1:sarr; yk2:sarr);

```

var
  i : Integer;

```

Begin

```

    For i := 1 to 4 do
    Begin
        yk1[i] := yk2[i];
    End;

```

End;

```

{*****
*This procedure will calculate the required torque by using      *
*proportional and derivative (PD) controller.                  *
*****}

```

Procedure calctorq(Var torq : inarr; yd, yd1, ya : sarr);

Begin

```

    torq[1] := (yd1[1] - ya[1]) * kp + (yd1[2] - ya[2]) * kd ;
    torq[2] := (yd1[3] - ya[3]) * kp + (yd1[4] - ya[4]) * kd ;

```

```

    data_set[ss,1] := ya[1];
    data_set[ss,2] := ya[2];
    data_set[ss,3] := yd1[1];
    data_set[ss,4] := yd1[2];

```

```

    data_set[ss,5] := ya[3];
    data_set[ss,6] := ya[4];
    data_set[ss,7] := yd1[3];
    data_set[ss,8] := yd1[4];

```

```

    data_set[ss,9] := (yd[2] - yd1[2]) / 1.005016708E-2 ;
    data_set[ss,10] := (yd[4] - yd1[4]) / 1.005016708E-2 ;

```

```

    data_set[ss,11] := ( torq[1] + addt1 ) / divt1;

```

```

    data_set[ss,12] := ( torq[2] + addt2 ) / divt2;
    ss := ss + 1;
End;

{*****
* This procedure will simulate the dynamic of the 2DOF robot. *
* In this procedure Neural Network will be trained to estimate *
* the dynamics of the robot *
*****}

Procedure compute_f(Var ya : sarr ; t1 : inarr);
Var
    cos1,sin1,cos3,sin3,a,b,c,d,adcb : Double;
Begin
    cos1 := cos(ya[1]);
    sin1 := sin(ya[1]);

    cos3 := cos(ya[3]);
    sin3 := sin(ya[3]);

    a := m112 + 2.0*m212 + 2.0*m212*cos3;
    b := m212 + m212*cos3;
    c := m212 + m212*cos3;
    d := m212;
    adcb := a*d - c*b;

    f[1] := ya[2];
    f[2] := (d/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
        + (b/adcb) * m212*ya[2]*ya[2]*sin3 + (d/adcb) * t1[1]
        - (b/adcb) * t1[2];

    f[3] := ya[4];
    f[4] := -(c/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
        - (a/adcb) * m212*ya[2]*ya[2]*sin3 - (c/adcb) * t1[1]
        + (a/adcb) * t1[2];

End;

Procedure runge_kutta(Var y:sarr; t1 : inarr);
{*****
* This is the numerical solution to the nonlinear dynamics *
* of the robot manipulator. This procedure calls another *
* procedure, which is compute_f *
* compute_f, computes the derivatives of the states in *
* vector ya. These derivatives are stored in vector f(y(t)). *
* The state of the manipulator is calculated based on the *
* following recursion; *
*  $y(t + dt) = y(t) + 1/6 (k_0 + 2k_1 + 2k_2 + k_3)$  *
* *
* where *
*  $k_0 = f(y(t)) * dt$  *
*****}

```

```

*      k1 = f(y(t) + 1/2 k0) * dt      *
*      k2 = f(y(t) + 1/2 k1) * dt      *
*      k3 = f(y(t) + k2) * dt          *
*                                     *
*   The function f(y(t)) has to be calculated four times *
*****}

Var
  i,j : Integer;
  k    : array[1..4,1..4] of Double;
  y1   : sarr;

Begin
  For i := 1 to 4 do
    Begin
      Compute_f(y,t1);

      {* If this is the first time through the loop present state
        of the manipulator will be stored and k0 and y(t) + 1/2k0
        which will be used as the argument for f(y(t) + 1/2k0) will
        be computed.

      *}

      If i = 1 then
        Begin
          for j := 1 to 4 do
            Begin
              y1[j] := y[j];
              k[i,j] := f[j] * dt;
              y[j] := y1[j] + 0.5 * k[i,j];
            End;
          End;

          If i = 2 then
            Begin
              for j := 1 to 4 do
                Begin
                  k[i,j] := f[j] * dt;
                  y[j] := y1[j] + 0.5 * k[i,j];
                End;
              End;

              If i = 3 then
                Begin
                  for j := 1 to 4 do
                    Begin
                      k[i,j] := f[j] * dt;
                      y[j] := y1[j] + 0.5 * k[i,j];
                    End;
                  End;

                  If i = 4 then
                    Begin
                      for j := 1 to 4 do
                        Begin

```

```

        k[i,j] := f[j] * dt;
        { * compute updated state of the robot * }
        y[j] := y1[j] + (1.0/6.0)*(k[1,j] + 2.0*k[2,j] +
            2.0*k[3,j] + k[4,j]);

        { * The robot state has been updated * }
    End;
End;
End;
End;

{=====
=                               Main Program!!!!                               =
=====}

Begin

    Randomize;
    Assign(f1,'a1arob1.out');
    Rewrite(f1);
    Assign(f2,'a1arob2.out');
    Rewrite(f2);
    Assign(f3,'a1arob3.out');
    Rewrite(f3);
    Assign(f4,'a1arob4.out');
    Rewrite(f4);
    Assign(f5,'a1arob5.out');
    Rewrite(f5);
    Assign(f6,'a1arob6.out');
    Rewrite(f6);

    For l := 1 to first_node_num do
    Begin
        For m := 1 to num_of_input do
        Begin
            first_hid_weight[m,l] := (Random - 0.5) * 2.0;
        End;
    End;
    For l := 1 to second_node_num do
    Begin
        For m := 1 to first_node_num do
        Begin
            sec_hid_weight[m,l] := (Random - 0.5) * 2.0;
        End;
    End;
    For l := 1 to num_of_output do
    Begin
        For m := 1 to second_node_num do
        Begin
            output_weight[m,l] := (Random - 0.5) * 2.0;
        End;
    End;
    For ii := 1 to 4 do
    Begin
        y1k1[ii] := 0.0;
        y1k2[ii] := 0.0;
    End;
    End;

```

```

    rout[ii] := 0.0;
End;

Count2 := 0;
While not KeyPressed do
Begin
    writeln(Count2);
    ss := 1;
    While ss <= st1 do
    Begin
        xinp[1] := Random;
        xinp[2] := Random;
        st2 := (Random(5) + 1) * 50;
        For jj := 1 to st2 do
        Begin
            While ss <= st1 do
            Begin
                sfiltin(xinp,y1k1,y1k2);
                calctorq(tin,y1k2,y1k1,rout);
                runge_kutta(rout,tin);
                update(y1k1,y1k2);
            End;
        End;
    End;
End;
For jj := 1 to 1000 do
Begin
    ds := Random(498) + 2 ;
    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
    input_lay[5] := data_set[ds,5];
    input_lay[6] := data_set[ds,6];
    input_lay[7] := data_set[ds,7];
    input_lay[8] := data_set[ds,8];
    input_lay[9] := data_set[ds,9];
    input_lay[10] := data_set[ds,10];
    target[1] := data_set[ds,11];
    target[2] := data_set[ds,12];
    prop_forward;
    compute_errors;
    update_weights;
End;
Count2 := Count2 + 1;
End;
count1 := 0;
ss := 1;
While ss <= st1 do
Begin
    xinp[1] := Random;
    xinp[2] := Random;
    st2 := (Random(5) + 1) * 50;
    For jj := 1 to st2 do
    Begin
        While ss <= st1 do
        Begin

```



```

        sfiltin(xinp,y1k1,y1k2);
        calctorq(tin,y1k2,y1k1,rout);
        runge_kutta(rout,tin);
        update(y1k1,y1k2);
        If ((count1 Mod 2) = 0 ) then
        Begin
            writeln(f1,y1k2[1],', ',rout[1],', ',
,y1k2[3],', ',rout[3]);
        End;
        count1 := count1 + 1;
    End;
End;
count1 := 0;
For ji := 1 to 1000 do
Begin
    ds := Random(498) + 2 ;
    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
    input_lay[5] := data_set[ds,5];
    input_lay[6] := data_set[ds,6];
    input_lay[7] := data_set[ds,7];
    input_lay[8] := data_set[ds,8];
    input_lay[9] := data_set[ds,9];
    input_lay[10] := data_set[ds,10];
    target[1] := data_set[ds,11];
    target[2] := data_set[ds,12];

    prop_forward;
    compute_errors;
    update_weights;

    count1 := count1 + 1;
    If ((count1 Mod 4) = 0 ) then
    Begin
        For l := 1 to num_of_output do
        Begin
            For m := 1 to 4 do
            Begin
                write(f5,output_weight[m,l],', ');
            End;
        End;
        writeln(f5);
        For l := 1 to 4 do
        Begin
            For m := 1 to 2 do
            Begin
                write(f3,first_hid_weight[m,l],', ');
            End;
        End;
        writeln(f3);
        For l := 1 to 4 do
        Begin
            For m := 1 to 2 do

```

```

        Begin
            write(f4,sec_hid_weight[m,1], ' ');
        End;
    End;
    writeln(f4);
    write(f6,target[1], ' ',output_out[1], ' ');
    write(f6,target[2], ' ',output_out[2], ' ');
    writeln(f6);
End;
End;
For l := 1 to first_node_num do
Begin
    For m := 1 to num_of_input do
    Begin
        write(f2,first_hid_weight[m,1], ' ');
    End;
    writeln(f2);
End;
For l := 1 to second_node_num do
Begin
    For m := 1 to first_node_num do
    Begin
        write(f2,sec_hid_weight[m,1], ' ');
    End;
    writeln(f2);
End;
For l := 1 to num_of_output do
Begin
    For m := 1 to second_node_num do
    Begin
        write(f2,output_weight[m,1], ' ');
    End;
    writeln(f2);
End;
Close(f1);
Close(f2);
Close(f3);
Close(f4);
Close(f5);
Close(f6);
end.

```

B.2 Second Stage Of The Training

```

program sim1pdnn;

{*****}
* Name : Zaryab Hamavand *
* Date : Nov 1993 *
* *
* This program is the second stage of training. *
{*****}

{$N+}
{$R+} {Enable range checking}

uses CRT;
Const
    num_of_input      = 10;
    num_of_output     = 2;
    first_node_num    = 35;
    second_node_num   = 35;
    etha              = 0.2;

    A11               = 0.9998;
    A12               = 0.0098;
    A21               = -0.0392;
    A22               = 0.9606;

    B1                = 0.0002;
    B2                = 0.0392;

    st1               = 1000;

    dt                = 0.01;
    kd                = 35.0;
    kp                = 60.0;

    m112              = 1.0;
    m212              = 2.0;
    wtof              = -1;

    addt1              = 20.0;
    divt1              = 50.0;
    addt2              = 10.0;
    divt2              = 25.0;

Type
    sarr = array [1..4] of Double;
    inarr = array [1..2] of Double;

Var
    f1,f2,f3,f4,f5,f6      : text;
    y1k1,y1k2,rout         : sarr;
    xinp,tin               : inarr;
    ii,jj,kk               : Integer;
    f                      : sarr;
    intwei                 : Double;
    m,l,ss,ds,count1       : Integer;
    Count2,st2             : Integer;

```

```

first_hid_weight : array[1..num_of_input, 1..first_node_num] of Double;
sec_hid_weight : array[1..first_node_num, 1..second_node_num] of Double;
output_weight : array[1..second_node_num, 1..num_of_output] of Double;

first_hid_out : array[1..first_node_num] of Double;
first_hid_err : array[1..first_node_num] of Double;

sec_hid_out : array[1..second_node_num] of Double;
sec_hid_err : array[1..second_node_num] of Double;

output_out : array[1..num_of_output] of Double;
output_err : array[1..num_of_output] of Double;
target : array[1..num_of_output] of Double;

input_lay : array[1..num_of_input] of Double;
data_set : array[1..500, 1..12] of double;

{*****
* This procedure will propagate the input vector through the *
* network and will produce output. This network consists of *
* two hidden layers. *
*****}

procedure prop_forward;

Var
  i, j : Integer;
  sum : Double;

Begin
  For i := 1 To first_node_num Do
    Begin
      sum := 0.0;
      For j := 1 to num_of_input Do
        Begin
          sum := sum + (first_hid_weight[j,i] * input_lay[j]);
        End;
      first_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
    End;

    For i := 1 To second_node_num Do
      Begin
        sum := 0.0;
        For j := 1 to first_node_num Do
          Begin
            sum := sum + (first_hid_out[j] * sec_hid_weight[j,i]);
          End;
        sec_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
      End;

      For i := 1 To num_of_output Do
        Begin
          sum := 0.0;
          For j := 1 to second_node_num Do
            Begin
              sum := sum + (sec_hid_out[j] * output_weight[j,i]);
            End;

```

```

        output_out[i] := 1.0 / (1.0 + exp(-1 * sum));
    End;

End;

{*****
* This procedure will calculate the error on each layer's nodes, *
* according to the error on the output. *
*****}

Procedure compute_errors;

Var
    i, j : Integer;

Begin
    For i := 1 to num_of_output do
        Begin
            output_err[i] := output_out[i] * (1 - output_out[i]) *
                             (target[i] - output_out[i]);
        End;

        For i := 1 to second_node_num do
            Begin
                sec_hid_err[i] := 0.0;
                For j := 1 to num_of_output do
                    Begin
                        sec_hid_err[i] := sec_hid_err[i] + ( output_err[j] *
                                                             output_weight[i,j]);
                    End;

                    sec_hid_err[i] := sec_hid_err[i] * sec_hid_out[i] *
                                     (1 - sec_hid_out[i]);
                End;

                For i := 1 to first_node_num do
                    Begin
                        first_hid_err[i] := 0.0;
                        For j := 1 to second_node_num do
                            Begin
                                first_hid_err[i] := first_hid_err[i] + ( sec_hid_err[j] *
                                                                           sec_hid_weight[i,j]);
                            End;

                            first_hid_err[i] := first_hid_err[i] * first_hid_out[i] *
                                                  (1 - first_hid_out[i]);
                        End;
                    End;
                End;
            End;

        End;

    End;

{*****
*This procedure will update the weights of each layer according to*
*the error on each node. *
*The 1st hidden layer will be updated, then the 2nd hidden layer, *
*****}

```

```

*and finally the output layer.
*****}

Procedure update_weights;

Var
  i, j : Integer;

Begin
  For i := 1 to first_node_num Do
    Begin
      For j := 1 to num_of_input Do
        Begin
          first_hid_weight[j,i] := first_hid_weight[j,i] + ( etha *
                                                                first_hid_err[i] * input_lay[j]);
        End;
      End;
    End;
  For i := 1 to second_node_num Do
    Begin
      For j := 1 to first_node_num Do
        Begin
          sec_hid_weight[j,i] := sec_hid_weight[j,i] + ( etha *
                                                                sec_hid_err[i] * sec_hid_out[j]);
        End;
      End;
    End;

  For i := 1 to num_of_output Do
    Begin
      For j := 1 to second_node_num Do
        Begin
          output_weight[j,i] := output_weight[j,i] + ( etha *
                                                                output_err[i] * sec_hid_out[j]);
        End;
      End;
    End;
  End;

  {*****}
  * This procedure will calculate the desired position and velocity*
  * by using a second order filter (IIR filter)
  {*****}

  Procedure sfiltin(xin :inarr; yk1:sarr; var yk2:sarr);

  Begin
    yk2[1] := (A11 * yk1[1]) + (A12 * yk1[2]) + (B1 * xin[1]);
    yk2[2] := (A21 * yk1[1]) + (A22 * yk1[2]) + (B2 * xin[1]);

    yk2[3] := (A11 * yk1[3]) + (A12 * yk1[4]) + (B1 * xin[2]);
    yk2[4] := (A21 * yk1[3]) + (A22 * yk1[4]) + (B2 * xin[2]);

  End;

```

```

{*****
* This procedure will put the curent position and velocity in the *
* previous position and velocity *
*****}

```

```

Procedure update(var yk1:sarr; yk2:sarr);

```

```

var
  i : Integer;

```

```

Begin

```

```

  For i := 1 to 4 do
  Begin
    yk1[i] := yk2[i];
  End;

```

```

End;

```

```

{*****
* This procedure will calculate the required torque by using *
* proportional and derivative (PD) controller and NN output. *
*****}

```

```

Procedure calctorq(Var torq : inarr; yd, yd1, ya : sarr);

```

```

Var
  torqe : inarr;
  torqn : inarr;

```

```

Begin

```

```

  torqe[1] := (yd1[1] - ya[1]) * kp + (yd1[2] - ya[2]) * kd ;
  torqe[2] := (yd1[3] - ya[3]) * kp + (yd1[4] - ya[4]) * kd ;

```

```

  data_set[ss,1] := ya[1];
  data_set[ss,2] := ya[2];
  data_set[ss,3] := yd[1];
  data_set[ss,4] := yd1[2];

```

```

  data_set[ss,5] := ya[3];
  data_set[ss,6] := ya[4];
  data_set[ss,7] := yd1[3];
  data_set[ss,8] := yd1[4];

```

```

  data_set[ss,9] := (yd[2] - yd1[2]) / 1.005016708E-2 ;
  data_set[ss,10] := (yd[4] - yd1[4]) / 1.005016708E-2 ;

```

```

  input_lay[1] := ya[1];
  input_lay[2] := ya[2];
  input_lay[3] := yd1[1];
  input_lay[4] := yd1[2];

```

```

  input_lay[5] := ya[3];
  input_lay[6] := ya[4];

```

```

input_lay[7] := yd1[3];
input_lay[8] := yd1[4];

input_lay[9] := (yd[2] - yd1[2]) / 1.005016708E-2 ;
input_lay[10] := (yd[4] - yd1[4]) / 1.005016708E-2 ;

prop_forward;

torqn[1] := (output_out[1] * divt1) - addt1;
torqn[2] := (output_out[2] * divt2) - addt2;

torq[1] := torqn[1] + torque[1];
torq[2] := torqn[2] + torque[2];

data_set[ss,11] := ( torq[1] + addt1 ) / divt1;
data_set[ss,12] := ( torq[2] + addt2 ) / divt2;

ss := ss + 1;

End;

{*****
* This procedure will simulate the dynamic of the 2DOF robot. *
* In this procedure Neural Network will be trained to estimate*
* the dynamics of the robot *
*****}

Procedure compute_f(Var ya : sarr ; t1 : inarr);
Var
    cos1,sin1,cos3,sin3,a,b,c,d,adcb : Double;
Begin
    cos1 := cos(ya[1]);
    sin1 := sin(ya[1]);

    cos3 := cos(ya[3]);
    sin3 := sin(ya[3]);

    a := m112 + 2.0*m212 + 2.0*m212*cos3;
    b := m212 + m212*cos3;
    c := m212 + m212*cos3;
    d := m212;
    adcb := a*d - c*b;

    f[1] := ya[2];
    f[2] := (d/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
            + (b/adcb) * m212*ya[2]*ya[2]*sin3 + (d/adcb) * t1[1]
            - (b/adcb) * t1[2];

    f[3] := ya[4];
    f[4] := -(c/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
            - (a/adcb) * m212*ya[2]*ya[2]*sin3 - (c/adcb) * t1[1]
            + (a/adcb) * t1[2];

```



```

End;

Procedure runge_kutta(Var y:sarr; t1 : inarr);

{*****
* This is the numerical solution to the nonlinear dynamics *
* of the robot manipulator. This procedure calls another *
* procedure, which is compute_f *
* compute_f, computes the derivatives of the states in *
* vector ya. These derivatives are stored in vector f(y(t)). *
* The state of the manipulator is calculated based on the *
* following recursion; *
*  $y(t + dt) = y(t) + 1/6 (k_0 + 2k_1 + 2k_2 + k_3)$  *
* *
* where *
*  $k_0 = f(y(t)) * dt$  *
*  $k_1 = f(y(t) + 1/2 k_0) * dt$  *
*  $k_2 = f(y(t) + 1/2 k_1) * dt$  *
*  $k_3 = f(y(t) + k_2) * dt$  *
* *
* The function f(y(t)) has to be calculated four times *
*****}

Var
  i,j : Integer;
  k : array[1..4,1..4] of Double;
  y1 : sarr;

Begin
  For i := 1 to 4 do
    Begin
      Compute_f(y,t1);

      {* If this is the first time through the loop present state
        of the manipulator will be stored and k0 and y(t) + 1/2k0
        which will be used as the argument for f(y(t) + 1/2k0) will
        be computed.

        *}

      If i = 1 then
        Begin
          for j := 1 to 4 do
            Begin
              y1[j] := y[j];
              k[i,j] := f[j] * dt;
              y[j] := y1[j] + 0.5 * k[i,j];
            End;
          End;

          If i = 2 then
            Begin
              for j := 1 to 4 do
                Begin
                  k[i,j] := f[j] * dt;

```

```

        y[j] := y1[j] + 0.5 * k[i,j];
    End;
End;

If i = 3 then
Begin
    for j := 1 to 4 do
    Begin
        k[i,j] := f[j] * dt;
        y[j] := y1[j] + 0.5 * k[i,j];
    End;
End;

If i = 4 then
Begin
    for j := 1 to 4 do
    Begin
        k[i,j] := f[j] * dt;
        { * compute updated state of the robot * }

        y[j] := y1[j] + (1.0/6.0)*(k[1,j] + 2.0*k[2,j] +
            2.0*k[3,j] + k[4,j]);

        { * The robot state has been updated * }
    End;
End;
End;

End;

{=====
=                               Main Program!!!!                               =
=====}

Begin
    Randomize;
    Assign(f1,'alarob1.out');
    Rewrite(f1);
    Assign(f2,'alarob2.out');
    Rewrite(f2);
    Assign(f3,'alarob3.out');
    Rewrite(f3);
    Assign(f4,'alarob4.out');
    Rewrite(f4);
    Assign(f5,'alarob5.out');
    Rewrite(f5);
    Assign(f6,'alarob6.out');
    Rewrite(f6);
    Assign(f7,'datawei.in');
    Reset(f7);

    For l := 1 to first_node_num do
    Begin
        For m := 1 to num_of_input do
        Begin
            read(f7,first_hid_weight[m,1]);
        End;
        readln(f7);
    End;
End;

```

```

End;
For l := 1 to second_node_num do
Begin
  For m := 1 to first_node_num do
  Begin
    read(f7,sec_hid_weight[m,1]);
  End;
  readln(f7);
End;
For l := 1 to num_of_output do
Begin
  For m := 1 to second_node_num do
  Begin
    read(f7,output_weight[m,1]);
  End;
  readln(f7);
End;
For ii := 1 to 4 do
Begin
  y1k1[ii] := 0.0;
  y1k2[ii] := 0.0;
  rout[ii] := 0.0;
End;
Count2 := 0;
While not KeyPressed do
Begin
  writeln(Count2);
  ss := 1;
  While ss <= st1 do
  Begin
    xinp[1] := Random;
    xinp[2] := Random;
    st2 := (Random(5) + 1) * 50;
    For jj := 1 to st2 do
    Begin
      While ss <= st1 do
      Begin
        sfiltin(xinp,y1k1,y1k2);
        calctorq(tin,y1k2,y1k1,rout);
        runge_kutta(rout,tin);
        update(y1k1,y1k2);
      End;
      End;
    End;
  End;
  For jj := 1 to 1000 do
  Begin
    ds := Random(498) + 2 ;
    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
    input_lay[5] := data_set[ds,5];
    input_lay[6] := data_set[ds,6];
    input_lay[7] := data_set[ds,7];
    input_lay[8] := data_set[ds,8];
    input_lay[9] := data_set[ds,9];
  End;
End;

```

```

        input_lay[10] := data_set[ds,10];
        target[1]     := data_set[ds,11];
        target[2]     := data_set[ds,12];
        prop_forward;
        compute_errors;
        update_weights;
    End;
    Count2 := Count2 + 1;
End;
count1 := 0;
ss := 1;
While ss <= st1 do
Begin
    xinp[1] := Random;
    xinp[2] := Random;
    st2 := (Random(5) + 1) * 50;
    For jj := 1 to st2 do
    Begin
        While ss <= st1 do
        Begin
            sfiltin(xinp,y1k1,y1k2);
            calctorq(tin,y1k2,y1k1,rout);
            runge_kutta(rout,tin);
            update(y1k1,y1k2);
            If ((count1 Mod 2) = 0 ) then
            Begin
                writeln(f1,y1k2[1],',',rout[1],',',y1k2[3]
, ', ',rout[3]));
            End;
            count1 := count1 + 1;
        End;
    End;
End;
count1 := 0;
For jj := 1 to 1000 do
Begin
    ds := Random(498) + 2 ;
    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
    input_lay[5] := data_set[ds,5];
    input_lay[6] := data_set[ds,6];
    input_lay[7] := data_set[ds,7];
    input_lay[8] := data_set[ds,8];
    input_lay[9] := data_set[ds,9];
    input_lay[10] := data_set[ds,10];
    target[1]     := data_set[ds,11];
    target[2]     := data_set[ds,12];
    prop_forward;
    compute_errors;
    update_weights;
    count1 := count1 + 1;
    If ((count1 Mod 4) = 0 ) then
    Begin
        For l := 1 to num_of_output do

```

```

        Begin
            For m := 1 to 4 do
                Begin
                    write(f5,output_weight[m,1], ' ');
                End;
            End;
            writeln(f5);
            For l := 1 to 4 do
                Begin
                    For m := 1 to 2 do
                        Begin
                            write(f3,first_hid_weight[m,1], ' ');
                        End;
                    End;
                    writeln(f3);
                    For l := 1 to 4 do
                        Begin
                            For m := 1 to 2 do
                                Begin
                                    write(f4,sec_hid_weight[m,1], ' ');
                                End;
                            End;
                            writeln(f4);
                            write(f6,target[1], ' ',output_out[1], ' ');
                            write(f6,target[2], ' ',output_out[2], ' ');
                            writeln(f6);
                        End;
                    End;
                End;
            End;
            For l := 1 to first_node_num do
                Begin
                    For m := 1 to num_of_input do
                        Begin
                            write(f2,first_hid_weight[m,1], ' ');
                        End;
                    End;
                    writeln(f2);
                End;
            End;
            For l := 1 to second_node_num do
                Begin
                    For m := 1 to first_node_num do
                        Begin
                            write(f2,sec_hid_weight[m,1], ' ');
                        End;
                    End;
                    writeln(f2);
                End;
            End;
            For l := 1 to num_of_output do
                Begin
                    For m := 1 to second_node_num do
                        Begin
                            write(f2,output_weight[m,1], ' ');
                        End;
                    End;
                    writeln(f2);
                End;
            End;
            Close(f1);
            Close(f2);
            Close(f3);
            Close(f4);
            Close(f5);
            Close(f6);

```

end.

B.3 Neural Network Test

```

program simites;
{*****}
* Name : Zaryab Hamavand *
* Date : Nov 1993 *
* *
* This program is testing the Neural *
* Network controller. *
{*****}
{$N+}
{$R+} {Enable range checking}
uses CRT;
Const
    num_of_input    = 10;
    num_of_output    = 2;
    first_node_num   = 35;
    second_node_num  = 35;
    etha             = 0.2;

    A11              = 0.9998;
    A12              = 0.0098;
    A21              = -0.0392;
    A22              = 0.9606;

    B1               = 0.0002;
    B2               = 0.0392;

    st1              = 500;

    dt               = 0.01;
    kd               = 35.0;
    kp               = 60.0;

    m112             = 1.0;
    m212             = 2.0;
    wtof             = -1;

    addt1            = 20.0;
    divt1            = 50.0;
    addt2            = 10.0;
    divt2            = 25.0;

Type
    sarr = array [1..4] of Double;
    inarr = array [1..2] of Double;

Var
    f1,f2,f3,f7      : text;
    y1k1,y1k2,rout,f : sarr;
    xinp ,tin        : inarr;
    ii,jj,kk         : Integer;
    intwei           : Double;
    n,l,ss,ds,st2    : Integer;
    Count1,Count2    : Integer;

    first_hid_weight : array[1..num_of_input, 1..first_node_num] of Double;
    sec_hid_weight   : array[1..first_node_num, 1..second_node_num] of Double;
    output_weight    : array[1..second_node_num, 1..num_of_output] of Double;

```

```

first_hid_out  array[1..first_node_num] of Double;
first_hid_err  : array[1..first_node_num] of Double;

sec_hid_out   : array[1..second_node_num] of Double;
sec_hid_err   : array[1..second_node_num] of Double;

output_out    : array[1..num_of_output] of Double;
output_err    : array[1..num_of_output] of Double;
target        : array[1..num_of_output] of Double;

input_lay     : array[1..num_of_input] of Double;
data_set      : array[1..500, 1..12] of double;

{*****
* This procedure will propagate the input vector through the      *
* network and will produce output. This network consists of two hidden *
* layers.                                                         *
*****}

procedure prop_forward;

Var
  i, j : Integer;
  sum : Double;

Begin
  For i := 1 To first_node_num Do
    Begin
      sum := 0.0;
      For j := 1 to num_of_input Do
        Begin
          sum := sum + (first_hid_weight[j,i] * input_lay[j]);
        End;
      first_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
    End;

    For i := 1 To second_node_num Do
      Begin
        sum := 0.0;
        For j := 1 to first_node_num Do
          Begin
            sum := sum + (first_hid_out[j] * sec_hid_weight[j,i]);
          End;
        sec_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
      End;

      For i := 1 To num_of_output Do
        Begin
          sum := 0.0;
          For j := 1 to second_node_num Do
            Begin
              sum := sum + (sec_hid_out[j] * output_weight[j,i]);
            End;
          output_out[i] := 1.0 / (1.0 + exp(-1 * sum));
        End;
      End;
    End;
  End;

```


End;

```
{*****
* This procedure will calculate the desired position and velocity *
* by using a second order filter (IIR filter) *
*****}
```

Procedure sfiltin(xin : inarr; yk1: sarr; var yk2: sarr);

Begin

```
yk2[1] := (A11 * yk1[1]) + (A12 * yk1[2]) + (B1 * xin[1]);
yk2[2] := (A21 * yk1[1]) + (A22 * yk1[2]) + (B2 * xin[1]);

yk2[3] := (A11 * yk1[3]) + (A12 * yk1[4]) + (B1 * xin[2]);
yk2[4] := (A21 * yk1[3]) + (A22 * yk1[4]) + (B2 * xin[2]);
```

End;

```
{*****
* This procedure will put the current position and velocity in the *
* previous position and velocity *
*****}
```

Procedure update(var yk1: sarr; yk2: sarr);

var

i : Integer;

Begin

```
For i := 1 to 4 do
Begin
yk1[i] := yk2[i];
End;
```

End;

```
{*****
* This procedure will calculate the required torque by using *
* proportional and derivative (PD) controller and NN output. *
* The desired torque will be calculated by using the dynamic *
* equation of the manipulator. *
*****}
```

Procedure calctorq(Var torq : inarr; yd, yd1, ya : sarr);

Var

```
torqe : inarr;
torqn : inarr;
torqd : inarr;
cos3, sin3, am, bm, cm, dm, ca, cb : Double;
```

Begin

```
cos3 := cos(ya[3]);
```

```

sin3 := sin(ya[3]);
am := m112 + 2.0*m212 + 2.0*m212*cos3;
bm := m212 + m212*cos3;
cm := m212 + m212*cos3;
dm := m212;
ca := -2.0*m212*ya[2]*ya[4]*sin3 - m212*ya[4]*ya[4]*sin3;
cb := m212*ya[2]*ya[2]*sin3;

torqe[1] := (yd1[1] - ya[1]) * kp + (yd1[2] - ya[2]) * kd ;
torqe[2] := (yd1[3] - ya[3]) * kp + (yd1[4] - ya[4]) * kd ;
input_lay[1] := ya[1];
input_lay[2] := ya[2];
input_lay[3] := yd1[1];
input_lay[4] := yd1[2];
input_lay[5] := ya[3];
input_lay[6] := ya[4];
input_lay[7] := yd1[3];
input_lay[8] := yd1[4];
input_lay[9] := (yd[2] - yd1[2]) / 1.005016708E-2 ;
input_lay[10] := (yd[4] - yd1[4]) / 1.005016708E-2 ;
prop_forward;
torqn[1] := (output_out[1] * divt1) - addt1;
torqn[2] := (output_out[2] * divt2) - addt2;
torq[1] := torqn[1] + torqe[1];
torq[2] := torqn[2] + torqe[2];
torqd[1] := (am * input_lay[9]) + (bm * input_lay[10]) + ca;
torqd[2] := (cm * input_lay[9]) + (dm * input_lay[10]) + cb;
If ((ss Mod 5) = 0 ) then
Begin
    writeln(f2,torqe[1],', ',torqn[1],', ',torq[1],', ',torqd[1]);
    writeln(f3,torqe[2],', ',torqn[2],', ',torq[2],', ',torqd[2]);
End;
ss := ss + 1;
End;

{*****
* This procedure will simulate the dynamic of the 2DOF robot. *
* In this procedure Neural Network will be trained to estimate *
* the dynamics of the robot *
*****}

Procedure compute_f(Var ya : sarr ; t1 : inarr);

Var
    cos1,sin1,cos3,sin3,a,b,c,d,adcb : Double;

Begin
    cos1 := cos(ya[1]);
    sin1 := sin(ya[1]);
    cos3 := cos(ya[3]);
    sin3 := sin(ya[3]);
    a := m112 + 2.0*m212 + 2.0*m212*cos3;
    b := m212 + m212*cos3;
    c := m212 + m212*cos3;
    d := m212;

```

```

adcb := a*d - c*b;
f[1] := ya[2];
f[2] := (d/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
        + (b/adcb) * m212*ya[2]*ya[2]*sin3 + (d/adcb) * t1[1]
        - (b/adcb) * t1[2];
f[3] := ya[4];
f[4] := -(c/adcb) * (2.0*m212*ya[2]*ya[4]*sin3 + m212*ya[4]*ya[4]*sin3)
        - (a/adcb) * m212*ya[2]*ya[2]*sin3 - (c/adcb) * t1[1]
        + (a/adcb) * t1[2];

End;
{*****
* This is the numerical solution to the nonlinear dynamics *
* of the robot manipulator. This procedure calls another *
* procedure, which is compute_f *
* compute_f, computes the derivatives of the states in *
* vector ya. These derivatives are stored in vector f(y(t)). *
* The state of the manipulator is calculated based on the *
* following recursion; *
*  $y(t + dt) = y(t) + 1/6 (k_0 + 2k_1 + 2k_2 + k_3)$  *
* *
* where *
*  $k_0 = f(y(t)) * dt$  *
*  $k_1 = f(y(t) + 1/2 k_0) * dt$  *
*  $k_2 = f(y(t) + 1/2 k_1) * dt$  *
*  $k_3 = f(y(t) + k_2) * dt$  *
* *
* The function f(y(t)) has to be calculated four times *
*****}

Var
  i,j : Integer;
  k : array[1..4,1..4] of Double;
  y1 : sarr;

Begin
  For i := 1 to 4 do
    Begin
      Compute_f(y,t1);

      {* If this is the first time through the loop present state
        of the manipulator will be stored and k0 and y(t) + 1/2k0
        which will be used as the argument for f(y(t) + 1/2k0) will
        be computed.

      *}

      If i = 1 then
        Begin
          for j := 1 to 4 do
            Begin
              y1[j] := y[j];
              k[i,j] := f[j] * dt;
              y[j] := y1[j] + 0.5 * k[i,j];
            End;
          End;
        End;
      End;
    End;
  End;

```

```

End;
If i = 2 then
Begin
  for j := 1 to 4 do
  Begin
    k[i,j] := f[j] * dt;
    y[j]   := y1[j] + 0.5 * k[i,j];
  End;
End;
End;

If i = 3 then
Begin
  for j := 1 to 4 do
  Begin
    k[i,j] := f[j] * dt;
    y[j]   := y1[j] + 0.5 * k[i,j];
  End;
End;
End;

If i = 4 then
Begin
  for j := 1 to 4 do
  Begin
    k[i,j] := f[j] * dt;
    { * compute updated state of the robot * }

    y[j] := y1[j] + (1.0/6.0)*(k[1,j] + 2.0*k[2,j] +
      2.0*k[3,j] + k[4,j]);

    { * The robot state has been updated * }
  End;
End;
End;

End;
{=====
=                               Main Program!!!!                               =
=====}
Begin
  Randomize;
  Assign(f1,'simpos1.out');
  Rewrite(f1);
  Assign(f2,'simtor1.out');
  Rewrite(f2);
  Assign(f3,'simtor2.out');
  Rewrite(f3);
  Assign(f7,'simt.in');
  Reset(f7);
  For l := 1 to first_node_num do
  Begin
    For m := 1 to num_of_input do
    Begin
      read(f7,first_hid_weight[m,l]);
    End;
    readln(f7);
  End;
  For l := 1 to second_node_num do
  Begin

```

```

        For m := 1 to first_node_num do
        Begin
            read(f7,sec_hid_weight[m,1]);
        End;
        readln(f7);
    End;
    For l := 1 to num_of_output do
    Begin
        For m := 1 to second_node_num do
        Begin
            read(f7,output_weight[m,1]);
        End;
        readln(f7);
    End;
    For ii := 1 to 4 do
    Begin
        y1k1[ii] := 0.0;
        y1k2[ii] := 0.0;
        rout[ii] := 0.0;
    End;
    ss := 0;
    While ss <= 2500 do
    Begin
        xinp[1] := Random;
        xinp[2] := Random;
        st2 := (Random(5) + 1) * 50;
        For jj := 1 to st2 do
        Begin
            sfiltin(xinp,y1k1,y1k2);
            calctorq(tin,y1k2,y1k1,rout);
            runge_kutta(rout,tin);
            update(y1k1,y1k2);
            If ((ss Mod 5) = 0 ) then
            Begin
                writeln(f1,y1k2[1], ' ', rout[1], ' ', y1k2[3], ' ', rout[3]);
            End;
        End;
    End;
    End;
    Close(f1);
    Close(f2);
    Close(f3);
    Close(f7);
end.

```

Appendix C

Experiment Software Listing

In this appendix the listing of the software which was used for the Experiment has been shown. First the program used to define the trajectory for the manipulator has been shown. In the next section all the program used to control the manipulator and collect the required data have been presented. Finally, the software used for training the Neural Network has been shown.

C.1 Define The Trajectory

```

program exp2pat;
{$N+}
{$R+} {Enable range checking}
{*****}
* Name: Zaryab Hamavand *
* Date: Jan 1994 *
* *
* This program will produce a trajectory for the *
* manipulator by using third order filter. *
{*****}
uses CRT,DOS;
Const
    A11      = 0.9998;
    A12      = 0.0199;
    A13      = 0.0002;

    A21      = -0.0226;
    A22      = 0.9863;
    A23      = 0.0172;

    A31      = -2.1490;
    A32      = -1.3120;
    A33      = 0.7284;

    B1       = 0.0002;
    B2       = 0.0226;
    B3       = 2.1490;
    st1      = 750;
    dt       = 0.02;
Type
    sarr = array [1..6] of single;
    inarr = array [1..2] of Double;
Var
    f1,f2    : text;
    y1k1,y1k2 : sarr;
    xinp,xinp1 : inarr;
    y1k11,y1k21 : sarr;

    ii,jj,kk : Integer;
    diff      : single;
    ss,st2    : Integer;
    year,month,day,dayofweek: word;

{*****}
* This procedure will calculate the desired position and velocity *

```

```

* by using a second order filter (IIR filter)
*****}

Procedure sfiltin(xin :inarr; yk1:sarr; var yk2:sarr);

Begin
    yk2[1] := (A11 * yk1[1]) + (A12 * yk1[2]) + (A13 * yk1[3])
              + (B1 * xin[1]);
    yk2[2] := (A21 * yk1[1]) + (A22 * yk1[2]) + (A23 * yk1[3])
              + (B2 * xin[1]);
    yk2[3] := (A31 * yk1[1]) + (A32 * yk1[2]) + (A33 * yk1[3])
              + (B3 * xin[1]);
    yk2[4] := (A11 * yk1[4]) + (A12 * yk1[5]) + (A13 * yk1[6])
              + (B1 * xin[2]);
    yk2[5] := (A21 * yk1[4]) + (A22 * yk1[5]) + (A23 * yk1[6])
              + (B2 * xin[2]);
    yk2[6] := (A31 * yk1[4]) + (A32 * yk1[5]) + (A33 * yk1[6])
              + (B3 * xin[2]);
    ss := ss + 1;
End;

{*****}
* This procedure will put the curent position and velocity in the
* previous position and velocity
{*****}

Procedure update(var yk1:sarr; yk2:sarr);

var
    i : Integer;

Begin
    For i := 1 to 6 do
        Begin
            yk1[i] := yk2[i];
        End;
    End;
End;

{=====
=                      Main Program!!!!                      =
=====}

Begin
    Randomize;

    getdate(year,month,day,dayofweek);
    Assign(f1,'d41.via');
    Rewrite(f1);
    Assign(f2,'d41.vin');
    Rewrite(f2);
    writeln(f2, 'file name : exp2pa11 ', ' ', ' Date: ',month, '/',
            day, '/',year);

```



```

writeln(f2, 'Code used to produce date : exp2pat.pas ');
writeln(f2, ' data: Sampling freq(Hz),Run time (Sec),# data pts');
writeln(f2, ' 50 15.00 750');
writeln(f2, ' Comments : ');
writeln(f2, ' P.E. trajectory for 2 motors, consisting of steps');
writeln(f2, ' combination with different time interval. ');
writeln(f2, ' Minimum & Maximum difference 50, 150 degree');
For ii := 1 to 6 do
Begin
    y1k1[ii] := 0.0;
    y1k2[ii] := 0.0;

End;
y1k1[1] := 140.0;
y1k1[4] := 37.0;
writeln(f1,y1k1[1], ' ',y1k1[4]);
ss := 1;
While ss <= 750 do
Begin
    xinp[1] := (Random(120) + 140.0);
    xinp[2] := (Random(100) + 37);
    diff := xinp[1] - xinp[2];
    While((diff < 50.0) or (diff > 150.0)) do
    Begin
        xinp[1] := (Random(120) + 140.0);
        xinp[2] := (Random(100) + 37.0);
        diff := xinp[1] - xinp[2];
    End;

    st2 := (Random(4) + 1) * 25;
    For jj := 1 to st2 do
    Begin
        sfiltin(xinp,y1k1,y1k2);
        update(y1k1,y1k2);
        If ss<= 750 then
        Begin
            writeln(f1,y1k1[1], ' ',y1k1[4]);
        End;
    End;

End;
Close(f1);
Close(f2);

end.

```

C.2 Robot Control

In this section the software used to control the robot has been shown. The main program listing has been presented at the beginning of this section. In addition the subprogram has been shown.

C.2.1 Main Program

```
program nncontro;
```

```
{-----}
{Written by: Gabriel D. Warshaw}
{Revised: Mar.12,1993}
{
{Revised by : Zaryab Hamavand}
{Date      : Feb 1994}
{This program performs adaptive control of 2
degrees-of-freedom of the Carleton Direct Drive
Manipulator, using Slotine and Li's Direct algorithm,
with a compensation term added.
The coupled non-linear dynamic model is used, including
torque bias terms for the (time-varying) torque bias of
the servo amplifiers. Five parameters are estimated.
The manipulator is made to follow a pre-defined path in
joint space.
This path (position) is loaded from a file, and is
pre-filtered to limit the bandwidth of the desired
trajectory, and to generate desired velocity and
acceleration. A butterworth low-pass filter is used for
velocity smoothing. The program is interrupt-driven
(instead of polling) for the A/D conversion.}
{-----}
```

```
uses dos,crt,datacard,filter,m_arith;
```

```
const
```

```
    {i/f card register addresses}
    ad_status_addr = $218;
    ad_chan_addr = $219;
    da0_loaddr = $21A; {same as a/d data low byte }
    da0_hiaddr = $21B; { " " " hi " }
    dd_cntrlreg_addr = $228;
    dd_port0_addr = $229; {motor 1 position low byte}
    dd_port1_addr = $22A; { " 1 " high " }
    dd_port2_addr = $22B; { " 2 " low " }
    dd_port3_addr = $22C; { " 2 " high " }

    maxpoints = 800; {maximum number of data points to be saved}
    maxvia = 1000; { " " " via " " " " }

    nparam = 5; {number of model parameters to be estimated}
    torque_limit=4.9; {control volt. amplitude limit i.e. s/w
defined saturation point.}

    njoint=3; {code for number of joints under control:
              1 = joint 1
              2 = joint 2
              3 = joint 1 and 2 }

    name_of_code='Nncontro';

    max_vel=250.; {maximum possible velocity used to detect faulty
position data, in degrees/second}
```

```

lambda=10.0; {constant for calculation of filtered error}
gamma_scaling=array[1..nparam] of
single=(1.,1.,0.01745,3283.0,3283.0);

freqs=50; {effective sampling frequency, Hz (for 2 a/d's),
          a/d interrupts set for 2*freqs}

data_items=12; {number of different data items for file output}

pole=-10.0; {location of (triple) pole for pre-filter}
          {-10.0 gives BW = 5.0 rad/sec = 0.80 Hz}

bandwidth=25.0; {bandwidth (Hz) for butterworth low-pass filter}

a= 0.0 {0.0179}; {compensation term gain parameter}
norm_sprime= 0.0 {5*3.76/2.0}; {compensation term "dead zone"
                                parameter}
      {bounds on allowable range of parameter estimates}
para_min: array[1..nparam] of single = (0.0,0.0,-0.002,-1.0,-1.0);
para_max: array[1..nparam] of single = (0.02,0.01,0.002,1.0,1.0);

{interrupt constants}
PIC_00 = $20; {Programmable Interrupt Controller control register}
PIC_01 = $21; { " " " " " mask register}
EOI = $20; {end of interrupt command to PIC}

{Neural network setup}

num_of_input      = 10;
num_of_output     = 2;
first_node_num    = 40;
second_node_num   = 40;
addt1             = 6.0;
divt1             = 12.0;
addt2             = 2.0;
divt2             = 5.0;

var
dummy_input, {dummy value of a/d input (binary)}
step, {current number of time step for desired trajectory}
nsteps, {number of steps for data collection during run}
numb_via, {number of trajectory via points}
divisor, {number used to index the via points}
counter, {current step used for data storage}
skip, {number of time steps to skip for data output}
skipcount, {current skip count}
icount, {counter used in ISR for sequencing}
glitch_count1,glitch_count2
      {counters used for faulty position measure. detection}
      :integer;
bkgnd_count {count of no. of complete cycles of "background" tasks}
      :longint;

hr,min,sec,frac {arguments from procedure which gets time}

```

```

:word;

pos1,pos2,      {joint angular positions in degrees}
pos1last,pos2last, {last values of motor 1 and 2 positions}
pos1_init,pos2_init, {actual values of motor positions at start of
run}
t,      (* Time from start of run *)
dt,dt2,  (* Time interval for a step, twice the time interval *)
dtdtover2, {time interval squared over 2 (speeds calculation)}
reject, {max. reasonable change in position over 1 sampling
interval}
ftime, (* Final time of the run *)
time_via, {length of time for run of loaded trajectory}
freq_via, {sampling frequency for loaded via points}
gamma, (* gain for parameter adaptation law *)
g,      {gain for compensation term}
norms, {norm of "s" vector}
max_normds, {max. value of the norm of the change in "s"}
tmax {time at which "max_normds" occurred}
      :single;
connn, {Neural network control}
kv, {velocity feedback gain}
control, (* Control input for motors, in volts *)
posd, {array of desired positions for motor 1 and 2}
accd, { " " " " accelerations " " " " " }
err, (* robot position error *)
derr, (* error in the velocity *)
velr,accr, {reference velocities and accelerations}
s, {filtered tracking error}
slast, {"s" from previous time step}
posdlast,veldlast, {desired pos.& vel. for previous step}
posdf,posdfm1, {arrays of pre-filtered desired ...}
veldf,veldfm1, {...positions, velocities and accelerations...}
accdf,accdfm1 {...for motor 1 and 2}
      : array[1..2] of single;

vel, {joint angular velocities in deg/sec}
velf,velfm1, {low-pass filtered velocity, and previous values}
velfdot,velfdotm1 {derivatives of low-pass filtered velocity}
      : state;

phi {regression matrix}
      : array[1..2,1..nparam] of single;

para {parameter vector}
      :array[1..nparam] of single;

data: array[1..data_items,1..maxpoints] of single; {array of data
for output to a file (state of robot)}

trajectory: array[1..2,1..maxvia] of single; {array of trajectory
(via) points input from a file}

outdata,outinfo,indata,ininfo: text;      {i/o file assignment names}

```

```

filename: string[20];
comments: string[160];

enabled,      {flag indicating if motors are enabled}
done_flag,    { "          "          end of run time}
ad_error      { "          "          occurance of a/d conversion error}
: boolean;

yes_comment,  {variable for user to select input of comments}
key_press,    {dummy variable for user key press action}
choice: char; {variable for input of user selection character}

fb:state3;     {vector of input term coefficients for pre-filter}
fa:matrix3;    {matrix of AR      "          "          "          "          "          }

bfb:state;     {vector of input term coeff's for butterworth filter}
bfa:matrix2;   {matrix of AR      "          "          "          "          "          }

old_level2_intr, {old level 2 interrupt vector to be saved}
exitsave:pointer; {pointer saved for exit pointer used by system}

{Neural Network variable}

f2      : text;
m, 1    : Integer;

first_hid_weight:array[1..num_of_input,1..first_node_num] of Single;
sec_hid_weight:array[1..first_node_num,1..second_node_num] of Double;
output_weight:array[1..second_node_num,1..num_of_output] of Double;
first_hid_out:array[1..first_node_num] of Double;
sec_hid_out : array[1..second_node_num] of Double;
output_out : array[1..num_of_output] of Double;
input_lay : array[1..num_of_input] of Double;

{*****
* This procedure will propagate the input vector through the          *
* network andwill produce output. This network consists of two hidden *
* layers.                                                              *
*****}

procedure prop_forward;

Var
    ii, jj : Integer;
    sum : Double;

Begin
    For ii := 1 To first_node_num Do
        Begin
            sum := 0.0;
            For jj := 1 to num_of_input Do
                Begin
                    sum := sum + (first_hid_weight[jj,ii] * input_lay[jj]);
                End;
            first_hid_out[ii] := 1.0 / (1.0 + exp(-1 * sum));
        End;
    End;
End;

```

```

For ii := 1 To second_node_num Do
Begin
    sum := 0.0;
    For jj := 1 to first_node_num Do
    Begin
        sum := sum + (first_hid_out[jj] * sec_hid_weight[jj,ii]);
    End;
    sec_hid_out[ii] := 1.0 / (1.0 + exp(-1 * sum));
End;

For ii := 1 To num_of_output Do
Begin
    sum := 0.0;
    For jj := 1 to second_node_num Do
    Begin
        sum := sum + (sec_hid_out[jj] * output_weight[jj,ii]);
    End;
    output_out[ii] := 1.0 / (1.0 + exp(-1 * sum));
End;

End;

(*****)
procedure check_status(first_addr,second_addr:integer;
                      var enable_status:boolean);

const bit6 = $40;

begin
    if(port[first_addr] AND port[second_addr] AND bit6) = bit6 then
        enable_status:=true
    else enable_status:=false;
end;

(*****)
procedure initialize_interfaces;

begin
{initialize a/d}
    port[dd_cntrlreg_addr]:=0; {set all d/d ports for input}
    port[ad_status_addr]:=$10; {set a/d to mode 0, clear error bit}
    delay(50);
    dummy_input:= port[da0_lcaddr]+(port[da0_hiaddr])shl 8;
    port[ad_status_addr]:=$15; {set a/d to mode 1, clear error bit,
                              enable interrupt}
    if(timer_freq(2*freqs)) then {"timer_freq" sets timer for chosen
                                sample rate and detects error}
    begin
        writeln('Invalid sampling frequency !!');
        writeln('Program aborted');
        halt;
    end;

{set d/a outputs to 0 Volts}

```

```

    send_volt_da(1,0);
    send_volt_da(2,0);

{set a/d conversion error flag to false}
    ad_error:=false;

{check to see if motor (servos) are enabled}
    check_status(dd_port1_addr,dd_port3_addr,enabled);
    if enabled=false then begin
        writeln('**Motor is disabled --> enable it now.');
```

(press any key when ready)');

```
        key_press:=readkey;
        end; {if}
        enabled:=true;
    end;

    (*****
procedure create_info_file;
(* This procedure creates an information file with the same
   name as the data file but with a ".inf" extension. The
   file contains numeric and text information which describes
   the run which produced the data.*)

var year,month,day,dayofweek: word;
    i:integer;

begin

    getdate(year,month,day,dayofweek);

    rewrite(outinfo);
    writeln(outinfo,'filename: ',filename,'           Date: ',month,'/',
              day,'/',year);
    writeln(outinfo,'code used to produce data: "',name_of_code,'"');
    writeln(outinfo,' data: sampling freq(Hz),runtime(sec)',
              ',skip interval,# data pts, joint #');
    writeln(outinfo,' successive lines list: Kp,Kv');
    writeln(outinfo);
    writeln(outinfo,freqs,' ',ftime:6:2,' ',skip,' ',counter,' ',njoint);
    writeln(outinfo,lamba*kv[1]:10:5,kv[1]:10:5);
    writeln(outinfo,lamba*kv[2]:10:5,kv[2]:10:5);
    writeln(outinfo);

    write('Do you wish to enter comments for the information file (y/n)?
          ');
    readln(yes_comment);
    if ((yes_comment = 'y') OR (yes_comment = 'Y')) then begin
        writeln(' Enter comments, terminate with <CR> (max 160
                  characters):');
        readln(comments);
        writeln(outinfo,'Comments:');
        writeln(outinfo,comments);
        end;
    writeln(outinfo,'Gamma= ',gamma:12);
    writeln(outinfo,'Gamma scaling vector= ');
    for i:=1 to nparam do write(outinfo, gamma_scaling[i]:11);
    writeln(outinfo);

```

```

        writeln(outinfo,'Compensator parameters: a= ',a:10:3,
                ', norm_sprime= ',norm_sprime:10:3);
        writeln(outinfo,'Max. norm of delta_s = ',max_normds:11,
                ', @ t= ',tmax:7:2,' sec');

        close(outinfo);
    end;

    (*****)
    procedure load_trajectory_file;

    var i: integer; {loop index}

    begin
        {prompt user and load input files}
        writeln('Enter path and name for trajectory input file ');
        writeln(' (extensions ".via" for the data file and ".vin" for the');
        writeln(' matching information file will be appended
            automatically):');
        readln(filename);
        assign(indata,filename+'.via');
        reset(indata);
        assign(ininfo,filename+'.vin');
        reset(ininfo);

        writeln('--> data being loaded from file.....');
        readln(ininfo);
        readln(ininfo);
        readln(ininfo);
        readln(ininfo,freq_via,time_via,numb_via);
        for i:=1 to numb_via do readln(indata,trajectory[1,i],trajectory[2,i]);
        close(ininfo);
        close(indata);

        divisor:=round(freqs/freq_via);

    end;

    (*****)
    procedure initialize_parameters;
    (* This procedure initializes all the parameters in
        the system, including those input by the user. *)

    var i,j:integer; {loop indices}

    begin
        {set up output files}
        writeln('Enter path and name for data output file ');
        writeln(' (extensions ".dat" for the data file and ".inf" for the');
        writeln(' matching information file will be appended
            automatically):');
        readln(filename);
        assign(outdata,filename+'.dat');
        assign(outinfo,filename+'.inf');

        write('Enter Motor 1 velocity feedback gain: ');

```



```

readln(kv[1]);
write('Enter Motor 2 velocity feedback gain: ');
readln(kv[2]);
write('Enter adaptation gain, Gamma: ');
readln(gamma);
writeln('Run will take ',time_via:6:2,' seconds. ');
ftime:=time_via;
write('Do you wish to specify a different run time (y/n)? ');
readln(choice);
if (choice='Y') or (choice='y') then begin
    write('Enter run time (seconds): ');
    readln(ftime); end{if};
write('Enter number of data points to be stored during the run: ');
readln(nsteps);
skip:=round((ftime*freqs)/nsteps);
writeln('--> At ',freqs,' Hz sampling, data will be stored'
        , ' every <',skip,'> time steps');

{initialize variables}
skipcount:=skip;
t:=0.0;
dt:=1.0/freqs; dt2:=dt*2.0; dtdtover2:=dt*dt/2.0;
counter:=0;
icount:=1;
bkgnd_count:=0;
glitch_count1:=1; glitch_count2:=1;
done_flag:=false;
step:=0;
max_normds:=0.0;

reject:=max_vel*dt; {calculate rejection value for faulty position
                    measurement}

for i:=1 to nparam do para[i]:=0.0; {initialize parameter estimates}
for i:=1 to 2 do                    {zero the regression matrix}
    for j:=1 to nparam do
        phi[i,j]:=0.0;

    {set regression terms for torque bias parameters to constant values}
    phi[1,4]:=-1.0; phi[1,5]:=0.0;
    phi[2,4]:=0.0; phi[2,5]:=-1.0;

    for i:=1 to 2 do begin
        posd[i]:=0.0;
        posdf[i]:=0.0; veldf[i]:=0.0; accdf[i]:=0.0;
        velf[i]:=0.0; velfdot[i]:=0.0; s[i]:=0.0;
    end;{for}

    writeln('...computing pre-filter and butterworth filter
parameters...');

    pre_filter3(dt,pole,fa,fb); {compute filter coefficients for desired
path 3rd order pre-filter}

    butterworth2(dt,bandwidth*2.0*pi,bfa,bfb); {compute filter coeff's
for butterworth low-pass filtering of velocity

```

```

measurements}

end;

(*****)
procedure compute_desired_state;

(* This procedure computes the desired position
   and velocity of the manipulator.*)

const twopi=6.283185;
      f=1.0; {freq.for sine input (Hz)}
      w=twopi*f;
      amplitude=10.0; {amplitude of sine input (deg)}
      amplitude_2=1.5;
      w_2=2.7*w;

var index, i: integer;

begin
{select the appropriate trajectory via point}
  index:=(step div divisor)+1;
  if index > numb_via then begin {return to begin. of trajectory}
    step:=0; index:=1; end; {if}
  for i:=1 to 2 do posd[i]:=trajectory[i,index];
  step:=step+1;

{store desired states from last step}
  posdlast[1]:=posdf[1]; veldlast[1]:=veldf[1];
  posdlast[2]:=posdf[2]; veldlast[2]:=veldf[2];

{pre-filter the desired positions, to ensure that the requested
 trajectory stays within the system bandwidth -- generate
 pre-filtered position for one time step into the future}
  for i:=1 to 2 do begin
    posdfm1[i]:=posdf[i];
    veldfm1[i]:=veldf[i];
    accdfm1[i]:=accdf[i];
    posdf[i]:=fa[1,1]*posdfm1[i]+fa[1,2]*veldfm1[i]
      +fa[1,3]*accdfm1[i]+fb[1]*posd[i];
    veldf[i]:=fa[2,1]*posdfm1[i]+fa[2,2]*veldfm1[i]
      +fa[2,3]*accdfm1[i]+fb[2]*posd[i];
    accdf[i]:=fa[3,1]*posdfm1[i]+fa[3,2]*veldfm1[i]
      +fa[3,3]*accdfm1[i]+fb[3]*posd[i];
  end; {for}

  (*
{sinusoidal input of position for motor 2}
  posdf[2]:=pos2_init+amplitude - (amplitude*cos(w*t)
    +amplitude_2*cos(w_2*t) );
  veldf[2]:=amplitude*w*sin(w*t) +amplitude_2*w_2*sin(w_2*t) ;
  accd[2]:=amplitude*w*w*cos(w*t) +amplitude_2*w_2*w_2*cos(w_2*t) ;
  *)
end;

(*****)

```

```

procedure get_initial_positions;
  {gets initial value of motor positions}

begin
  pos1_init:= get_position_d(1);
  pos1:=pos1_init;
  pos1last:=pos1;
  posdf[1]:=pos1;

  pos2_init:= get_position_d(2);
  pos2:=pos2_init;
  pos2last:=pos2;
  posdf[2]:=pos2,

end;

{*****}
procedure save_data_to_file;

var i,j:integer; {loop indices}

begin
  writeln('--> data being saved to file.....');
  rewrite(outdata);
  if counter > maxpoints then counter:=maxpoints;
  for i := 1 to counter do
    begin
      for j := 1 to data_items do write(outdata,data[j,i], ' ');
      writeln(outdata);
    end;
  close(outdata);
end;

{*****}
procedure get_measurements_1;

begin
  {get velocity of motor 1}
  vel[1] := get_vel_d_i(1,ad_error);
  if ad_error then begin
    writeln('A/D conversion error has occurred -- PROGRAM ABORTED');
    halt;
  end;

  {start a/d conversion for channel 1 (motor 2), gain=1}
  port[ad_chan_addr]:=1;

  {get position of motor 1
  including check and correct for spurious position data}
  pos1:= get_position_d(1);
  if (abs(pos1-pos1last) > glitch_count1~reject) then begin
    pos1:=pos1last;
    glitch_count1:=glitch_count1+1;
  end
  else glitch_count1:=1;

```

```

    pos1last:=pos1;
end;

(*****)
procedure get_measurements_2;

begin
  {get velocity of motor 2}
  vel[2] := get_vel_d_i(2,ad_error);
  if ad_error then begin
    writeln('A/D conversion error has occurred -- PROGRAM ABORTED');
    halt;
  end;

  {start a/d conversion for channel 0 (motor 1), gain=1}
  port[ad_chan_addr]:=0;

  {get position of motor 2
   including check and correct for spurious position data}
  pos2:= get_position_d(2);
  if (abs(pos2-pos2last) > glitch_count2*reject) then begin
    pos2:=pos2last;
    glitch_count2:=glitch_count2+1;
  end
  else glitch_count2:=1;
  pos2last:=pos2;
end;

(*****)
procedure filter_velocities;
  {low-pass filter the measured velocities, to reduce the noise}

var i: integer;

begin
  for i:=1 to 2 do begin
    velfm1[i]:=velf[i];
    velfdotm1[i]:=velfdot[i];
    velf[i]:=bfa[1,1]*velfm1[i]+bfa[1,2]*velfdotm1[i]+bfb[1]*vel[i];
    velfdot[i]:=bfa[2,1]*velfm1[i]+bfa[2,2]*velfdotm1[i]+bfb[2]*vel[i];
  end; {for}
end;

(*****)
procedure identify_parameters;

{This procedure identifies the parameters which describe the
dynamics of the manipulator. An adaptation law based on
Lyapunov stability, passivity and the direct method is used,
according to Slotine and Li's method.}

const deg_to_rad:single = 3.1415927/180.0;

var j,k: integer; {loop indices}
    x12,c12,s12, {intermediate calculated values}
    vel1,vel2:single;

begin

```

```

    {using measured velocity}
    vel1:=vel[1]; vel2:=vel[2];
    (* {using low-pass filtered velocity}
    vel1:=velf[1]; vel2:=velf[2]; *)

    {calculate tracking errors}
    err[1] := posdf[1] - pos1;
    derr[1] := veldf[1] - vel1;
    err[2] := posdf[2] - pos2;
    derr[2] := veldf[2] - vel2;

    {calculate sine and cosine values}
    x12:= deg_to_rad*(pos1-pos2);
    c12:= cos(x12);
    s12:= sin(x12);

    {compute the reference trajectories}
    for k:=1 to 2 do begin
        velr[k]:=veldf[k]+lambda*err[k];
        accr[k]:=accdf[k]+lambda*derr[k];
    end; {for}

    {compute reference traj. errors = filtered tracking errors}
    for k:=1 to 2 do slast[k]:=s[k];
    s[1]:=vel1-velr[1];
    s[2]:=vel2-velr[2];

    {compute regression matrix}
    phi[1,1]:=accr[1];
    phi[1,3]:=c12*accr[2]+s12*vel2*velr[2];
    phi[2,2]:=accr[2];
    phi[2,3]:=c12*accr[1]-s12*vel1*velr[1];
    {regression terms for torque bias param's (4 and 5) are constants
    (initialized previously)}

    {discrete update of parameter vector with new estimates}
    for k:=1 to nparam do begin
        para[k]:=para[k]-dt*gamma*gamma_scaling[k]*
            (phi[1,k]*s[1]+phi[2,k]*s[2]);

        {limit parameter estimates to pre-defined bounds}
        if para[k]<para_min[k] then para[k]:=para_min[k]
        else if para[k]>para_max[k] then para[k]:=para_max[k];
    end;
end;

(*****)
procedure compute_control_input;

var i,j: integer;
    abs_val:single;    {absolute value of control input voltage}

begin
    input_lay[1] := pos1;

```

```

    input_lay[2] := vel[1];
    input_lay[3] := posdf[1];
    input_lay[4] := veldf[1];
    input_lay[5] := accdf[1];
    input_lay[6] := pos2;
    input_lay[7] := vel[2];
    input_lay[8] := posdf[2];
    input_lay[9] := veldf[2];
    input_lay[10] := accdf[2];

    prop_forward;

    connn[1] := (output_out[1] * divt1) - addt1;
    connn[2] := (output_out[2] * divt2) - addt2;

{calculate norm of filtered error and compensation term gain}
    norms:=sqrt(sqr(s[1])+sqr(s[2]));
    if norms>norm_sprime then g:=a*(norms-norm_sprime)
{   if norms>norm_sprime then g:=a*norms           }
        else g:=0.0;

{calculate control input values (voltages),
  corresponding to a torque command for each motor}
    for i:=1 to 2 do begin
        control[i]:=0.0;
        for j:=1 to nparam do
            control[i]:=control[i]+phi[i,j]*para[j];
        control[i]:=connn[i] + control[i]-(kv[i]+g)*s[i];
        {check for saturation}
        abs_val:=abs(control[i]);
        if abs_val > torque_limit then begin
            (*   writeln('**Caution: torque command ',i,' has saturated at',
                        torque_limit,' V');   *)
            control[i]:=(abs_val/control[i])*torque_limit;
        end; {if}
    end; {for}

{convert control voltages to 12 bit binary, and send to d/a}
    send_volt_da(1,control[1]);
    send_volt_da(2,control[2]);
end;

(*****)
procedure store_data;

(* This procedure collects data which will be plotted at
   a later time, as well as data used for debugging *)

begin
    if counter <= maxpoints then
        begin
            data[1,counter] := pos1;
            data[2,counter] := vel[1];
            data[3,counter] := posdf[1]; {pre-filtered desired path}

```

```

        data[4,counter] := veldf[1];
        data[5,counter] := accdf[1];

        data[6,counter] := pos2;
        data[7,counter] := vel[2];
        data[8,counter] := posdf[2]; {pre-filtered desired path}
        data[9,counter] := veldf[2];
        data[10,counter] := accdf[2];

        data[11,counter] := control[1];
        data[12,counter] := control[2];

    end;
end;
(*****)
procedure display_state;
{this procedure displays the state of the robot on the screen}

begin
    gotoXY(1,2);
    writeln('posdf[1]:',posdf[1]:6:2,' pos1:',pos1:6:2,'
err[1]:',err[1]:6:2);
    writeln('          vel1:',vel[1]:6:2,'
control[1]:',control[1]:6:2);
    writeln('posdf[2]:',posdf[2]:6:2,' pos2:',pos2:6:2,'
err[2]:',err[2]:6:2);
    writeln('          vel12:',vel[2]:6:2,'
control[2]:',control[2]:6:2);
end;

(*****)
procedure find_max_norm_delta_s;
{finds maximum of the norm of the change in the filtered error,
"s", between time steps}

var norm_delta_s:single;

begin
    norm_delta_s:=sqrt(sqr(s[1]-slast[1])+sqr(s[2]-slast[2]));
    if norm_delta_s > max_normds then begin
        max_normds:=norm_delta_s;
        tmax:=t
    end {if};
end;

(*****)
procedure enableinterrupts;
inline($FB/$90); {sti/nop}

procedure disableinterrupts;
inline($FA/$90); {cli/nop}

(*****)
procedure interrupt_service; interrupt;
{this procedure (ISR) performs the measurement and control tasks which
are required at the synchronized "sampling" frequency, driven by
the A/D card interrupts}

```

```

begin
  disableinterrupts;
  case icount of
    1: begin
        t:=t+dt;
        if t>=ftime then done_flag:=true;
        get_measurements_1;
      end; {case 1}

    2: begin
        compute_desired_state;
        get_measurements_2;
        filter_velocities;
        identify_parameters;
        compute_control_input;
        find_max_norm_delta_s;

        if skipcount = skip then begin
            counter:=counter+1;
            store_data;
            skipcount:=0;
          end;
        skipcount:=skipcount+1;

        icount:=0;
      end; {case 2}

  end; {case}

  icount:=icount+1;
  port[PIC_00]:= EOI; {clears the PIC}
  enableinterrupts;
end;

(*****)
{$F+}
procedure exit_handler;{$F-}

(*this procedure handles normal and abnormal termination --
  it saves all data and closes files*)

begin
  exitproc:=exitsave; {restore original exit pointer}

  disableinterrupts;
  setintvec(10,old_level2_intr); {restore old level 2 interrupt vector}
  enableinterrupts;

  if exitcode<>0 then begin
    writeln('****Run-Time Error:');
    writeln('          Error code is ',exitcode);
  end;
end;

```



```

end;

{set d/a back to 0 volts}
    send_volt_da(1,0);
    send_volt_da(2,0);

    save_data_to_file;

    create_info_file;

    writeln('** Please Disable Motor Servos Now **');
end;
(*****)
(*****)
begin {main}

    {Read the NN weights from the file}

    Assign(f2,'alarob2.out');
    Reset(f2);

    For l := 1 to first_node_num do
    Begin
        For m := 1 to num_of_input do
        Begin
            read(f2,first_hid_weight[m,1]);
        End;
        readln(f2);
    End;

    For l := 1 to second_node_num do
    Begin
        For m := 1 to first_node_num do
        Begin
            read(f2,sec_hid_weight[m,1]);
        End;
        readln(f2);
    End;

    For l := 1 to num_of_output do
    Begin
        For m := 1 to second_node_num do
        Begin
            read(f2,output_weight[m,1]);
        End;
        readln(f2);
    End;

    {set up pointer to exit handler procedure}
    exitsave:=exitproc;
    exitproc:=@exit_handler;

    {save old level 2 interrupt vector }
    getintvec(10,old_level2_intr); {interrupt type 10 = interrupt line IR2}

    {set timer interrupt vector to new interrupt service routine (ISR)}
    disableinterrupts;
    setintvec(10,@interrupt_service);
    enableinterrupts;

```

```

    {***special note: interrupts are automatically enabled when
      the "writeln" command calls DOS}

{start-up message to user}
  writeln;
  writeln('++++++ ROBOT ADAPTIVE CONTROL PROGRAM (Motor 1 & 2) ++++++');
  writeln;

{enter loop to do repeated runs}
  repeat
    initialize_interfaces;

    load_trajectory_file;

    initialize_parameters;

    get_initial_positions;
    writeln;
    writeln('Current position of Motor 1 is ',pos1:6:2,' degrees. ');
    writeln('Current position of Motor 2 is ',pos2:6:2,' degrees. ');
    writeln;

    writeln('Hit any key to start operation (after 5 second delay). ');
    key_press:=readkey;
    clrscr;

{5 second delay to give operator time}
    delay(10000);

{get the time}
    gotoXY(1,1);
    gettime(hr,min,sec,frac);
    if frac<10 then
      writeln('--> Operation starting now, at time ',hr,':',min,':',
        sec,':.0',frac)
    else writeln('--> Operation starting now, at time ',hr,':',min,':',
      sec,':.',frac);

    get_initial_positions; {in case motors were repositioned }

{start a/d conversion for channel 0 (motor 1), gain=1}
    port[ad_chan_addr]:=0;

(* {start a/d conversion for channel 1 (motor 2), gain=1}
    port[ad_chan_addr]:=1; *)

{enter loop for background tasks, while waiting for interrupts}
  repeat
    display_state;

(*      check_status(dd_port1_addr,dd_port3_addr,enabled);
*)
    bkgnd_count:=bkgnd_count+1;

    until (enabled=false) OR done_flag;

```

```

    port[ad_status_addr]:= $10; {set a/d to mode 0, clear error bit,
disable
    any further interrupts from A/D}

{stop operation if motor(s) disabled}
    if enabled=false then
        writeln('***Operation halted:  motor is disabled.');
```

```

{set d/a back to 0 volts}
    send_volt_da(1,0);
    send_volt_da(2,0);

{get the time}
    gettime(hr,min,sec,frac);
    gotoXY(1,10);
    if frac<10 then
        writeln('--> Operation complete, at time ',hr,':',min,':',
            sec, '.',frac);
    else writeln('--> Operation complete, at time ',hr,':',min,':',
            sec, '.',frac);
    writeln('    --> ',bkgnd_count,' cycles of background tasks
completed');
```

```

{prompt user for another run}
    write('Do you wish to quit -- saving data to file (y/n)? ');
    readln(choice);
    until ((choice='Y') OR (choice='y'));

{"exit_handler" procedure called by default}

    Close(f2);

end.
```

C.2.2 Filter Program Listing

```

{-----}
{ unit FILTER.PAS
{ written by:
{
{     Gabriel D. Warshaw
{ and
{     Richard Gourdeau
{
{ Revised: November 17, 1992 (GDW)
{
{ This file contain a collection of functions and
{ procedures to be used for filtering of data during
{ control of the 2 DOF direct drive robot.
{
{ The interface section shows the prototypes of the
{ functions and the procedures while the implementation
{ section contain the actual code
{
{
{ If you find a bug in this implementation, DO NOT
{ MODIFY THIS FILE Ask one of the writers to fix the bug.}
```

```

{-----}

unit filter;

interface

uses m_arith;

    type
        matrix = array[1..2] of state;

    (*****)
    procedure kalman_gains(a,b,dt:single; var kal:state);
        {calculate kalman gains to be used for kalman filtering
         of state measurements, based on first order system
         parameters and sampling interval}

    procedure filter_states(t,dt2,pos:single; xk,kal:state; var x:state);
        {calculate kalman filtered values of measured states}

    procedure predict_next_states(x:state;
        a,b,control,dt,dtdtover2:single; var xk:state);
        {calculate predicted values of next states, for use in
         kalman filtering of next set of state measurements}

    procedure pre_filter(delta_t,pole:single; var ad:matrix2;
        var bd:state);
        {calculate the coefficients for a discrete 2nd order
         filter, to be used for pre-filtering the desired
         trajectory (position) of the manipulator joints.
         The filter is critically damped, with two (identical)
         real poles placed at the value "pole" in the
         arguments.}

    procedure pre_filter3(delta_t,pole:single; var ad:matrix3;
        var bd:state3);
        {calculate the coefficients for a discrete 3rd order
         filter, to be used for pre-filtering the desired
         trajectory (position) of the manipulator joints. The
         filter has 3 (identical) real poles placed at the
         value "pole" in the arguments.}

    procedure butterworth2(delta_t,w0:single; var ad:matrix2;
        var bd:state);
        {calculate the coefficients for a discrete 2nd order
         butterworth filter, to be used for low-pass filtering
         the velocity measurements for the manipulator joints,
         to reduce the noise. The bandwidth of the filter is
         set at "w0" rad/sec.}

    procedure butterworth3(delta_t,w0:single; var ad:matrix3;
        var bd:state3);
        {calculate the coefficients for a discrete 3rd order
         butterworth filter, to be used for low-pass filtering
         the velocity measurements for the manipulator joints,
         to reduce the noise The bandwidth of the filter is

```

```
set at "w0" rad/sec.)
```

```
(*****  
implementation  
*****)
```

```
procedure kalman_gains(a,b,dt:single; var kal:state);
```

```
var  
  i: integer;  
  hphv,vv,vw:single;  
  pk,p:matrix;
```

```
begin  
  vv := 0.1;  
  vw := 0.04;  
  p[1][1] := 0.0;  
  p[1][2] := 0.0;  
  p[2][2] := 0.0;  
  
  for i := 1 to 5000 do  
    begin  
      pk[1][1] := dt*dt*b/2*vw*dt*dt*b/2+p[1][1]+(dt*(1-dt*a/2))*p[1][2]+  
        (p[1][2]+(dt*(1-dt*a/2))*p[2][2])*(dt*(1-dt*a/2));  
      pk[1][2] := dt*dt*b/2*vw*dt*b+(p[1][2]+(dt*(1-dt*a/2))  
* p[2][2])*(1-dt*a);  
      pk[2][2] := dt*b*vw*dt*b+((1-dt*a)*p[2][2])*(1-dt*a);  
      hphv := 1.0/(pk[1][1]+vv);  
      p[1][1] := pk[1][1]-pk[1][1]*hphv*pk[1][1];  
      p[1][2] := pk[1][2]-pk[1][1]*hphv*pk[1][2];  
      p[2][2] := pk[2][2]-pk[1][2]*hphv*pk[1][2];  
    end;  
    kal[1] := p[1][1]/vv;  
    kal[2] := p[1][2]/vv;  
  end;
```

```
(*****  
procedure filter_states(t,dt2,pos:single; xk,kal:state;  
var x:state);
```

```
var  
  perr:single; {prediction error}
```

```
begin  
  if t < dt2 then xk[1] := pos;  
  perr := pos-xk[1];  
  x[1] := xk[1]+kal[1]*perr;  
  x[2] := xk[2]+kal[2]*perr;  
end;
```

```
(*****  
procedure predict_next_states(x:state; a,b,control,  
  dt,dtdtover2:single; var xk:state);  
begin  
  xk[1] := x[1]+(dt-dtdtover2*a)*x[2]+dtdtover2*b*control;  
  xk[2] := (1.0-dt*a)*x[2]+b*dt*control;
```

```

end;

(*****)
procedure pre_filter(delta_t,pole:single; var ad:matrix2;
var bd:state);

const iterations=20;

var i,j: integer;
    factorial,alpha,beta,omega,tpower: single;
    amatrix,atemp,atemp2,i2x2,btemp:matrix2;
    bvector:state;

begin
    beta:= -2.0*pole;
    omega:=-pole;
    alpha:=omega*omega;

    i2x2[1,1]:=1.0;
    i2x2[1,2]:=0.0;
    i2x2[2,1]:=0.0;
    i2x2[2,2]:=1.0;

    amatrix[1,1]:=0.0;
    amatrix[1,2]:=1.0;
    amatrix[2,1]:=-alpha;
    amatrix[2,2]:=-beta;

    bvector[1]:=0.0;
    bvector[2]:=alpha;

    matrix_const(1.0,i2x2,ad);      {set ad = I matrix}
    matrix_const(1.0,i2x2,atemp);   {set atemp = I matrix}
    factorial:=1.0;
    tpower:=1.0;

    {calculate Ad matrix}
    for i:=1 to iterations do begin
        matrix_mult(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const(tpower/factorial,atemp,atemp2);
        matrix_add(ad,atemp2,ad);
    end; {for}

    matrix_const(delta_t,i2x2,btemp); {set btemp=
I*delta_t matrix}
    matrix_const(1.0,i2x2,atemp); {set atemp = I matrix}
    factorial:=1.0;
    tpower:=delta_t;

    {calculate Bd matrix}
    for i:=2 to iterations+1 do begin
        matrix_mult(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const(tpower/factorial,atemp,atemp2);
        matrix_add(btemp,atemp2,btemp);
    end; {for}

```

```

        end; {for}
        vector_mult(btemp,bvector,bd);
    end;

    (*****)
    procedure pre_filter3(delta_t,pole:single; var ad:matrix3; var bd:state3);

    const iterations=20;

    var i,j: integer;
        factorial,a,b,c,omega,tpower: single;
        amatrix,atemp,atemp2,i3x3,btemp:matrix3;
        bvector:state3;

    begin
        a:=-3.0*pole;
        b:=3.0*pole*pole;
        c:=-pole*pole*pole;

        for i:=1 to 3 do for j:=1 to 3 do i3x3[i,j]:=0.0;
        for i:=1 to 3 do i3x3[i,i]:=1.0;

        amatrix[1,1]:=0.0; amatrix[1,2]:=1.0; amatrix[1,3]:=0.0;
        amatrix[2,1]:=0.0; amatrix[2,2]:=0.0; amatrix[2,3]:=1.0;
        amatrix[3,1]:=-c; amatrix[3,2]:=-b; amatrix[3,3]:=-a;

        bvector[1]:=0.0;
        bvector[2]:=0.0;
        bvector[3]:=c;

        matrix_const3(1.0,i3x3,ad);      {set ad = I matrix}
        matrix_const3(1.0,i3x3,atemp);   {set atemp = I matrix}
        factorial:=1.0;
        tpower:=1.0;

        {calculate Ad matrix}
        for i:=1 to iterations do begin
            matrix_mult3(atemp,amatrix,atemp);
            factorial:=i*factorial;
            tpower:=tpower*delta_t;
            matrix_const3(tpower/factorial,atemp,atemp2);
            matrix_add3(ad,atemp2,ad);
        end; {for}

        matrix_const3(delta_t,i3x3,btemp); {set btemp=
I*delta_t matrix}
        matrix_const3(1.0,i3x3,atemp); {set atemp = I matrix}
        factorial:=1.0;
        tpower:=delta_t;

        {calculate Bd matrix}
        for i:=2 to iterations+1 do begin
            matrix_mult3(atemp,amatrix,atemp);
            factorial:=i*factorial;
            tpower:=tpower*delta_t;
            matrix_const3(tpower/factorial,atemp,atemp2);
            matrix_add3(btemp,atemp2,btemp);
        end; {for}
    end;

```

```

    vector_mult3(btemp,bvector,bd);
end;

(*****)
procedure butterworth2(delta_t,w0:single; var ad:matrix2; var bd:state);
const iterations=6;
var i,j: integer;
    factorial,alpha,beta,tpower: single;
    amatrix,atemp,atemp2,i2x2,btemp:matrix2;
    bvector:state;
begin
    beta:=sqrt(2.0)*w0;
    alpha:=w0*w0;

    i2x2[1,1]:=1.0;
    i2x2[1,2]:=0.0;
    i2x2[2,1]:=0.0;
    i2x2[2,2]:=1.0;

    amatrix[1,1]:=0.0;
    amatrix[1,2]:=1.0;
    amatrix[2,1]:=-alpha;
    amatrix[2,2]:=-beta;

    bvector[1]:=0.0;
    bvector[2]:=alpha;

    matrix_const(1.0,i2x2,ad);      {set ad = I matrix}
    matrix_const(1.0,i2x2,atemp);   {set atemp = I matrix}
    factorial:=1.0;
    tpower:=1.0;

    {calculate Ad matrix}
    for i:=1 to iterations do begin
        matrix_mult(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const(tpower/factorial,atemp,atemp2);
        matrix_add(ad,atemp2,ad);
    end; {for}

    matrix_const(delta_t,i2x2,btemp); {set btemp=
I*delta_t matrix}
    matrix_const(1.0,i2x2,atemp);   {set atemp = I matrix}
    factorial:=1.0;
    tpower:=delta_t;

    {calculate Bd matrix}
    for i:=2 to iterations+1 do begin
        matrix_mult(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const(tpower/factorial,atemp,atemp2);
        matrix_add(btemp,atemp2,btemp);
    end; {for}

```



```

    vector_mult(btemp,bvector,bd);
end;

(*****)
procedure butterworth3(delta_t,w0:single; var ad:matrix3; var bd:state3);

const iterations=20;

var i,j: integer;
    factorial,a,b,c,tpower: single;
    amatrix,atemp,atemp2,i3x3,btemp:matrix3;
    bvector:state3;

begin
    a:=2.0*w0;
    b:=2.0*w0*w0;
    c:=w0*w0*w0;

    for i:=1 to 3 do for j:=1 to 3 do i3x3[i,j]:=0.0;
    for i:=1 to 3 do i3x3[i,i]:=1.0;

    amatrix[1,1]:=0.0; amatrix[1,2]:=1.0; amatrix[1,3]:=0.0;
    amatrix[2,1]:=0.0; amatrix[2,2]:=0.0; amatrix[2,3]:=1.0;
    amatrix[3,1]:=-c; amatrix[3,2]:=-b; amatrix[3,3]:=-a;

    bvector[1]:=0.0;
    bvector[2]:=0.0;
    bvector[3]:=c;

    matrix_const3(1.0,i3x3,ad);      {set ad = I matrix}
    matrix_const3(1.0,i3x3,atemp);   {set atemp = I matrix}
    factorial:=1.0;
    tpower:=1.0;

    {calculate Ad matrix}
    for i:=1 to iterations do begin
        matrix_mult3(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const3(tpower/factorial,atemp,atemp2);
        matrix_add3(ad,atemp2,ad);
    end; {for}

    matrix_const3(delta_t,i3x3,btemp); {set btemp=
I*delta_t matrix}
    matrix_const3(1.0,i3x3,atemp); {set atemp = I matrix}
    factorial:=1.0;
    tpower:=delta_t;

    {calculate Bd matrix}
    for i:=2 to iterations+1 do begin
        matrix_mult3(atemp,amatrix,atemp);
        factorial:=i*factorial;
        tpower:=tpower*delta_t;
        matrix_const3(tpower/factorial,atemp,atemp2);
        matrix_add3(btemp,atemp2,btemp);
    end; {for}
    vector_mult3(btemp,bvector,bd);

```

end;

end.

C.2.3 Databcard Program Listing

```

{-----}
{ unit DATACARD.PAS
{ written by:
{       Gabriel D. Warshaw
{       and
{       Richard Gourdeau
{
{ Revised: December 16, 1992 (G.D.W.) --
{ reversed polarity for DA #2 output
{
{ This file contain a collection of functions and
{ procedures to be used with the DATA TRANSLATION CARD
{ to control the 2 DOF robot
{
{ The interface section shows the prototypes of the
{ functions and the procedures while the implementation
{ section contain the actual code
{
{ If you find a bug in this implementation, DO NOT
{ MODIFY THIS FILE Ask one of the writers to fix the bug.}
{-----}
unit databcard;

interface

function timer_freq(freq : integer) : boolean;
    {return True if an error occurred}

function get_dd_word(motor : byte) : integer;
    {get 12 bits value from digital input composed of two ports}

function get_dd_byte(motor : byte) : byte;
    {get 8 bits value from digital port}

function get_position_d(motor : byte) : single;
    {get motor 1 or 2 position in degrees}

function get_position_r(motor : byte) : single;
    {get motor 1 or 2 position in radians}

procedure send_volt_da(da_number: byte; value : single);
    {set output da_number to value volts}
    {corrects for reversed direction of motor #2 of
    direct drive robot}

function get_ad_word(motor : byte; var error:boolean):
integer;
    {get motor 1 or 2 velocity in binary form}
    {set error to true if a conversion error occurred}

```

```

function get_ad_word_i(var error:boolean): integer;
    {get motor 1 or 2 velocity in binary form
(channel selection must be done beforehand, outside of
this function)}
    {set error to true if a conversion error occurred}
    {uses interrupt to signal conversion complete}

function get_vel_d(motor : byte; var error:boolean): single;
    {get motor 1 or 2 velocity in degree/sec}
    {set error to true if a conversion error occurred}

function get_vel_d_i(motor : byte; var error:boolean): single;
    {get motor 1 or 2 velocity in degree/sec}
    {set error to true if a conversion error occurred}
    {uses interrupt-driven A/D}

function get_vel_r(motor : byte; var error:boolean): single;
    {get motor 1 or 2 velocity in radian/sec}
    {set error to true if a conversion error occurred}

```

implementation

```

{*****}
function timer_freq(freq : integer) : boolean;

const
    {timer address}
    timer_addr : word = $21F;

    {valid integer frequencies}
    fchoice : array[1..26] of integer = (6000,600,60,6,
                                           3000,300,30,3,
                                           2000,200,20,2,
                                           1500,150,15,
                                           1200,120,12,
                                           1000,100,10,1,
                                           5000,500,50,5);

    {corresponding frequency codes}
    tmrctr : array[1..26] of byte = ($09,$0A,$0B,$0C,
                                       $12,$13,$14,$15,
                                       $1A,$1B,$1C,$1D,
                                       $22,$23,$24,
                                       $2A,$2B,$2C,
                                       $32,$33,$34,$35,
                                       $39,$3A,$3B,$3C);

var
    i : integer;

begin
    i := 0;
    timer_freq := true;
    repeat

```

[illegible]

```

begin
  get_dd_word := port[dd_port_addr[motor,1]]
                + (port[dd_port_addr[motor,2]]
and 1o4bits) shl 8;
end;

{*****}
function get_dd_byte(motor : byte) : byte;

const
  {digital inputs port addresses for values}
  dd_port_addr : array[1..4] of integer =
    ($229,$22A,$22B,$22C);

begin
  get_dd_byte := port[dd_port_addr[motor]];
end;

{*****}
procedure send_volt_da(da_number: byte; value : single);

const
  {port addresses of low and high byte of d/a}
  da_addr : array[1..2,1..2] of integer = (($21A,$21B),
                                              ($21C,$21D));

  {12 bit binary value for 0 volts, based on -5 to +5 range}
  zerovolts_bin : integer = $800;

  da_cvrt : single = 4096.0/10.0;

var
  v_bin : integer;

begin
  if da_number=2 then value:=-value;
{reverse direction for motor #2}
  v_bin := zerovolts_bin + round(value*da_cvrt);
  port[da_addr[da_number,1]] := lo(v_bin);
  port[da_addr[da_number,2]] := hi(v_bin);
end;

{*****}
function get_ad_word(motor : byte; var error:boolean):
  integer;

const
  bit6 = $40;
  bit7 = $80;
  ad_loadaddr = $21A;
  ad_hiaddr = $21B;
  ad_status_addr = $218;

```

```

    ad_chan_addr := $219;

var    status:byte;

begin
    port[ad_chan_addr] := motor-1;
    {switch a/d to channel motor-1, gain=1}
    repeat status := port[ad_status_addr]
    until (status AND bit7) = bit7;
    if(status and bit6) = bit6 then error := true;
    get_ad_word := port[ad_loaddr] + port[ad_hiaddr] shl 8;
end;

{*****}
function get_ad_word_i(var error:boolean): integer;

const
    bit6 = $40;
    bit7 = $80;
    ad_loaddr = $21A;
    ad_hiaddr = $21B;
    ad_status_addr := $218;
    ad_chan_addr := $219;

var    status:byte;

begin
    status:=port[ad_status_addr];
    if(status and bit6) = bit6 then error := true;
    get_ad_word_i := port[ad_loaddr] + port[ad_hiaddr] shl 8;
end;

{*****}
function get_vel_d(motor : byte; var error:boolean): single;

const
    resol = 10.0/4096.0;
    ktach : array[1..2] of single = (328.0,244.0); {deg/s*volt}

begin
    get_vel_d :=
        (get_ad_word(motor,error)*resol-5.0)*ktach[motor];
end;

{*****}
function get_vel_d_i(motor : byte; var error:boolean): single;

const
    resol = 10.0/4096.0;
    ktach : array[1..2] of single = (328.0,244.0); {deg/s*volt}

begin
    get_vel_d_i :=
        (get_ad_word_i(error)*resol-5.0)*ktach[motor];
end;

```

```

end;

{*****}
function get_vel_r(motor : byte; var error:boolean): single;

const
  resol = 10.0/4096.0;
  ktach : array[1..2] of single = (5.7247,4.2586);
{rad/s*volt}

begin
  get_vel_r :=
  (get_ad_word(motor,error)*resol-5.0)*ktach[motor];
end;

end.

```

C.2.4 Marith Program Listing

```

{-----}
{ unit M_ARITH.PAS
{ written by:
{           Gabriel D. Warshaw
{
{ Revised: Oct. 5, 1992
{
{ This file contain a collection of functions and
{ procedures to be used for performing a range of simple
{ matrix computations.
{
{ The interface section shows the prototypes of the
{ functions and the procedures while the implementation
{ section contain the actual code.
{
{ If you find a bug in this implementation, DO NOT
{ MODIFY THIS FILE Ask one of the writers to fix the bug.
{-----}

unit m_arith;

interface

  type
    state = array[1..2] of single;
    matrix2 = array[1..2,1..2] of single;
    state3 = array[1..3] of single;
    matrix3 = array[1..3,1..3] of single;

  (*****)
  procedure matrix_add(a,b:matrix2; var c:matrix2);
    {add two 2x2 matrices}

  procedure matrix_mult(a,b:matrix2; var c:matrix2);

```

```

    {multiply two 2x2 matrices}

procedure vector_mult(a:matrix2; b:state; var c:state);
    {multiply a 2x2 matrix by a 2x1 vector}

procedure matrix_const(a:single; b:matrix2; var c:matrix2);
    {multiply a 2x2 matrix by a scalar constant}

procedure matrix_add3(a,b:matrix3; var c:matrix3);
    {add two 3x3 matrices}

procedure matrix_mult3(a,b:matrix3; var c:matrix3);
    {multiply two 3x3 matrices}

procedure vector_mult3(a:matrix3; b:state3; var c:state3);
    {multiply a 3x3 matrix by a 3x1 vector}

procedure matrix_const3(a:single; b:matrix3; var c:matrix3);
    {multiply a 3x3 matrix by a scalar constant}

(*****
implementation
*****)
procedure matrix_add(a,b:matrix2; var c:matrix2);
    {add two 2x2 matrices}

var i,j: integer;

begin
    for i:=1 to 2 do
        for j:=1 to 2 do
            c[i,j]:=a[i,j]+b[i,j];
        end;
    end;

(*****
procedure matrix_mult(a,b:matrix2; var c:matrix2);
    {multiply two 2x2 matrices}

var i,j,k: integer;
    sum:single;
begin
    for i:=1 to 2 do
        for j:=1 to 2 do begin
            sum:=0.0;
            for k:=1 to 2 do
                sum:=sum+a[i,k]*b[k,j];
            end;
            c[i,j]:=sum;
        end;
    end;

(*****
procedure vector_mult(a:matrix2; b:state; var c:state);
    {multiply a 2x2 matrix by a 2x1 vector}

var i,k: integer;
    sum:single;
begin
    for i:=1 to 2 do begin

```



```

        sum:=0.0;
        for k:=1 to 2 do
            sum:=sum+a[i,k]*b[k];
        c[i]:=sum;
    end;
end;

(*****)
procedure matrix_const(a:single; b:matrix2; var c:matrix2);
    {multiply a 2x2 matrix by a scalar constant}

var i,j: integer;

begin
    for i:=1 to 2 do
        for j:=1 to 2 do
            c[i,j]:=a*b[i,j];
        end;
    end;

(*****)
procedure matrix_add3(a,b:matrix3; var c:matrix3);
    {add two 3x3 matrices}

var i,j: integer;

begin
    for i:=1 to 3 do
        for j:=1 to 3 do
            c[i,j]:=a[i,j]+b[i,j];
        end;
    end;

(*****)
procedure matrix_mult3(a,b:matrix3; var c:matrix3);
    {multiply two 3x3 matrices}

var i,j,k: integer;
    sum:single;
begin
    for i:=1 to 3 do
        for j:=1 to 3 do begin
            sum:=0.0;
            for k:=1 to 3 do
                sum:=sum+a[i,k]*b[k,j];
            c[i,j]:=sum;
            end;
        end;
    end;

(*****)
procedure vector_mult3(a:matrix3; b:state3; var c:state3);
    {multiply a 3x3 matrix by a 3x1 vector}

var i,k: integer;
    sum:single;
begin
    for i:=1 to 3 do begin
        sum:=0.0;
        for k:=1 to 3 do
            sum:=sum+a[i,k]*b[k];
        c[i]:=sum;
    end;
end;

```

```

    end;
end;

(*****)
procedure matrix_const3(a:single; b:matrix3; var c:matrix3);
    {multiply a 3x3 matrix by a scalar constant}

var i,j: integer;

begin
    for i:=1 to 3 do
        for j:=1 to 3 do
            c[i,j]:=a*b[i,j];
        end;
    end;
end.

```

C.3 Training The Neural Network

```

program exp2tra3;
{*****}
* Name : Zaryab Hamavand *
* Date : Jan 1994 *
* *
* This program will use the experiment data *
* files to train the Neural Network. *
{*****}
{$N+}
{$R+} {Enable range checking}
uses CRT;
Const
    num_of_input    = 10;
    num_of_output   = 2;
    first_node_num  = 40;
    second_node_num = 40;
    etha            = 0.05;

    addt1           = 6.0;
    divt1           = 12.0;
    addt2           = 2.0;
    divt2           = 5.0;
Var
    f11,f1,f2,f3,f4,f5,f6 : text;
    ii,jj,kk,m,l,ds,filte,dd : Integer;
    Count1 : LongInt;
    first_hid_weight : array[1..num_of_input, 1..first_node_num] of Double;
    sec_hid_weight : array[1..first_node_num, 1..second_node_num] of Double;
    output_weight : array[1..second_node_num, 1..num_of_output] of Double;

    first_hid_out : array[1..first_node_num] of Double;
    first_hid_err : array[1..first_node_num] of Double;

    sec_hid_out : array[1..second_node_num] of Double;
    sec_hid_err : array[1..second_node_num] of Double;

```

```

output_out : array[1..num_of_output] of Double;
output_err : array[1..num_of_output] of Double;
target : array[1..num_of_output] of Double;

input_lay : array[1..num_of_input] of Double;
data_set : array[1..749, 1..12] of single;

{*****
* This procedure will propagate the input vector through the      *
* network and will produce output. This network consists of two hidden *
* layers.                                                         *
*****}

procedure prop_forward;

Var
  i, j : Integer;
  sum : Double;

Begin
  For i := 1 To first_node_num Do
    Begin
      sum := 0.0;
      For j := 1 to num_of_input Do
        Begin
          sum := sum + (first_hid_weight[j,i] * input_lay[j]);
        End;
        first_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
      End;

      For i := 1 To second_node_num Do
        Begin
          sum := 0.0;
          For j := 1 to first_node_num Do
            Begin
              sum := sum + (first_hid_out[j] * sec_hid_weight[j,i]);
            End;
            sec_hid_out[i] := 1.0 / (1.0 + exp(-1 * sum));
          End;

          For i := 1 To num_of_output Do
            Begin
              sum := 0.0;
              For j := 1 to second_node_num Do
                Begin
                  sum := sum + (sec_hid_out[j] * output_weight[j,i]);
                End;
                output_out[i] := 1.0 / (1.0 + exp(-1 * sum));
              End;
            End;
          End;

        End;

      {*****
      * This procedure will calculate the error on each layer's nodes, *
      * according to the error on the output.                         *
      *****}

```

```
Procedure compute_errors;
```

```
Var
  i, j : Integer;
```

```
Begin
```

```
  For i := 1 to num_of_output do
```

```
  Begin
```

```
    output_err[i] := output_out[i] * (1 - output_out[i]) *
                      (target[i] - output_out[i]);
```

```
  End;
```

```
  For i := 1 to second_node_num do
```

```
  Begin
```

```
    sec_hid_err[i] := 0.0;
```

```
    For j := 1 to num_of_output do
```

```
    Begin
```

```
      sec_hid_err[i] := sec_hid_err[i] + ( output_err[j] *
                                             output_weight[i,j]);
```

```
    End;
```

```
    sec_hid_err[i] := sec_hid_err[i] * sec_hid_out[i] *
                      (1 - sec_hid_out[i]);
```

```
  End;
```

```
  For i := 1 to first_node_num do
```

```
  Begin
```

```
    first_hid_err[i] := 0.0;
```

```
    For j := 1 to second_node_num do
```

```
    Begin
```

```
      first_hid_err[i] := first_hid_err[i] + ( sec_hid_err[j] *
                                             sec_hid_weight[i,j]);
```

```
    End;
```

```
    first_hid_err[i] := first_hid_err[i] * first_hid_out[i] *
                      (1 - first_hid_out[i]);
```

```
  End;
```

```
End;
```

```
{*****
* This procedure will update the weights of each layer according to *
* the error on each node. *
* The 1st hidden layer will be updated, then the 2nd hidden layer, *
* and finally the output layer. *
*****}
```

```
Procedure update_weights;
```

```
Var
  i, j : Integer;
```

```
Begin
```

```

For i := 1 to first_node_num Do
Begin
  For j := 1 to num_of_input Do
  Begin
    first_hid_weight[j,i] := first_hid_weight[j,i] + ( etha *
                                                         first_hid_err[i] * input_lay[j]);
  End;
End;

For i := 1 to second_node_num Do
Begin
  For j := 1 to first_node_num Do
  Begin
    sec_hid_weight[j,i] := sec_hid_weight[j,i] + ( etha *
                                                         sec_hid_err[i] * sec_hid_out[j]);
  End;
End;

For i := 1 to num_of_output Do
Begin
  For j := 1 to second_node_num Do
  Begin
    output_weight[j,i] := output_weight[j,i] + ( etha *
                                                         output_err[i] * sec_hid_out[j]);
  End;
End;

End;

{*****
* This procedure loads the chosen data file to the*
*program. In addition, this data will be used to *
*train the Neural Network.                        *
*****}
Procedure train_file(var m1 : text);

Begin
  For kk := 1 to 749 do
  Begin
    For ii := 1 to 12 do
    Begin
      Read(m1,data_set[kk,ii]);
    End;
    Readln(m1);
  End;

  Writeln(count1);

  For ii := 0 to 3000 do
  Begin
    ds := Random(748) + 1 ;

    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
  End;
End;

```

```

        input_lay[5] := data_set[ds,5];
        input_lay[6] := data_set[ds,6];
        input_lay[7] := data_set[ds,7];
        input_lay[8] := data_set[ds,8];
        input_lay[9] := data_set[ds,9];
        input_lay[10] := data_set[ds,10];

        target[1] := (data_set[ds,11] + addt1) / divt1;
        target[2] := (data_set[ds,12] + addt2) / divt2;

        prop_forward;
        compute_errors;
        update_weights;

    End;
    count1 := count1 + 1;
End;

{=====
=                                     Main Program!!!!                                     =
=====}
Begin
    Randomize;
    Assign(f1,'alarob1.out');
    Reset(f1);
    Assign(f2,'alarob2.out');
    Rewrite(f2);
    Assign(f3,'alarob3.out');
    Rewrite(f3);
    Assign(f4,'alarob4.out');
    Rewrite(f4);
    Assign(f5,'alarob5.out');
    Rewrite(f5);
    Assign(f6,'alarob6.out');
    Rewrite(f6);
    For l := 1 to first_node_num do
    Begin
        For m := 1 to num_of_input do
        Begin
            read(f1,first_hid_weight[m,1]);
        End;
        readln(f1);
    End;
    For l := 1 to second_node_num do
    Begin
        For m := 1 to first_node_num do
        Begin
            read(f1,sec_hid_weight[m,1]);
        End;
        readln(f1);
    End;
    For l := 1 to num_of_output do
    Begin
        For m := 1 to second_node_num do
        Begin
            read(f1,output_weight[m,1]);
        End;
    End;
    End;

```

```
        readln(f1);
End;
count1 := 0;
For dd := 0 to 29 do
Begin
    filte := dd;
    Case filte of
        0 : begin
            Assign(f11,'z1.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        1 : begin
            Assign(f11,'z2.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        2 : begin
            Assign(f11,'z3.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        3 : begin
            Assign(f11,'z4.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        4 : begin
            Assign(f11,'z5.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        5 : begin
            Assign(f11,'z6.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        6 : begin
            Assign(f11,'z7.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        7 : begin
            Assign(f11,'z8.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
        End;
        8 : begin
```

```
        Assign(f11,'z9.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
9 : begin
    Assign(f11,'z10.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
10 : begin
    Assign(f11,'z11.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
11 : begin
    Assign(f11,'z12.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
12 : begin
    Assign(f11,'z13.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
13 : begin
    Assign(f11,'z14.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
14 : begin
    Assign(f11,'z15.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
15 : begin
    Assign(f11,'z16.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
16 : begin
    Assign(f11,'z17.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
17 : begin
    Assign(f11,'z18.dat');
    Reset(f11);
```



```
        train_file(f11);
        close(f11);
    End;
18 : begin
        Assign(f11,'z19.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
19 : begin
        Assign(f11,'z20.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
20 : begin
        Assign(f11,'z21.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
21 : begin
        Assign(f11,'z22.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
22 : begin
        Assign(f11,'z23.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
23 : begin
        Assign(f11,'z24.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
24 : begin
        Assign(f11,'z25.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
25 : begin
        Assign(f11,'z26.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
26 : begin
        Assign(f11,'z27.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
```

```

        End;
27 : begin
        Assign(f11,'z28.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
28 : begin
        Assign(f11,'z29.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
29 : begin
        Assign(f11,'z30.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
    End;
End;
While not KeyPressed do
Begin
    filte := Random(29);
    Case filte of
        0 : begin
            Assign(f11,'z1.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
            End;
        1 : begin
            Assign(f11,'z2.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
            End;
        2 : begin
            Assign(f11,'z3.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
            End;
        3 : begin
            Assign(f11,'z4.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
            End;
        4 : begin
            Assign(f11,'z5.dat');
            Reset(f11);
            train_file(f11);
            close(f11);
            End;
        5 : begin

```

```
        Assign(f11,'z6.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
6 : begin
    Assign(f11,'z7.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
7 : begin
    Assign(f11,'z8.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
8 : begin
    Assign(f11,'z9.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
9 : begin
    Assign(f11,'z10.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
10 : begin
    Assign(f11,'z11.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
11 : begin
    Assign(f11,'z12.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
12 : begin
    Assign(f11,'z13.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
13 : begin
    Assign(f11,'z14.dat');
    Reset(f11);
    train_file(f11);
    close(f11);
    End;
14 : begin
    Assign(f11,'z15.dat');
    Reset(f11);
```

```
        train_file(f11);
        close(f11);
    End;
15 : begin
        Assign(f11,'z16.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
16 : begin
        Assign(f11,'z17.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
17 : begin
        Assign(f11,'z18.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
18 : begin
        Assign(f11,'z19.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
19 : begin
        Assign(f11,'z20.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
20 : begin
        Assign(f11,'z21.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
21 : begin
        Assign(f11,'z22.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
22 : begin
        Assign(f11,'z23.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
    End;
23 : begin
        Assign(f11,'z24.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
```

```

        End;
24 : begin
        Assign(f11,'z25.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
25 : begin
        Assign(f11,'z26.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
26 : begin
        Assign(f11,'z27.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
27 : begin
        Assign(f11,'z28.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
28 : begin
        Assign(f11,'z29.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
29 : begin
        Assign(f11,'z30.dat');
        Reset(f11);
        train_file(f11);
        close(f11);
        End;
    End;
End;
For jj := 1 to 200 do
Begin
    ds := Random(748) + 1 ;
    input_lay[1] := data_set[ds,1];
    input_lay[2] := data_set[ds,2];
    input_lay[3] := data_set[ds,3];
    input_lay[4] := data_set[ds,4];
    input_lay[5] := data_set[ds,5];
    input_lay[6] := data_set[ds,6];
    input_lay[7] := data_set[ds,7];
    input_lay[8] := data_set[ds,8];
    input_lay[9] := data_set[ds,9];
    input_lay[10] := data_set[ds,10];
    target[1] := (data_set[ds,11] + addt1) / divt1;
    target[2] := (data_set[ds,12] + addt2) / divt2;
    prop_forward;
    compute_errors;

```

```

        update_weights;
        writeln(f6,target[1],', ',output_out[1],', ',target[2],', ',
            output_out[2]);
        For l := 1 to num_of_output do
        Begin
            For m := 1 to 4 do
            Begin
                write(f5,output_weight[m,l],', ');
            End;
        End;
        writeln(f5);
        For l := 1 to 4 do
        Begin
            For m := 1 to 2 do
            Begin
                write(f3,first_hid_weight[m,l],', ');
            End;
        End;
        writeln(f3);
        For l := 1 to 4 do
        Begin
            For m := 1 to 2 do
            Begin
                write(f4,sec_hid_weight[m,l],', ');
            End;
        End;
        writeln(f4);
    End;
    For l := 1 to first_node_num do
    Begin
        For m := 1 to num_of_input do
        Begin
            write(f2,first_hid_weight[m,l],', ');
        End;
        writeln(f2);
    End;
    For l := 1 to second_node_num do
    Begin
        For m := 1 to first_node_num do
        Begin
            write(f2,sec_hid_weight[m,l],', ');
        End;
        writeln(f2);
    End;
    For l := 1 to num_of_output do
    Begin
        For m := 1 to second_node_num do
        Begin
            write(f2,output_weight[m,l],', ');
        End;
        writeln(f2);
    End;
    Close(f2);
    Close(f3);
    Close(f4);
    Close(f5);
    Close(f6);
end.

```

END

3 1 - 0 5 - 9 5

FIN