ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/291822057

Implementation and Testing of Variable-Time-Delays-Robust Telemanipulation Through Master State Prediction

Thesis · September 1999

DOI: 10.13140/RG.2.1.3887.2081

CITATIONS
0

READS
153

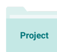
1 author:

Kyriakos M. Deliparaschos
Cyprus University of Technology

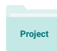
70 PUBLICATIONS **595** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Horizon scanning - Future of UGVs View project



RautoR: Railway Autonomy and Reliability View project

All content following this page was uploaded by Kyriakos M. Deliparaschos on 25 January 2016.

The user has requested enhancement of the downloaded file.

SCHOOL OF ENGINEERING AND MANUFACTURE

MSc. MECHATRONICS

DE MONTFORT UNIVERSITY
LEICESTER
98/99

IMPLEMENTATION AND TESTING OF VARIABLE-TIME-DELAYS-ROBUST TELEMANIPULATION THROUGH MASTER STATE PREDICTION

PROJECT REPORT

STUDENT: KYRIAKOS DELIPARASCHOS

SUPERVISORS: PROF S. G. TZAFESTAS
PROF P. R. MOORE

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my first supervisor Professor S. G. Tzafestas of Intelligent, Robotics & Automation Lab., Dept. of Electrical & Computer Eng., National Technical University of Athens (NTUA), for supplying me with this Master's thesis and for his kind guidance. I would like to thank, Ph.D. candidate, P. A. Prokopiou, of Intelligent, Robotics & Automation Lab., Dept. of Electrical & Computer Engineering, (NTUA), for his continued help and support to my questions and queries.

I would also like to acknowledge my appreciation to my second supervisor, Professor P. R. Moore of Mechanical & Manufacturing Eng. Dept. at De Montfort University, Gateway, Leicester, for his help. Moreover the attention and guidance of my course leader, Mr H. Pancholi of Mechanical & Manufacturing Eng. Dept. at De Montfort University, Gateway, Leicester, is sincerely appreciated.

Finally, I'm grateful to my parents, Michael and Catherine, my grandmother, Anastasia Kalliontzi, and my girlfriend, Helen Karligiotou, for their love and encouragement during the preparation of this Master's thesis.

ABSTRACT

This project is based on the **implementation and testing of variable-time-delays-robust telemanipulation through master state prediction** by using high level languages (C++) and Matlab software package.

Time delay compensation in teleoperation can be achieved by predicting the human arm position and force (effectively the master state). The method is based on the prediction of the master state (position x_m and force f_m) only, which can be much more simple and accurate than predicting the slave and the remote environment, and incorporates this in a stable force-feedback scheme.

The telemanipulation method was split into its fundamental elements and implemented as a number of functions. Furthermore two different methods (*interpolation, curve fitting theories*) for implementing the predictor model were developed and tested.

Finally the telemanipulation method was simulated (using sinusoidal inputs as the neural input) several times and the results produced, were evaluated. Due to time limitations and programming difficulties, the programming of the force feedback joystick (role of master robot) was not included.

CONTENTS

ACKNOWLEDGEMENTS I

ABSTRACT II

CONTENTS III

LIST OF FIGURES VI

LIST OF TABLES VIII

INTRODUCTION 1

1.1 BRIEF INTRODUCTION TO TELEOPERATION 1
1.2 EARLY HISTORY AND APPLICATIONS 6
1.3 SCOPE OF THE PROJECT 11

CONTROL OF TELEOPERATION SYSTEMS 12

2.1 INTRODUCTION 12
2.2 CLASSIFICATION OF CONTROL ARCHITECTURES 12
2.2.1 Position to Position Loop 13
2.2.2 Position – Force Loop 13
2.2.3 Position – Force Loop 15
2.2.4 Force - Force Loop 15
2.3 MASTER-SLAVE SYSTEM REPRESENTATION BY TWO-TERMINAL-PAIR NETWORK 15
2.4 EVALUATION OF STABILITY BASED ON PASSIVITY OF THE SYSTEM 17
2.5 PERFORMANCE EVALUATION OF TELEOPERATOR SYSTEMS 18
2.6 HUMAN REACTION AND MODELLING 19
2.7 ENVIRONMENT MODELLING 21
2.8 THE YOKOKOHJI AND YOSHIKAWA CONTROL LAW 22
2.9 OTHER CONTROL ARCHITECTURES 26
2.10 TRANSMISSION TIME-DELAYS 26
2.10.1 Scattering Theory 27
2.10.2 The Wave Variable or Energy Approach 28
2.11 SEMIAUTONOMOUS CONTROL 28

TIME-DELAYS-ROBUST TELEMANIPULATION THROUGH MASTER STATE PREDICTION 30

3.1 INTRODUCTION 30
3.2 THE CONCEPT OF PREDICTING THE MASTER STATE 32
3.3 MODEL-BASED PREDICTION 34
3.4 TRAJECTORY EXTRAPOLATING PREDICTION 37
3.5 THE ENHANCED YOKOKOHJI AND YOSHIKAWA SCHEME 38

IMPLEMENTATION OF PROPOSED METHOD 41

4.1 INTRODUCTION 41
4.2 PROGRAM EXPLANATION 41
4.3 INTERPOLATION 42
4.3.1 Interpolation Theory 42
4.3.2 Interpolation Function 45
4.3.3 Pseudo Code for Function, lagrance_poly() 46
4.4 POLYNOMIAL LEAST SQUARES CURVE FITTING 46
4.4.1 Polynomial Least Squares Curve Fitting Theory 46
4.4.2 Polynomial Least Squares Curve Fitting Function 48
4.4.3 Pseudo Code for Function, poly_leastsqr() 50
4.5 TRIANGULAR FACTORISATION 50
4.5.1 Triangular Factorisation Theory 50
4.5.2 Triangular Factorisation with Pivoting Function 52
4.5.3 Pseudo Code for Function, Triangular_factorisation() 52
4.6 MASTER SECTION 54
4.6.1 Master Theory 54
4.6.2 Neural Input Routine 54
4.6.3 Read Data Files Routine 54
4.6.4 Master Algorithm 55
4.6.5 Write Data Files Routine 56
4.6.6 Pseudo Code for Function, Master() 57
4.7 SLAVE SECTION 58
4.7.1 Slave Theory 58
4.7.2 Read Data Files Routine 58
4.7.3 Predictor Model 58
4.7.4 Slave Algorithm 59
4.7.5 Write Data Files Routine 59
4.7.6 Pseudo Code for Function, Slave() 59
4.8 SIMULATION LOGFILE SECTION 61
4.8.1 Logfile Creation Function 61
4.8.2 Pseudo Code for Function, logfile_create() 61
4.9 SIMULATION ELAPSED TIME SECTION 62
4.9.1 Elapsed Time Start and Finish Functions 62
4.9.2 Pseudo Code for Function, elapsed_time_start() 62
4.9.3 Pseudo Code for Function, elapsed_time_finish() 62
4.10 PROGRAM VARIABLE INITIALISATION SECTION 63
4.10.1 Initialisation Function 63
4.10.2 Pseudo Code for Function, Init() 63
4.11 INCLUDE FILES AND DECLARATION OF GLOBAL VARIABLES 63
4.12 SIMULATION SCRIPT FILE SECTION 65
4.13 MAIN SECTION 66
4.13.1 Main Function 66
4.13.2 Pseudo Code for Function, main() 66

SOURCE CODE 67

5.1 INTRODUCTION 67
5.2 SOURCE CODE OF MAIN PROGRAM 68
5.3 SOURCE CODE OF SCRIPT.H INCLUDE FILE 81
5.4 SOURCE CODE OF INTERPOLATION FUNCTION 82

SIMULATION RESULTS 83

6.1 INTRODUCTION 83
6.2 SIMULATION RESULTS AND EVALUATION 84

CONCLUSION 93

RECOMMENDATIONS FOR FURTHER WORK 96

REFERENCES 98

APPENDIX A MODIFIED STARK MODEL FOR THE HUMAN ARM - STATE EQUATIONS 101

APPENDIX B GANTT CHART..102

LIST OF FIGURES

FIG. 1.1: BLOCK DIAGRAM OF TELEOPERATOR SYSTEM

FIG. 1.2: CONCEPT OF TELEROBOTICS

FIG. 1.3: E1, THE FIRST (1954) ELECTRIC MASTER-SLAVE TELEOPERATOR

FIG 1.4: HANDYMAN, THE FIRST (1958) ELECTROHYDRAULIC MASTER-SLAVE TELEOPERATOR

FIG. 1.5: AN EARLY WHEELCHAIR ARM-AID OPERATED BY THE HANDICAPPED PERSON'S TONGUE

FIG 2.1: POSITION – POSITION LOOP

FIG 2.2: POSITION – FORCE LOOP

FIG. 2.3: TWO-TERMINAL-PAIR NETWORK

FIG. 2.4: CONNECTION OF POWER SOURCE AND LOAD TO A TWO-TERMINAL-PAIR NETWORK

FIG. 2.5: HUMAN REACTION

FIG. 2.6: LOCAL DEFORMATION OF OBJECT SURFACE DUE TO ROBOT ACTION

FIG. 2.7: IDEAL STATE OF MASTER-SLAVE SYSTEM

FIG. 2.8: INTERVENING IMPEDANCE MODEL

FIG. 2.9: CATEGORIES OF SEMIAUTONOMOUS CONTROL

FIG. 3.1: TELEOPERATION THROUGH TIME DELAY AND PREDICTOR

FIG. 3.2: TELEOPERATION THROUGH TIME DELAY AND PREDICTOR, BUT SETUP FOR NEUROPREDICTIVE TELEOPERATION

FIG. 3.3: (A) TRADITIONAL, AND (B) PROPOSED SCHEME

FIG. 3.4: THE ENHANCED YOKOKOHJI AND YOSHIKAWA TELEMANIPULATION SCHEME

FIG. 4.1: LINEAR INTERPOLATION [31]

FIG. 4.2: REGRESSION LINE AND ERROR ASSOCIATED WITH POINT (X_I, Y_I) [31]

FIG 6.1: MASTER AND SLAVE POSITION GRAPH

FIG 6.2: PREDICTED MASTER POSITION AND SLAVE POSITION GRAPH

FIG 6.3: PREDICTED MASTER FORCE GRAPH

FIG 6.4: MASTER ACTUATOR DRIVING FORCE GRAPH

FIG 6.5: SLAVE ACTUATOR DRIVING FORCE GRAPH

FIG 6.6: MASTER AND SLAVE POSITION GRAPH

FIG 6.7: PREDICTED MASTER POSITION AND SLAVE POSITION GRAPH

FIG 6.8: PREDICTED MASTER FORCE GRAPH

FIG 6.9: MASTER AND SLAVE ACTUATOR DRIVING FORCE GRAPH

FIG 6.10: MASTER AND SLAVE POSITION GRAPH

FIG 6.11: PREDICTED MASTER POSITION AND SLAVE POSITION GRAPH

FIG 6.12: PREDICTED MASTER FORCE GRAPH

FIG 6.13: MASTER AND SLAVE ACTUATOR DRIVING FORCE GRAPH

LIST OF TABLES

TABLE 2.1: RESPONSE STAGES
TABLE 2.2: HUMAN SENSES
TABLE 4.1: GLOBAL VARIABLE DECLARATIONS
TABLE 4.2: SCRIPT FILE PARAMETERS
TABLE 6.1: SIMULATION PARAMETERS

INTRODUCTION

1.1 Brief Introduction to Teleoperation

The 20th century has shown a massive technology increase in development, at a point that our civilization can be characterized as technical. The use of machines has already replaced human activity in most repeated and heavy works. The limited physical abilities of the human body have been increased through the use of machines, and allow tasks beyond the human will. On the other side, human's ability to think and decide has not been sufficiently understood, to allow intelligent systems to take over tasks that require important initiative and complex data processing. Recently the development of computers and more specifically of techniques of artificial intelligence has resulted in a concentration of science at this field of interest.

One of the human's inventions that hope to fully replace its inventor is no other than

the robot[1]. Programmed industrial robots are already used, for performing repeated tasks, such as welding, assembling and product spraying. The use of artificial intelligent in this type of robots is expected to allow them to be used in more complex duties, as to perform tasks in natural or unstructured environments. In the meantime, technology has not reached this point of maturity, but even when it does, several and serious problems will still require the cooperation between man and machine, or the continuous supervision for the correct function of the machines from qualified staff.

In the above category of machines belong tasks with unpredictable outcomes, such as discovering of the ocean bed or outer space or even more dangerous, like processing of nuclear radioactive material, mines, fires or handling gun systems for military operations. It is obvious that these kinds of tasks cannot be left completely under the machine's judgement, and even though human presence is necessary, it is impossible due to human biological limitations. Biological limitations include danger of death due to chemical toxic environments or generally due to inappropriate natural environments, as well as inadequacy of human senses and neural forces.

From the above, rises the need of a mechanical arrangement, which will execute the operator's orders and return the state of the current worksite back to him, while the operator is located in a safe or distant place. The machine acts in this way as an operator's representative. The robotic arrangement described above is known as *telerobotic* system.

A good staring point for realising telerobotic systems is the concept of *teleoperation*.

[1] The word *robot* is derived from the Slav word *robota*, meaning obligatory work or servitude.

According to an expert in this field, T. B. Sheridan [26], teleoperation is the extension of a person's sensing and manipulation capability to a remote location. A *teleoperato*r includes at the minimum, artificial sensors, arms and hands, a vehicle for carrying these, and communication channels to and from the human operator. The term teleoperation refers most commonly to direct and continuous human control of the teleoperator, but can also be used generally to encompass *telerobotics* as well.

A teleoperator system can be represented by the block diagram of Fig 1.1 and consists of five subsystems: the human operator, the master, the communication block, the slave and the environment. The slave is usually located in the worksite and is usually a classic industrial arm. The master is located in the same place with the human operator. Through the master, the human operator gives an order to the system and feels back the response of his actions. The master arm could very well be a force feedback joystick for example. The system roughly works as follows. The operator commands a force F_h through the master, communication block, and slave, to the environment. The master responds by changing the state of X_m (position and velocity), which is transmitted through the communication block to the slave. The slave also complies with environment force F_{es}. Both X_s and F_{es} are transmitted back to the master. Finally a new force F_{em} is produced towards the master in order to reconstruct the slave's state.

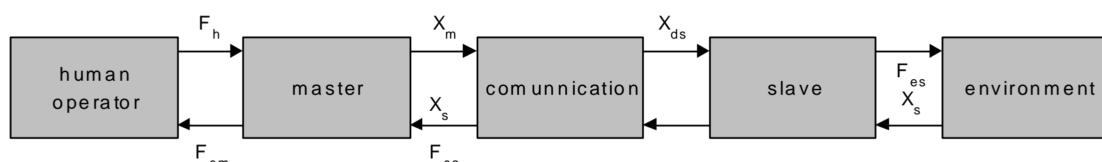


Fig. 1.1: Block diagram of teleoperator system

A teleoperator system must be able to reconstruct the touch feeling with the current object back to the master, who is away from the object. This way, the concept of

telepresence is achieved. Telepresence is the ideal of sensing sufficient information about the teleoperator and task environment, and communicating this to the human operator in a sufficiently natural way, that the operator feels physically present at the remote site [26]. A perfect system should have such a design, so the human cannot realise its existence, but have the illusion that his touching the object with its own hands and seeing it as well. For example if the object is chattering then the human operator must feel exactly the same movement on his hands through the master (e.g. force feedback joystick). An ideal response according to the concept of teleoperation is only achieved, if the position and force on the master arm are equivalent to those on the slave arm for every time interval.

The concept of teleoperation is playing a major role in *telerobotics.* Telerobotics is a form of teleoperation in which a human operator acts as a supervisor, intermittently communicating to a computer information about goals, constraints, plans, contingencies, assumptions, suggestions and orders relative to a limited task, getting back information about accomplishments, difficulties, concerns, and as requested, raw sensory data-while the subordinate *telerobot* executes the task based on information received from the human operator plus its own artificial intelligence [26]. The concept of telerobotics is illustrated in Fig. 1.2. The human operator provides largely symbolic commands (concatenations of typed symbols or specialized key presses) to the computer. However, some fraction of these commands still must be analogical (hand-control movements isomorphic to the space-time-force continuum of the physical task) in order to point to objects or otherwise demonstrate to the computer relationships that are difficult for the operator to put into symbols [26].

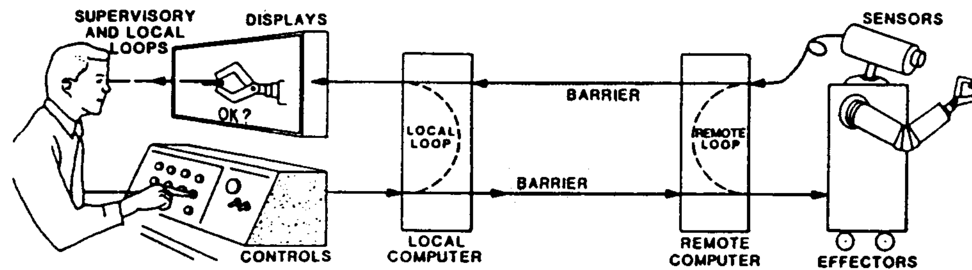


Fig. 1.2: Concept of telerobotics

Mostly synonymous with telerobotics is *supervisory control*, referring to the analogy of a human supervisor directing and monitoring the activities of a human subordinate. The term supervisory control is used commonly to refer to human supervision of any semi-autonomous system (including an aircraft, a power plant, etc.), while *telerobot* commonly refers to a device having arms for manipulating or processing discrete objects in its environment.

Teleoperation represents one of the first domains of robotics and one of the most challenging [2]. In teleoperation a human operator conducts a task in a remote environment via master and slave manipulators. Providing contact force information to the human operator can improve task performance. Although this information can be obtained from visual displays, it is more useful when provided directly, by reflecting the measured force to motors on the master. When this is done, the contact force is said to be "reflected" to the human operator, and the teleoperator is said to be controlled bilaterally [3], [4]. When teleoperation is performed over a great distance, such as in undersea and outer space operations, a time delay is incurred in the transmission of informing from one site to another [26].

1.2 Early History and Applications

From well before the sixteenth century there were teleoperators in the form of fire-tongs, animal prods and other simple arm extensions. Early in the nineteenth century there were crude teleoperators for earth moving, construction and related tasks. By the 1940s prosthetic limb fitters had developed arm hooks activated by leather thongs tied to other parts of the wearer's body.

In about 1945 the first modern master-slave teleoperators were developed by Goertz at Argonne National Laboratory near Chicago. These were mechanical pantograph mechanisms by which radioactive materials in a "hot cell" could be manipulated by an outside the cell. Electrical servomechanisms soon replaced the direct mechanical tape and able linkages (Goertz and Thompson, 1954), and closed circuit television was introduced, so that now the operator could be an arbitrary distance away. Figure 1.3 shows the first electric master-slave teleoperator, built by R. Geortz (shown) at Argonne National Laboratory.

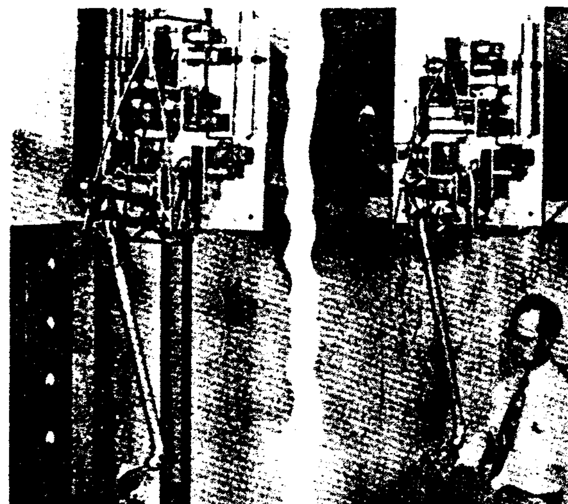


Fig. 1.3: E1, the first (1954) electric master-slave teleoperator

By the mid 1950s technological developments in telepresence (they did not call it that at the time) were being demonstrated. Among these were: force reflection simultaneous in all six degrees-of-freedom (DOF); coordinated two-arm teleoperators; and head-mounted displays, which drove the remote camera position and thereby produced remarkable visual telepresence. Particularly impressive was Mosher's (1964) development of the General Electric Co. Handy-man, which had two electro hydraulic arms with ten DOF in each arm (two DOF on each of two fingers). This is shown in Figure 1.4.

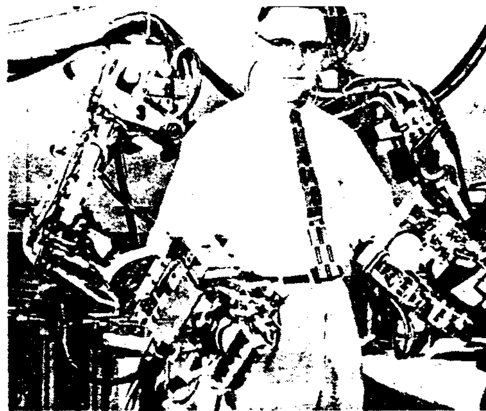


Fig 1.4: Handyman, the first (1958) electrohydraulic master-slave teleoperator

Already in the late 1950`s there was interest in applying this new servomechanism technology to human limb prostheses. Probably the first successful development was that of A. Kobrinskii (1960) in Moscow, a lower arm prosthesis driven by minute myoelectric signals picked up from the muscles in the stump or upper arm. This was followed rapidly by similar developments in the U.S. and Europe (in the mid to late 1960s), including teleoperators attached to the wheel chairs of quadriplegics, which could be commanded by the tongue or other remaining motor signals (shown in Fig. 1.5 overleaf). By that time remote touch sensing and display research was already underway.

Fig. 1.5: An early wheelchair arm-aid operated by the handicapped person's tongue

From the early 1960s telemanipulators and video cameras were being attached to submarines by the U.S., U.S.S.R., and French navies and used experimentally. For example, the U.S. Navy's CURV vehicle (Fig. 1.6) was used successfully in 1966 to retrieve a nuclear bomb from the deep ocean bottom, accidentally dropped from an airplane off Polomares, Spain. Offshore mineral extraction and cable-laying firms soon became interested in this technology to replace human divers, especially as oil and gas drilling operations got deeper.

By 1970 the western interest in teleoperation had turned to undersea, for there was great economic demand for offshore oil. The French developed their ERIC vehicle, the Americans the Hydro products RCV 150, both small-unmanned submarines with remotely controlled video and manipulation capability-plus the necessary thrusters for manoeuvring.

By 1970 industrial (manufacturing) robotics was coming into full development, for Unimation, General Electric, and Cincinnati Milacron in the U.S., Hitachi, Fujitsu and others in Japan, and many firms throughout both western and eastern Europe had

begun using relatively simple assembly-line robots, mostly for spot welding and paint spraying. By 1980 industrial robots had wrist-force sensing and primitive computer vision, and push-button *teach pendant* control boxes were being used for relatively simple programming from the shop floor. It became clear that human teleoperation for working in space, undersea or other hazardous environments was to follow a different course than was industrial robotics.

An example of a space teleoperator capability is the 20 m long remote manipulator system (RMS) built by the Canadian firm SPAR and carried aboard the U.S. space shuttle. It has six DOF and is controlled directly by a human operator viewing through a window or over video and using two three-axis variable rate command joysticks, one for three translations, one for three rotations.

As discussed previously the use of telerobotics has been used in undersea applications. Figure 1.7 illustrates two subsea structures fitted with telerobotic systems. The operator in the first one is located inside the craft and therefore he takes his own decisions about the morphology of the ocean bed and the objects of which are to be collected. The second one is operated remotely and was used for installing and maintaining oceanographic base station on the ocean bed. Teleoperators in undersea applications are usually referred as *ROVs* or remotely operated vehicles. ROVs are commonly used in oil extraction industry. By the use of ROVs a dramatically reduction in the cost of installation and supervision of subsea structures is achieved as well as in the risk of human life. Not to mention of course that a diving hour costs near 10,000 dollars and risks taken by the divers are highly increased.

One of the most interesting applications of telerobotics is *telesurgery* or more specifically operating patients from a distance, when under certain conditions their transfer in hospital is not feasible. The operation on a wounded soldier will take place in a special mobile surgery while the doctor-teleoperator stays away. In this way, fast medical attention is achieved without risking doctor's life. The arrangement plan discussed above, is described by R.M. Satava [5] as *telepresence surgery* (Fig 1.8).

Other areas of telerobotics include offshore mineral extraction, maintenance of underground installations - for example, Russians have constructed a robot for teleinspection and repair of pipelines – smart warehousing, firefighting, policing and military operations.

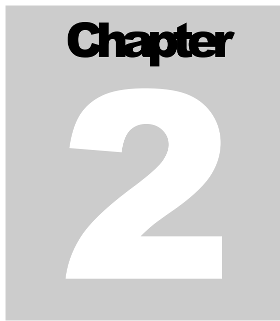
1.3 Scope of the project

This project is based on the implementation and **testing of variable-time-delays-robust telemanipulation through master state prediction** by using high level languages (C++) and Matlab software package.

A general technique for time delay compensation in teleoperation applications is utilised by predicting the human arm position and force (effectively the master state). The technique is based on the prediction of the master state (position x_m and force f_m) only, which can be much more simple and accurate than predicting the slave and environment dynamics.

The telemanipulation method can be split into its fundamental elements and implemented as a number of functions. The first task is to implement some predictor model based on different theories, and decide which one is suitable for the current application. After that the rest of the functions should be developed.

Furthermore the implemented method should be compiled, to make sure that there are no errors, and executed for a number of different parameters. The simulation results should be represented as a number of graphs and evaluated.



CONTROL OF TELEOPERATION SYSTEMS

2.1 Introduction

In this chapter a brief presentation of the most important matters concerning the control of teleoperator systems is attempted.

2.2 Classification of Control Architectures

The possible ways of control of a teleoperator system could be classified in four categories depending on the feedback information [6]. Although there are different variations of these basic control loops, all the existed arrangements could be fitted in one of those. The control loops mentioned before are further analysed on next page.

2.2.1 Position to Position Loop

Position is the fundamental variable in this case. Master and Slave positions appear on the left side of the equations and describe the response of the system, ignoring the reaction force. The control law equations are given by:

$$u_{master} = K_m \cdot (\vartheta_s - \vartheta_m) + K_{feedforward} \cdot \omega_s - K_{feedback} \cdot \omega_m - K_{tfeedback} \cdot \tau_m \quad (1)$$

$$u_{slave} = K_s \cdot (\vartheta_m - \vartheta_s) + K_{feedforward} \cdot \omega_m - K_{feedback} \cdot \omega_s - K_{tfeedback} \cdot \tau_s \quad (2)$$

Figure 1.9 illustrates the above loop in terms of a block diagram.

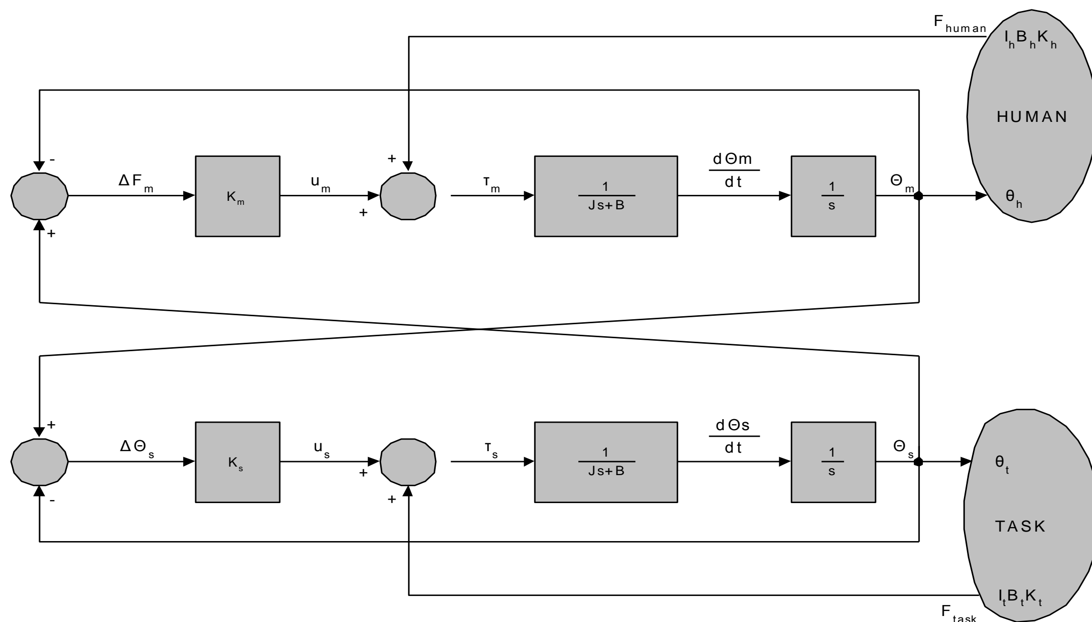


Fig 2.1: Position – position loop

2.2.2 Position – Force Loop

The transferred variables in this case are the master's position and the force at slave's arm. In other words, force control is applied to the master, while position control is applied to the slave. The control law equations that describe the above type of loop are shown below:

$$u_{master} = K_{\tau m} \cdot (f_s - [f_m]) + K_{feedforward} \cdot \omega_s - K_{feedback} \cdot \omega_m - K_{tfeedback} \cdot \tau_m \quad (3)$$

$$u_{slave} = K_s \cdot (\vartheta_m - \vartheta_s) + K_{feedforward} \cdot \omega_m - K_{feedback} \cdot \omega_s - K_{tfeedback} \cdot \tau_s \quad (4)$$

The block diagram representation of the current loop is shown in Figure 2.2.

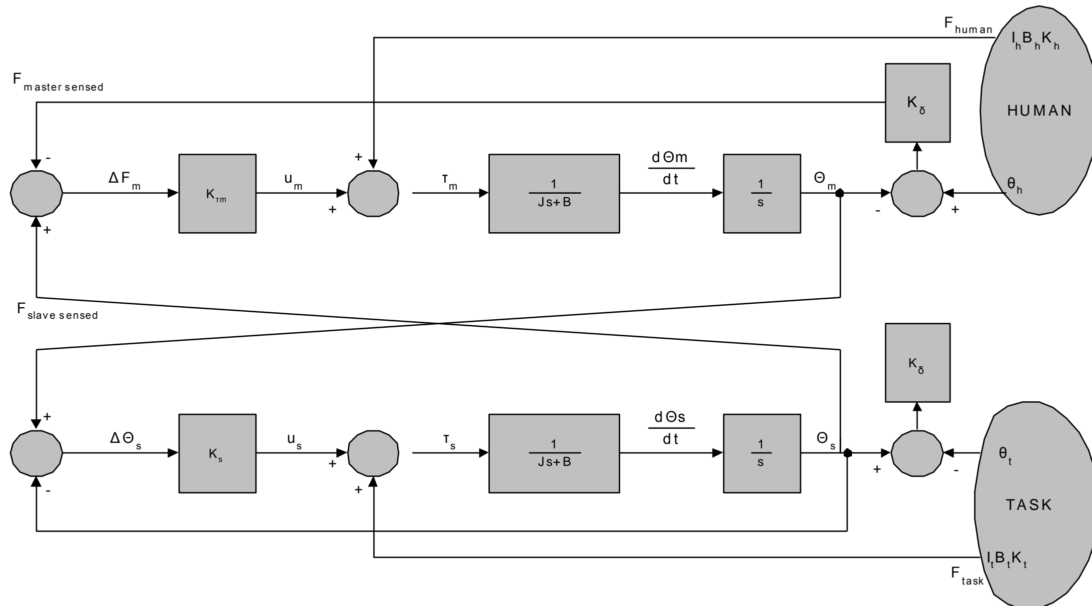


Fig 2.2: Position – force loop

It has to be mentioned that master's force f_m is not necessary required for the loop to close, due to the fact that the operator's hand reacts with the hand-held pendant and balances the open loop force order. Block K_δ represents force sensors fitted on the master control arm (e.g. force feedback joystick) and on the end-effector of the slave robot.

An advantage of this specific loop is that direct measurement of the master force can be achieved. As a result of that is that the slave force sent back to the master is free of friction or non-modelled parameters. In the previously discussed case of position – position loop (2.1.1), the reaction force sent to the master and felt by the operator was generated indirectly, with as a result to include all of the unwanted data mentioned above.

Another advantage of the current method is that the high frequency details of the force are directly measured and sent to the master. Since the master is lighter and faster

compared to the slave, is able of reconstructing the high frequency details while the last rejects them due to different design. Finally the discussed control loop allows the application of mechanical impedance control methods.

A great number of teleoperation methods use this type of control loop or variations based to that.

2.2.3 Position – Force Loop

This control method is a reversed version of the method described above (2.1.2). The method has several problems since it’s not possible for a robot controlled according to force to become stable when it touches a hard object. On the other side, the contact with soft objects is not causing any problems. On the master side, the force control is stable, since the operator’s hand conforms to the external force.

2.2.4 Force - Force Loop

Not any system with force control on both sides had been implemented by the time of publication of [6]. Bobgan and Kazerooni introduced the first system in 1991 and capable conditions were established for system stability [7].

2.3 Master-Slave System Representation by Two-Terminal-Pair Network

Two-terminal-pair network is usually used in the analysis of electrical circuits (shown in Fig 2.3).

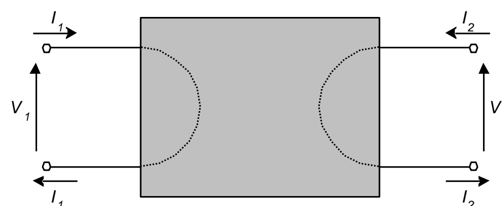


Fig. 2.3: Two-terminal-pair network

Impedance matrix **Z** is defined from the relations (shown below) between current and voltage of a two-terminal-pair network.

$$V_1 = z_{11} I_1 + z_{12} I_2 \tag{5}$$

$$V_2 = z_{21} I_1 + z_{22} I_2 \tag{6}$$

$$Z = \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix} \tag{7}$$

where I_1 and I_2 denote current at each terminal pair, and V_1 and V_2 denote voltage at each terminal pair.

Lets consider a two-terminal-pair network, which is connected to a power source and a load at each terminal pair as shown in Figure 2.4.

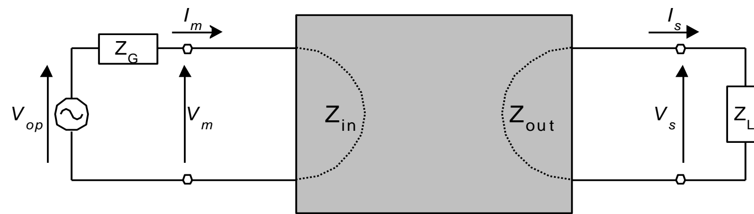


Fig. 2.4: Connection of power source and load to a two-terminal-pair network

Regarding the power source as an operator as an operator, the load as an object and the two-terminal-pair network as a master-slave system, the whole system can be replaced by the electric circuit of Fig. 2.4. The correspondence between a master-slave system and the circuit representation in Fig. 2.4 is given as:

- velocity of the master arm $\dot{x}_m$ ⟵⟶ current I_m
- velocity of the master arm $\dot{x}_s$ ⟵⟶ current I_s
- operator's force τ_{op} ⟵⟶ voltage V_{op}
- force at the master side f_m ⟵⟶ voltage V_m
- force at the master side f_s ⟵⟶ voltage V_s

Representation of master-slave system by a two-terminal-pair network is not a new idea. However, Razu [8], [9] has shown the framework where the operator and object are considered as a power source and load connected to the network. The concept of the two-terminal-pair network is well used to design electric filters. The master-slave can also be considered as assort of mechanical filter between the operator and the object.

This circuit representation does not change the nature of the problem all, but it enables us to formulate in compact forms [10], also the extraction of system equations becomes an easy task and could be solved by using one of many circuit analysis, computer packages available.

2.4 Evaluation of Stability based on Passivity of the System

The characteristic approach is applicable only when the dynamics of the operator and object can be represented by linear systems. Strictly speaking, however, the operator dynamics and some of the object dynamics may be non-linear. For this reason, the passivity of the system is used by many researchers, in order to study the stability of the system [10].

Passivity of the system can be a sufficient condition of stability only when then system interacts with passive environments. In the case of master-slave systems, assuming that the operator and the environment are passive systems, then the sufficient condition of stability is that the master-slave system itself must be passive [10]. However, the operator is not passive because he/she has muscles as the power source and therefore is not going to turn the system unstable.

2.5 Performance Evaluation of Teleoperator Systems

Performance testing and the relative problem of comparing teleoperator systems is quite complex due to the fact that the systems are composite in nature plus that human factor is present. Moreover, the system should take under consideration several tasks for execution, such as the movement in free space, in liquid or when comes in contact with an object.

A mathematical algorithm for performance evaluation is proposed by Yokokohji and Yoshikawa and presented below [10]. Initially, the ideal response is set, in which the slave and master position and force are identical at any instant of time, or in other words the perfect achieved form of teleoperation. Then a quantitative index of manoeuvrability[2] is proposed based on the concept of ideal responses previously discussed.

Let $G_{mp}(s)$, $G_{sp}(s)$, $G_{mf}(s)$, and $G_{sf}(s)$ be transfer function of the master-slave system from the operator's force to the master side displacement, slave side displacement, master side force, and slave side force respectively. Then the two following indexes are defined:

$$J_p = \int_0^{\omega_{max}} | G_{mp}(j\omega) - G_{sp}(j\omega) | \left| \frac{1}{1 + j\omega T} \right| \cdot d\omega \qquad (8)$$

$$J_f = \int_0^{\omega_{max}} | G_{mf}(j\omega) - G_{sf}(j\omega) | \left| \frac{1}{1 + j\omega T} \right| \cdot d\omega \qquad (9)$$

where ω_{max} is the maximum frequency of the manipulation bandwidth of human operators, $T(T\omega_{max} > 1)$ is time constant of first-order-lag.

[2] A high-performance master-slave system means that it can provide high manoeuvrability.

When index J_p is zero then the displacement is identical on both sides (master-slave). When index J_f is zero then the force is identical on both sides. For a non-ideal situation, the system becomes better as both J_p and J_f get closer to zero.

2.6 Human Reaction and Modelling

In [6] is discussed the nature of human senses that being used by master-slave systems. An analysis of the frequency response of the human body is performed and the bandwidth of stimulations that the operator can perceive is defined. Finally is concluded that human have an uneven number of input (stimulation perception) to output (action) abilities. Figure 2.5 illustrates in terms of a block diagram the human input/output abilities.

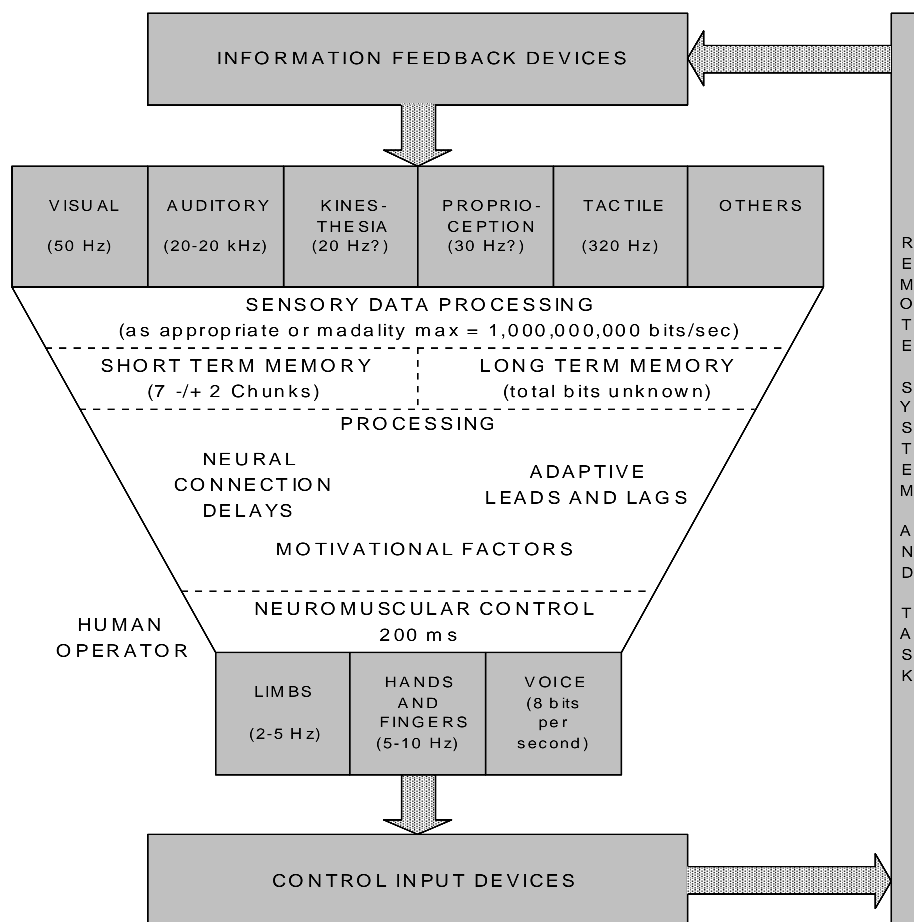


Fig. 2.5: Human Reaction

According to figure 2.5 a bandwidth of about 320 Hz is required for transmission of stimulations, while about 10 Hz is required for hands movement.

In the next table (Table 2.1), the reaction time of a human to a number of stimulations and the different stages that intervenes before the end of reaction is analysed [11].

Response Stages	Typical Delay (msec)
Sensor	1 - 380
Transmission delay to brain	70 - 300
Natural transmission to muscle	10 - 20
Reaction of muscle	30 - 70
Total	**113 - 528**

Table 2.1: Response stages

Generally "fast" humans need in ideal situations, about 200 msec.

Reaction time is also closely related to human senses and therefore affected by them as shown in table 2.2.

Senses	Time (msec)
Hearing	150
Vision	200
Scent	300
Pain	up to 700

Table 2.2: Human senses

Human modelling can be achieved with several methods depending on the required degree of precision. For example when master-slave systems are compared in analogy with electric circuits (for analysis purposes), the operator is represented as the current.

It very common to model the operator as a simple spring-damper-mass system given by the following equation:

$$\tau_{op} - f_m = m_{op}\ddot{x}_m + b_{op}\dot{x}_m + c_{op}x_m \tag{10}$$

where m_{op}, b_{op} and c_{op} denotes mass, viscous coefficient, and stiffness of the operator respectively, whereas τ_{op} means force generated by the operator's muscles and f_m denotes the force that the operator applies to the master arm. The displacement of the operator is represented by x_m because it is assumed that the operator is firmly grasping the master arm and he/she never releases it during the operation [10].

The procedure of decision-making by the operator (estimating τ_{op}) is a difficult task that requires the use of artificial intelligence knowledge.

2.7 Environment Modelling

When the robot touches an object, a force F_e is applied to the robot, given by:

$$F_e = -Z_e x'_e \tag{10}$$

where vector x'_e represents the local deformation of object surface due to robot action (Fig. 2.6), and is equal to:

$$x'_e = \begin{cases} x - x_e \\ 0 \end{cases} \tag{11}$$

The above simple model is used, since a precise model of contact would be very difficult to describe due to natural phenomena that appear. The error that occurs by using this simple model is corrected by the controller. Equation 10, describes the contact with an unbend surface, elastically conformed to the external pressure, without friction. A flat surface selection is a good approximation for the area near to the contact point, for regular curved surfaces.

Assuming an unbend surface, allows to omit local conformation results due to contact. Based on these considerations, matrix Z_e can be represented as:

$$K = k \cdot nn^{\tau} \qquad (12)$$

where $k>0$ is the stiffness coefficient and n is a unity vector vertically directed on the surface area and hence defines the orientation of last.

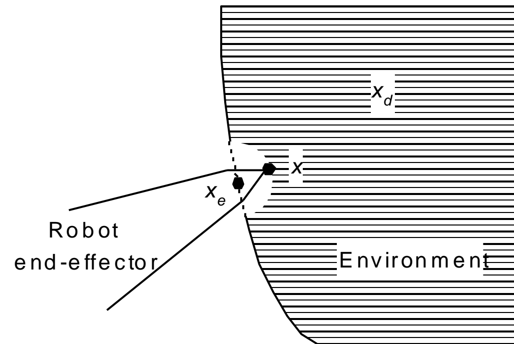


Fig. 2.6: Local deformation of object surface due to robot action

2.8 The Yokokohji and Yoshikawa Control Law

The Yokokohji and Yoshikawa [10] control law is a very important law that realizes teleoperation. The architecture used is a classical one, developed through a well-defined general framework of teleoperation, and adopting the widely accepted design specifications of system transparency and passivity [12].

Usually master-slave systems consist of arms with multiple DOF. However for problem simplicity a one DOF system is considered.

The dynamics of master and slave arms is given by the following equations:

$$\tau_m - f_m = m_m \ddot{x}_m + b_m \dot{x}_m \qquad (13)$$

$$\tau_s - f_s = m_s \ddot{x}_s + b_s \dot{x}_s \qquad (14)$$

where x_m and x_s denote the displacements of the master and slave arms, and m_m, b_m, m_s, b_s represent mass and viscous coefficient of the master and slave arms respectively. In addition, f_m denotes the force that the operator applies to the master arm, and f_s denote the force that the slave arm applies to the object. Finally, τ_m and τ_s represent actuator-driving forces of master and slave arms respectively.

The dynamics of the object interacting with the slave arm, is modelled by the following linear system:

$$f_s = m_w \ddot{x}_s + b_w \dot{x}_s + c_w x_s \tag{15}$$

where m_w, b_w, and c_w represent mass, viscous coefficient, and stiffness of the object respectively. It is assumed that the slave arm is contacting the object, in such a way that it may not depart from the object. Moreover is assumed that the dynamics of the operator can be approximately represented as a simple spring-damper-mass system, described by the following equation:

$$\tau_{op} - f_m = m_{op} \ddot{x}_m + b_{op} \dot{x}_m + c_{op} x_m \tag{16}$$

where m_{op}, b_{op} and c_{op} denotes mass, viscous coefficient, and stiffness of the operator respectively. In addition τ_{op} represents the force generated by the operator's muscles and f_m denotes the force that the operator applies to the master arm. The displacement of the operator is represented by x_m because it is assumed that the operator is firmly grasping the master arm and he/she never releases it during the operation. Figure 2.6 [10] shows the model of one DOF teleoperation system:

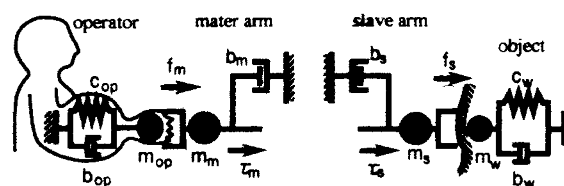


Fig. 2.6: Teleoperation system

Assuming that the force is identical on both sides, and the displacement difference between master and slave is zero at any time, the following control schemes for τ_m and τ_s can be considered:

$$\tau_m = m_m[x_{ms} + k_1(x_{ms} - x_m) + k_2(x_{ms} - x_m)] + b_m x_m - k_{mf}(f_{ms} - f_m) - f_{ms} \quad (17)$$

$$\tau_s = m_s[x_{ms} + k_1(x_{ms} - x_s) + k_2(x_{ms} - x_s)] + b_s x_s - k_{sf}(f_{ms} - f_s) - f_{ms} \quad (18)$$

where $x_{ms} \equiv (x_m + x_s)/2$ and $f_{ms} \equiv (f_m + f_s)/2$

The above control laws (Eqs. 17, 18) assume that all the information (position, velocity, acceleration, and force) are known and time delay due to data transmission between the master and slave sites is negligible. It is also assumed that the scales of position and force are identical between the master and slave sites.

If the above considerations are obeyed then the system remains stable and achieves the ideal state of master-slave system. In other words the system is equivalent to a weightless rigid bar connecting the operator with the object (Fig. 2.7).

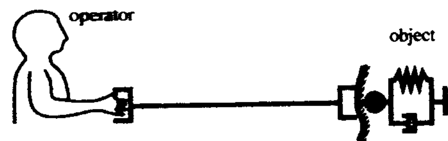


Fig. 2.7: Ideal state of master-slave system

However, the above state is very critical because only a small error of the inertia parameter may change the massless bar into a bar with negative mass.

In order to avoid that, the control law is modified in such a way, that the dynamics of master-slave system are not cancelled, but the operator feels as if he was operating the object through a virtual bar of given mechanical impedance. This mechanical

impedance is mentioned by the authors as *intervening impedance*, and illustrated in Fig. 2.8, below.

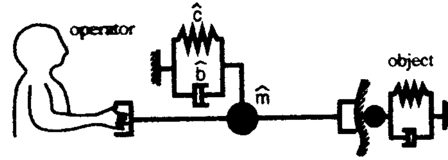


Fig. 2.8: Intervening impedance model

The state of Fig. 2.8 can be described by the following equation by setting $x_m = x_s = x$:

$$f_m - f_s = m'\ddot{x} + b'\dot{x} + c'x \tag{19}$$

where m', b' and c' are the mass, coefficient of viscous fiction, and stiffness of the intervening impedance respectively. Since xm and xs may not coincide all the time, Eq. 19 can be rewritten as:

$$f_m - f_s = m'\ddot{x}_{ms} + b'\dot{x}_{ms} + c'x_{ms} \tag{20}$$

The position error e converges asymptotically into zero according to the following equation:

$$\ddot{e} + k_1\dot{e} + k_2 e = \lambda \frac{f_m + f_s}{2} \tag{21}$$

where λ>0 is a positive constant. Finally the control law (Eqs. 17, 18) become:

$$\begin{aligned}\tau_m = {} & m_m\left[x_{ms} + k_1(x_{ms} - x_m) + k_2(x_{ms} - x_m)\right] + b_m x_m - k_{mf}(f_{ms} - f_m) - f_{ms} \\ & - \frac{(1 + k_{mf})}{2}\left[m'x_{ms} + b'x_{ms} + c'x_{ms}\right] + \frac{\lambda}{2} m_m f_{ms}\end{aligned} \tag{22}$$

$$\begin{aligned}\tau_s = {} & m_s\left[x_{ms} + k_1(x_{ms} - x_s) + k_2(x_{ms} - x_s)\right] + b_s x_s - k_{sf}(f_{ms} - f_s) - f_{ms} \\ & - \frac{(1 + k_{sf})}{2}\left[m'x_{ms} + b'x_{ms} + c'x_{ms}\right] + \frac{\lambda}{2} m_s f_{ms}\end{aligned} \tag{23}$$

Depending on the availability of parameters and by adjusting them appropriately, the ideal response could be achieved.

2.9 Other Control Architectures

B. Hannaford [13] introduced the bilateral impedance control method. According to this method, a local control loop on master tries to regenerate the intervening impedance that exist on the other side (operator), respectively a local control loop on slave tries to regenerate the intervening impedance that exist on the environment. So, this control scheme returns back except the master-slave position, the predicted intervening impedance as well. Therefore, it cannot be enlisted to any of the categories discussed in 2.2.

K. Funaya and N. Takanasi [14] introduced an interesting control method, for adjusting the stiffness of a teleoperation system, according to availability of object positions.

A. Strassberg, A. Goldberg, A. Mills proposed a variation of the force-position loop described in section 2.2.2. The transmit information are the master speed and the slave force. However, the master is controlled in a special way; the force error is converted to speed information and the master is controlled according to speed. The slave is controlled according to speed as well.

2.10 Transmission Time-Delays

In many teleoperation applications the master and generally the control station is located away from the slave. Therefore, every master control order is transmitted delayed by a certain amount of time to slave. It has being proved, that this delay worsens the quality of teleoperation and may lead the system to instability. A typical delay of 20 msec appears, when the master-slave arms are near to each other. Delays

of 60-200 msec appear when the master and slave are connected through a computer network. Over 500 msec to a number of seconds, delays are introduced, when the transmission is via a satellite link. An example of such a delay could be the assembly of a satellite station in orbit from earth.

Two main approaches that can be followed to produce a passive (stable) communication law between master and slave, and overcome the instability and functionality problems caused by time delays are discussed below.

2.10.1 Scattering Theory

The first approach developed by Anderson and Spong (1998) using scattering theory (Johnson, 1950). A scattering operator, S, can be defined for a two port network by the relationship between force and velocity:

$$F - V = S(f + v) \tag{23}$$

where S is a matrix in the frequency domain. Any communication law can be tested for stability using a theorem stating that a two-port network (section 2.3) is passive if only if the norm of its scattering operator is less than or equal to one.

Noting that an analogue electrical transmission line delays signals and is inherently passive, Anderson and Spong manipulated the transmission line equations to obtain the control laws for passive behaviour of the communications block:

$$\dot{x}_{sd}(t) = \dot{x}_m(t - T) - F_s(t) + F_{md}(t - T) \tag{24}$$

$$F_{md}(t) = F_s(t - T) - \dot{x}_m(t) + \dot{x}_{sd}(t - T) \tag{25}$$

where T is the communication time delay. These communications laws are passive (stable) for all time delays, assuming the human operator, master, slave, and

environment can be all represented as passive systems. Furthermore under steady state conditions, the forces and velocities of master and slave are identical [16].

2.10.2 The Wave Variable or Energy Approach

The second approach developed by Niemeyer and Slotine (1990), uses an energy-based formulation. The total power flow into the teleoperator network is given by:

$$P = \dot{x}_1 F_1 - \dot{x}_2 F_2 \tag{26}$$

The power flows can also be formulated with wave variables. Wave variables are motivated by the physical concept of waves with an input and output wave at each port of a network. In this manner, the total power flow can be written as:

$$P = \frac{1}{2}u_m^2 - \frac{1}{2}v_m^2 + \frac{1}{2}u_s^2 - \frac{1}{2}v_s^2 \tag{27}$$

Where u and v are the input wave variables. Equating equations (26) and (27) leads to asset of transformation equations between power variables and wave variables.

2.11 Semiautonomous Control

Even if the operator can execute a task by himself, it's a waste of human resources and time to have to execute repeatable tasks, which they could very well executed by the system, itself. However, a repeated task might tire out the operator and hence reduce his performance. A good solution to these problems is to use semiautonomous control on the slave side. Semiautonomous control is used, because there is no point using a teleoperation system, if the task is fully defined in a structured environment. The operator must be present, first of all to direct the system and finally to intervene in autonomous control when something unpredicted happens. The concept of semiautonomous control is closely related to supervisory control, introduced by Ferrell and Sheridan on 1967.

Semiautonomous control is split into two categories:

- *Serial type,* where manual and autonomous control are alternated serially. This is shown in Fig. 2.9-b. Traditional supervisory control (section 1.1) is included among this scheme.
- *Parallel type,* where manual and autonomous control are acting together. Two sub-categories can be distinguished according to this type.
    - The *combined case*, where the control inputs are combined together (Fig. 2.9-c). This is necessary when autonomous control alters the operator's order or when the operator wants to alter the results caused by autonomous control.
    - The *shared case,* where the current task is shared between the operator and autonomous control (Fig. 2.9-d) An example, could be the transfer of a glass full of water, where the orientation of the glass is autonomously controlled and the position and speed by the operator.

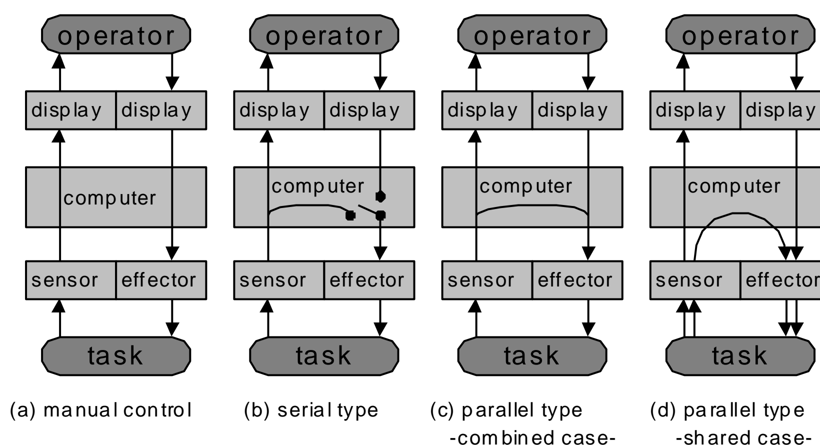
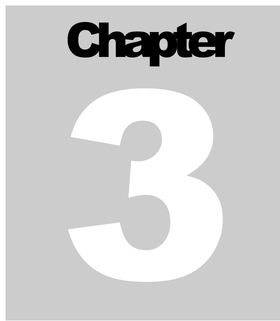


Fig. 2.9: Categories of Semiautonomous Control



TIME-DELAYS-ROBUST TELEMANIPULATION THROUGH MASTER STATE PREDICTION

3.1 Introduction

Although in the telerobotics community significant effort has been concentrated on the compensation of time delays (t.d.s) in the communication channel between master and slave, simple and reliable solutions are still being sought. Three main groups of techniques robusti-fying against t.d.s have appeared up to date (two of them discussed in sections 2.10.1 and 2.10.2). The first is based on the use of predictive displays for the slave and the remote environment [17]. The future slave state is calculated, so that the operator effectively interacts with an adaptable model of the remote site. Models have to be both dynamically and visually correct, thus requiring complicated graphics data and image processing. The second group entails the use of wave variables to form a passive communications channel [18-20]. These provide stability but alter the force

fed back. A final solution is to use supervisory control [17], leading to a non-continuous form of teleoperation, i.e. changing the basic specification of manipulating as close to physical as possible. Recently, significant interest was focused on variable t.d.s, arising for example through the Internet [21-25].

The method proposed in this chapter, uses a prediction of the master state (position x_m and force f_m) *only*, which can be much more simple and accurate than predicting the slave and the remote environment, and incorporates this in a stable force-feedback scheme. It will also be shown that this scheme cancels the computational burden of visually representing the slave future state, needed in predictive displays: the slave-side cameras' image is sufficient for optical feedback, since it turns out to be synchronized with the master. This is to the best of the authors' knowledge an unexplored approach. Two early efforts cited in [17], which predict the control input along with the rest of the system state in non-telemanipulation tasks, were judged there to be inadequate for telemanipulation.

Two predictor implementations have been explored. The first, called *trajectory extrapolating prediction*, simply predicts the values of the macroscopic measurable variables x_m and f_m. Simulations were carried out either, as is usual in the literature, ignoring the human arm dynamics and considering predefined shapes of force input, or, more realistically, also including the human dynamics, represented by the Stark model of the human arm, as modified in previous work [21] of the authors. The Stark model and its modifications are given in Appendix A. The second implementation is built around a *model-based* predictor, also employing the Stark model, and the prediction of the neural input to it. Relevant results are reported in [21], [22]. The

proposed method was applied to the "enhanced Yokokohji and Yoshikawa scheme" [22], a modification of [10] accommodating the master state predictor.

The chapter is organized as follows. The proposed concept is introduced in Section 3.2. Model-based predictors are discussed in Section 3.3, and trajectory extrapolating ones in Section 3.4. Section 3.5 outlines the control scheme used [1].

3.2 The Concept of Predicting the Master State

The basic feature of the present design is the incorporation of a predictor for the *master* state rather than the slave and environment one. This offers significant advantages over previous solutions. The key idea is to command the *slave* robot to follow the *predicted* command, so that it is "ahead in time" from the master. After the two transmissions of signals through the communications channel (shown in Fig. 3.1), the reflected slave position / force has the same time index as the local master variables. According to Fig. 3.1, exp($-sT_t/2$) denotes delay due to transmission through the communication channel and exp($+sT_t$) denotes prediction. The other blocks are free of delay. "Hat" ^ denotes estimate. Xss is only used to illustrate the signals' timing. Setup for predictors does not require neural input measurement or estimation (e.g. trajectory extrapolation).

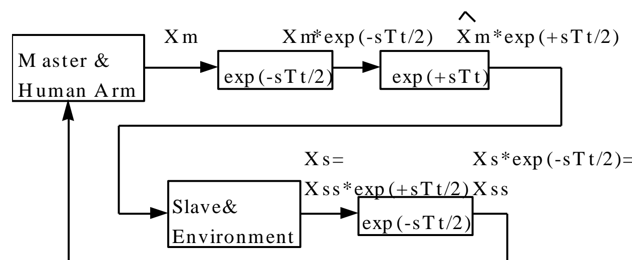


Fig. 3.1: Teleoperation through time delay and predictor

This way the "feel" of teleoperation is natural, since the variables are *simultaneous*, and are not altered by the algorithm, as in existing approaches [18], [20]. This is exactly what happens when a human manipulates by his own hands, i.e. the scheme is transparent. With a prediction horizon of T_t sec, t.d.s up to $T_t/2$ can be compensated for. If the t.d. is smaller than $T_t/2$ or the t.d. is not equal in the two directions, then an additional artificial delay (buffering) has to be introduced. This would be the case if the computational delays were taken into account. They would probably not be matched in the two robots, since incorporating simpler hardware at the slave can be advantageous in space, underwater and other on-field applications. Thus variable t.d.s, arising for example through the Internet, can also be accommodated, provided the exact t.d. is calculated as in [24]. Even for small t.d.s, this prediction can be helpful in providing us time for control error corrections and compliance.
A significant advantage offered by predicting the master state, is that only the plain camera image transmitted back to the operator is needed. No special analysis, such as object recognition to form a world model and make an accurate prediction or use of complicated graphics to overlap the predicted slave position on the camera image, demanded by conventional predictive display systems, is needed. This simplification occurs because the slave leads in time the master in the real world, not as a computer model. In other schemes is the opposite, i.e. the master leads in time the slave.

The performance is clearly affected by the prediction accuracy. When the error is significant, the reaction fed back from the slave corresponds to the falsely predicted operator movement rather than the actual one, so that the human will form a wrong impression for the results of his actions and the remote site, and instability may arise. The prediction fidelity depends on the complexity of the predictor and the profile of

the master movements. The latter are determined by the human physiology and the task to be performed, and can be smoother and thus more predictable by operator training. This chapter investigates the trade-off between predictor complexity and final accuracy, i.e. the feasible t.d. compensation capabilities for various predictor implementations.

The predictors considered can be cast in two groups: *model-based* and *trajectory extrapolating* predictors and are analytically discussed in sections 3.3 and 3.4.

3.3 Model-Based Prediction

Model-based predictors employ a model of the system that generates the master state, thus producing an accurate prediction, provided that this model and a prediction of its input are known. If the macroscopic variables x_m and f_m are considered as input, and due to the control feedback, the human arm, the two robots and the remote environment should be included in this model. Identifying an online-adapting neural network-based "holistic" model of this type was considered in earlier work, but has not so far resulted in robust performance. A non-adaptive model suffers from robot and environment parameter uncertainties and is anyhow quite complex. Despite the apparent similarities, this approach differs from predictive displays, since the master state is also predicted, the slave leads the way and no visual representation is needed.

A simplification tried in the simulations, consisted in considering the slave state as steady during the prediction horizon (only within the predictor, of course). Thus effectively the slave and remote environment are not taken into account. This led to slight degradation of the prediction fidelity, which was not destabilizing.

Luckily, well-established physiological evidence reveals that the brain, rather than controlling the movement on-line, “programs” the arm with an action plan of a complete movement, which is then executed largely in open loop, regulated only by local reflex loops [27]. Therefore, by measuring the neural input (NI- its measurement termed electroneurograph - ENG) to the arm muscles and predicting it before a new "program" is "downloaded", a reliable reproduction of the *intended* master state can be obtained without fear of sudden change of the input. In [22] a control scheme assuring that this intended movement is realized by the actual master state, is designed (shown in Figs. 3.2, 3.3). Fig. 3.2 is the same as Fig 3.1, but setup for Neuropredictive Teleoperation. **Φ** is the HNI (Hypothetical Neuropredictive Input). In Fig. 3.3, the traditional and proposed schemes are shown, where, **B**: Brain, **P**: Muscle command program buffer, **V**: Screen and optical pathway, **A**: Arm, **M**: Master, **eM**: Arm Model and Predictor, **S**: Slave, **E**: Environment. Dashed lines should be ignored, unless HNI is estimated through an inverse arm model or when the arm model is tuned online. The dotted connection is open most of the time.

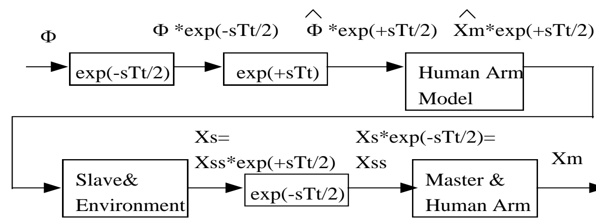


Fig. 3.2: Teleoperation through time delay and predictor, but setup for Neuropredictive Teleoperation

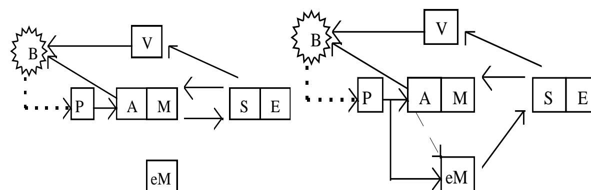


Fig. 3.3: (a) Traditional, and (b) proposed scheme

There, the control loop traditionally closed around the master and slave is broken, so that the system reduces to open loop, for the interval during which the arm moves autonomously from the brain, i.e. most of the time (since the forward path from the brain to the slave passes through the predictor rather than the physical arm and master). By the combination of measuring and predicting the NI, the human arm is reduced to a sensory feedback platform for the brain, and a source of correcting the model and the prediction. This scheme was named "Neuropredictive Teleoperation" (NPT). Since invasive techniques would be required to measure the ENG, the electromyograph (EMG) and relevant models can be employed instead. A multitude of human arm models can be found in bioengineering literature [21].

The ENG / EMG are not used in the models as measured, but are rectified and further processed [28], [21], [29]. To emphasize this, the term "hypothetical neural input" (HNI) will be used. The HNI has a simple form and some well studied characteristics: it is a three pulses' sequence, modeled by varying the amplitude *or* the period. It is a square or triangular waveform, either continuous or "spiky" modulated by a square or triangular function, whereas the rectified EMG is (roughly) sinusoidal [21]. Such a signal is indeed easily predictable. Depending on the model, this is accomplished either by simulation ("running the model forward") or by modifying it to a predictive formulation (i.e. by analytical calculations).

A central problem is specifying a horizon T_t for reliable prediction. A clear upper limit is posed by the frequency with which the brain changes its "program", i.e. its predefined sequence of pulses. Such an upper limit is estimated at about 1 sec, whereas a safe limit is 500 msec [30]. Within this limit a predictable set of three pulses

can be expected. To achieve this, good estimation of the duration of each pulse or robust control laws, with fast errors correction, are required. Given the margin of 500 msec, such control tactics are feasible. To be safer, we can resort to short term predictions of HNI / EMG and "invest" on the activation time of the muscle as a response to HNI. Since this is modeled as a 1st order linear system with typical time constant 50 msec [30], its step response to the squared input used in [30] will have settled after around 200 msec. So, after measuring the ENG, one can predict the muscle response after approx. 200 msec.

Considering typical values, t.d. is not considered for many ground applications, for example telesurgery with dedicated communication lines or Internet link (which introduces delays with a mean as small as 0.1 sec [23]), underwater ones for depth of 400m (delay of 1/850 sec/m [17]), or even single-link earth-to-orbit ones(delay of 0.4 sec,6 sec for multiple link [17]). Employment of hints about the intended movement or a combination with existing techniques could increase these limits. Intermediate systems, employing hints about the dynamics rather than an accurate model, are also under development by the authors [22].

3.4 Trajectory Extrapolating Prediction

The disadvantage of the model-based techniques is that identifying a model and measuring its input is not an easy task, while the complexity of the calculations involved is quite high. Another option is to ignore the internal system dynamics and just interpolate x_m and f_m with a simple, say polynomial, function. The prediction is then made by extrapolating this coarse model up to the desired horizon. The decision to test this idea was reinforced by observing that the profiles of position and force trajectories reported in the literature are rather simple. An advantage of this technique

is that it can be directly applied to existing schemes without any modification of the control laws or their underlying assumptions. In contrast to NPT, there is no need to consider the human physiology, since only measurable macroscopic variables are required. In addition, it is computationally non-demanding. However, by separately predicting $\boldsymbol{x}_m$ and $\boldsymbol{f}_m$ their values will not necessarily be compatible to each other, in the way dictated by the nonlinear human dynamics.

Several interpolating families of functions were considered. While the simplest were polynomial functions of low order, exponential ones, and (as in [29]) a 1st order Taylor extrapolator, cubic splines gave the double tolerance to delays. Neural Networks were considered but not tested, since they would either require off-line training to form an initial curve, or comprise of just a few neurons, leading to a performance not better than the other methods. Finally, heuristic modifications did not result in any significant improvement.

3.5 The Enhanced Yokokohji and Yoshikawa Scheme

As mentioned in the Introduction, a conservative improvement of the Yokokohji and Yoshikawa teleoperator architecture [10] was also developed [22].

The Yokokohji and Yoshikawa architecture is a classic one, developed through a well-defined general framework of teleoperation, and following the widely accepted design specifications of aiming at transparency and passivity of the system. In [22] it is robustified against time delays, by being augmented with the predictor outlined above. The general system set-up is actually the same as in [11], except that the goal is now modified to achieving $x_m(t) = x_s(t-T_p/2)$ (Ideal response I), or $f_m(t) = f_s(t-T_p/2)$ (Ideal response II), or both simultaneously (Ideal response III).

The enhanced scheme "pushes" the master and slave state to the mean values:

$$x'_{ms}(t) = \left[x_m(t) + x_s(t - T_p/2)\right]/2 \quad (28)$$

$$f'_{ms}(t) = \left[f_m(t) + f_s(t - T_p/2)\right]/2 \quad (29)$$

These differ from [11], as the slave states are now delayed. A more important difference is that while the master is directed to $x'_{ms}(t)$, $f'_{ms}(t)$, the slave is pushed to $x'_{ms}(t + T_p/2)$, $f'_{ms}(t + T_p/2)$,i.e. it leads the master arm along the desired trajectory.

The dynamics of master and slave arm are given by the equations:

$$\tau_m(t) + f_m = m_m \ddot{x}_m + b_m \dot{x}_m \quad (30)$$

$$\tau_s - f_s = m_s \ddot{x}_s + b_s \dot{x}_s \quad (31)$$

where τ is actuator driving force and m_m, m_s, b_m, b_s are constant parameters.

By applying the control law (Fig. 3.4):

$$\tau_m(t) = m_m\left[\ddot{x}'_{ms}(t) + k_1(\dot{x}'_{ms}(t) - \dot{x}_m(t)) + k_2(x'_{ms}(t) - x_m(t))\right] + b_m\dot{x}_m(t) - k_{mf}\left[f'_{ms}(t) - f_m(t)\right] - f'_{ms}(t) \quad (32)$$

$$\begin{aligned}\tau_s(t) = {} & b_s\dot{x}_s(t) - k_{sf}\left[f_s(t) - f'_{ms}(t + T_p/2)\right] + f'_{ms}(t + T_p/2) + \\ & + m_s\left[\ddot{x}'_{ms}(t + T_p/2) + k_1(\dot{x}'_{ms}(t + T_p/2) - \dot{x}_s(t)) + k_2(x'_{ms}(t + T_p/2) - x_s(t))\right]\end{aligned} \quad (33)$$

where k_{mf}, k_{sf}, k_1, k_2 are constant control parameters, it can be shown that $f_s(t - T_p/2) = f_m(t)$ and $\ddot{e}'(t) + k_1\dot{e}'(t) + k_2 e'(t) = 0$, where $e'(t) = x_m(t) - x_s(t - T_p/2)$, i.e. force tracking is perfect and position error is minimized through the 2nd order error equation above. So, the ideal performance is achieved, despite of the time delay.

As in [11], the above control law can turn the system unstable if the dynamic parameters' estimation is erroneous. It can be enhanced by applying the intervening impedance, concept of Yokokohji and Yoshikawa: the force tracking is relaxed, by imposing, through a modified control law:

$$f_m(t) - f_s(t - T_t/2) = \hat{m}\ddot{x}'_{ms}(t) + \hat{b}\dot{x}'_{ms}(t) + \hat{c}x'_{ms}(t) \qquad (34)$$

$$\ddot{e}'(t) + \dot{e}'(t) + e'(t) = \lambda\left[f_m(t) + f_s(t - T_t/2)\right]/2 \qquad (35)$$

where λ is a constant parameter. This way, the operator feels as if manipulating through a virtual rod, an intervening impedance, whose (constant) parameters are denoted by above "hat" symbols.

In [22] it is shown that the combination of the teleoperator and the predictor is passive under perfect prediction. It is also shown that it can be passive under non-perfect prediction, depending on the prediction error bounds and the control parameters.

If, in the control laws above, we use on the master side $x_m(t), f_m(t)$ instead of $\hat{x}_m(t), \hat{f}_m(t)$, we obtain a succession of blocks in an open-loop connection. Thus, through the design process following the classic thinking, the mechanical part reduced to an open-loop system. This was due to the use of NI and a perfect predictor, and indicates that the master-slave-master loop becomes an obsolete specification. Under non perfect prediction or if online adaptation of the model is needed, the enhanced scheme of this subsection remains closed loop.

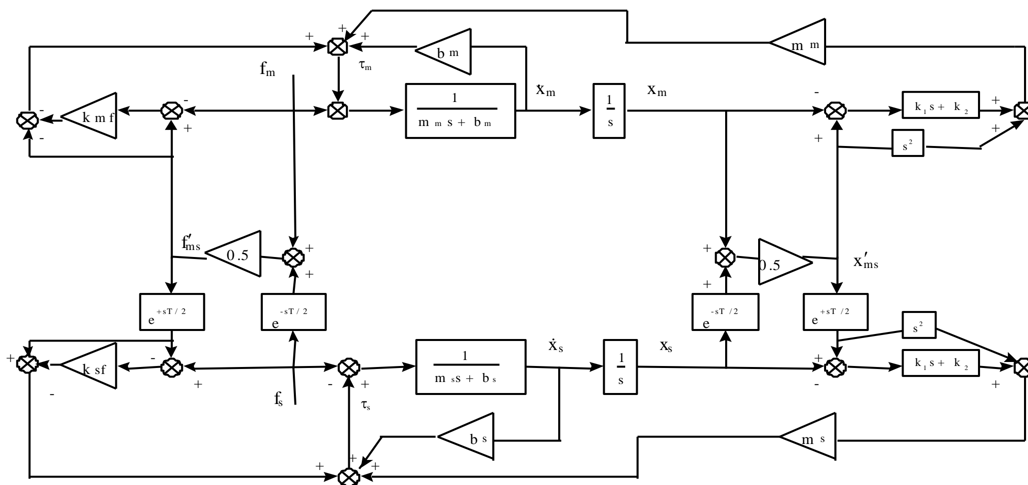
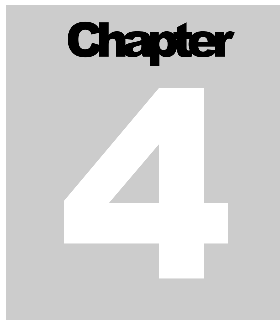


Fig. 3.4: The enhanced Yokokohji and Yoshikawa telemanipulation scheme



IMPLEMENTATION OF PROPOSED METHOD

4.1 Introduction

In this chapter the design work followed for the implementation of the variable-time-delays-robust telemanipulation through master state prediction (described in chapter 3) is presented. The program was developed and debugged using *Microsoft Visual C++ 6.0* part of *Microsoft Visual Studio 6.0.* Great attention was paid to the explanation of functions used during the program and the theory involved behind them. Two different functions for predicting the master state were investigated and developed, according to *interpolation using Lagrance polynomial*, *and polynomial least squares curve fitting* theories.

4.2 Program Explanation

The program accepts a script file, where the user is called to enter the desired simulation parameters. The two main functions of the program are the master and

slave. When the simulation is over a log file is created containing all the results (at each sample) that occurred during the execution of the algorithm.

In order to simulate a random delay line between master and slave, the master information (at every sample), are first written in the hard disk, and then red from the slave and vice versa, adding this way the resulted write-read time of hard disk (hard disk access time). The delay feature is neutral in the current program (master and slave are executed under the same counter) and was only developed for future expansion of the program, where master and slave would be using independent execution counters.

4.3 Interpolation

4.3.1 Interpolation Theory

In this section the problem of obtaining a function for the case when the data points are precisely known will be addressed. In the case of interpolation, the curve has to pass through every data point. The resulting polynomial is called an *interpolating polynomial*, and the process of obtaining intermediate points between precise known points is called *interpolation*. The most common use of interpolation is to obtain intermediate values from tabulated data.

As a first approximation the data points can be connected by a series of straight lines. Figure 4.1 on next page, shows one such a segment connecting x_i and x_{i+1}.

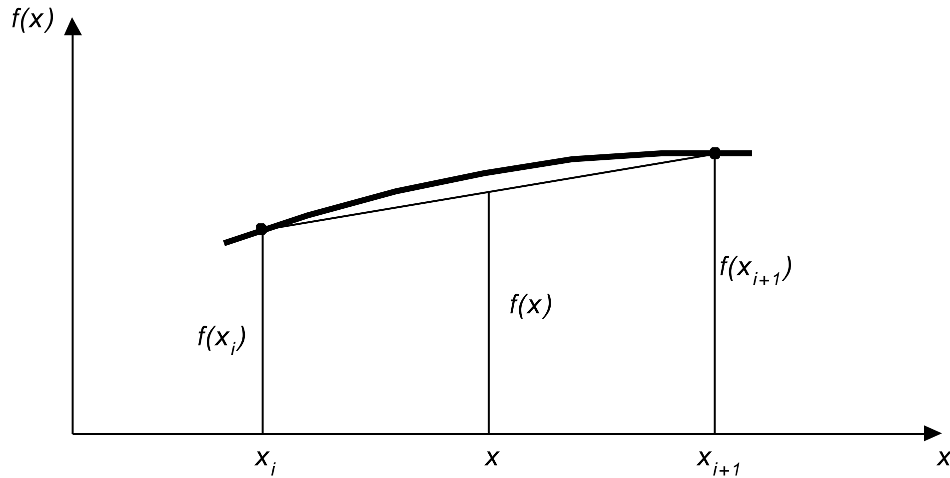


Fig. 4.1: Linear Interpolation [31]

The equation of the straight line of 1st degree interpolating function can be written as:

$$f(x) = a_0 + a_1 x \quad (36)$$

Considering the straight line segment between the points x_i and x_{i+1}, the coefficients a_0 and a_1 can be obtained by noting that the function $f(x)$ must pass through the points x_i and x_{i+1}:

$$f(x_{i+1}) = a_0 + a_1 x_{i+1} \quad (36)$$

Solving the above simultaneous equations the following values for a_0 and a_1:

$$a_0 = \frac{x_{i+1} f(x_i) - x_i f(x_{i+1})}{x_{i+1} - x_i} \quad (37)$$

$$a_0 = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \quad (38)$$

Substituting for a_0 and a_1 in Eq. (36) and rearranging:

$$f(x) = \frac{x - x_{i+1}}{x_i - x_{i+1}} f(x_i) + \frac{x - x_i}{x_{i+1} - x_i} f(x - x_i) \quad (39)$$

Equation (39) can be used for estimating the value of $f(x)$ corresponding to a given value of x. The particular form of the equation is called the *Lagrance 1st order interpolating polynomial*.

The accuracy of the approximation can be improved by presenting some curvature in the function connecting the points $(x_0,f(x_0)),\ldots,(x_n,f(x_n))$. This can be done by approximating $f(x)$ with a 2nd degree interpolating function:

$$f(x) = a_0+a_1x+a_2x^2 \tag{40}$$

Similarly as above, the equation for a 3rd order degree polynomial is:

$$f(x) = a_0+a_1x+a_2x^2+ a_3x^3 \tag{41}$$

The coefficients a_0, a_1, a_3 are determined by requiring that $f(x)$ pass through the points $(x_0,f(x_0))$, $(x_1,f(x_1))$, $(x_2, f(x_2))$, and $(x_3, f(x_3))$, which yields the following equation:

$$f(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} f(x_0) + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} f(x_1)$$

$$+ \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} f(x_3) \tag{42}$$

The Lagrance interpolating polynomial of degree n can be written as:

$$f(x)=L_0(x)f(x_0)+L_1f(x_1)+L_2f(x_2)+\ldots+L_nf(x_n) \tag{43}$$

or more simply as:

$$f(x) = \sum_{i=0}^{n} L_1(x) f(x_i) \tag{44}$$

The functions $L_i(x)$ are defined as:

$$L_i(x) = \frac{(x-x_0)(x-x_1)(x-x_2)\ldots(x-x_{n-1})(x-x_n)}{(x_i-x_0)(x_i-x_1)\ldots(x_i-x_{i-1})(x_i-x_{i+1})\ldots(x_i-x_n)} \tag{45}$$

or more simply as:

$$L_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \left(\frac{x-x_i}{x_i-x_j} \right) \tag{46}$$

where the Greek capital letter Π represents a repeated product. Replacing $L_i(x)$ from Eq. (46) in Eq. (44), the Lagrance interpolation function can be written as:

$$f(x) = \sum_{i=0}^{n} \left(\prod_{\substack{j=0 \\ j \neq 0}}^{n} \frac{x - x_j}{x_i - x_j} \right) \cdot f(x_i) \qquad (47)$$

The functions $L_i(x)$ have the following properties:

$L_i(x)=1$, at $x=x_i$

$L_i(x)=0$, at $x \neq x_i$

Thus the product $L_i(x)f(x_i)$ is equal to $f(x_i)$ at $x=xi$ and is zero for all other values of xi. This means that the polynomial passes exactly trough each of the $n+1$ points [31].

4.3.2 Interpolation Function

Function **lagrance_poly()** performs interpolation using an nth order Lagrance interpolating polynomial. It determines a polynomial that passes through a set of $n+1$ data points, $(x_0, f(x_0))$, and computes the value of the dependent variable for a given x value. The function accepts as input parameters:

- **x[]** array containing values of independent variable x_i
- **y[]** array containing values of independent variable $f(x_i)$
- **n** number of data points
- **x_value** x_value for interpolation

and returns:

- **fx** value of dependent variable $f(x)$ at $x=x_value$

The code for **lagrance_poly**() is very simple and consists of two nested **for** loops. The inner loop computes $L_i(x)$, while the outer loop computes the sum of $L_i(x)f(x_i)$. The function returns a value of type **double** in the variable **fx**, which is the interpolated value of $f(x)$ at the given value of x [31].

4.3.3 Pseudo Code for Function, *lagrance_poly()*

The pseudo code for the function is shown below.

```
Declare fx as double
Declare x, f as double arrays of size defined in variable MAXSIZE
Declare n as integer
Declare x_value as double

Declare loop counters i, j as integers
Declare fx, as double and set to 0.0
Declare li, as double and set to 1.0

Start incremental counter for i=0 to i< n
Set li equal to 1.0
Start incremental counter for j=0 to i< n
If j is not equal to i then
Compute li
Compute sum of li*f[i] and store in variable fx
Return variable fx
```

4.4 Polynomial Least Squares Curve Fitting

4.4.1 Polynomial Least Squares Curve Fitting Theory

In this section the problem of developing a curve that follows the general trend of the data and passes as close as possible but not necessarily through every data point, is presented.

The most widely used technique for fitting a line through a series of observed data points is the least squares method. This method is based on minimizing the sum of the squares of the difference between the observed data points and the values given by the approximating line. The method is illustrated graphically in Fig. 4.2.

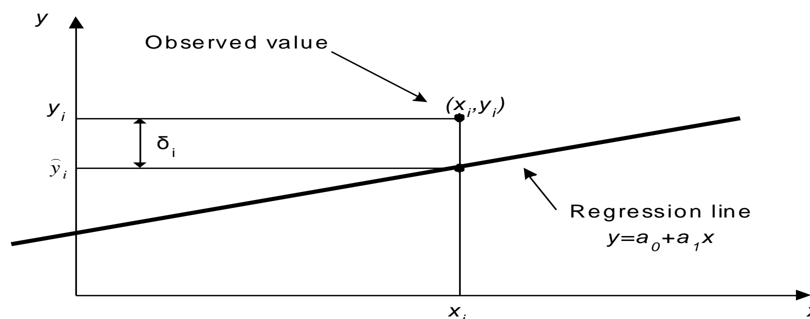


Fig. 4.2: Regression line and error associated with point (x_i, y_i) [31]

The predicted values are given by:

$$\hat{y}_i = a_0 + a_1 x_i \tag{48}$$

The vertical deviating δ_i of the *i*th point from the regression line is:

$$\delta_i = y_i - \hat{y}_i = y_i - (a_0 + a_1 x_i) \tag{49}$$

where δ_i is the difference between the observed value y_i and the ordinate $\hat{y}_i$ of the fitting straight line at x_i. The sum of the squares of deviations is:

$$S = \sum_{i=1}^{n} \delta_i^2 = \sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i)]^2 \tag{50}$$

The values of a_0 and a_1 are chosen so as to minimise S. S can be minimised by taking the partial derivatives of S with respect to a_0 and a_1 and setting the resulting equation to zero:

$$\frac{\partial S}{\partial a_0} = -2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i)] = 0 \tag{51}$$

$$\frac{\partial S}{\partial a_0} = -2\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_i)](x_i) = 0 \tag{52}$$

Replacing $\sum_{i=1}^{n} a_0$ with na_0 and $\sum_{i=1}^{n} a_1 x_i$ with $a_1 \sum_{i=1}^{n} x_i$ and rearranging gives the following two simultaneous equations:

$$na_0 + a_1 \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \tag{53}$$

$$a_0 \sum_{i=1}^{n} x_i + a_1 \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} x_i y_i \tag{54}$$

Solving the above simultaneous equations, the solutions of a_0 and a_1 can be obtained.

The method presented above can easily be extended to higher-order polynomials.

An N degree regression polynomial has the form:

$$y_i = a_0 + a_1 x + a_2 x^2 + ... + a_N x \tag{55}$$

the sum of squares of the deviations is given by:

$$S = \sum_{i=1}^{n} (y_i - a_0 - a_1 x_i - a_2 x_i - \ldots - a_N x_i^N)^2 \qquad (56)$$

By setting the partial derivatives, $\frac{\partial S}{\partial a_0}, \frac{\partial S}{\partial a_1}, \ldots, \frac{\partial S}{\partial a_2}$ equal to zero, the following system of *n+1* linear simultaneous equations in the unknown coefficients a_0, a_1, …, a_N are obtained.

$$\begin{aligned} na_0 + a_1 \sum x_i + a_2 \sum x_i^2 + \ldots + a_N \sum x_i^N &= \sum y_i \\ a_0 \sum x_i + a_1 \sum x_i^2 + a_2 \sum x_i^3 + \ldots + a_N \sum x_i^{N+1} &= \sum x_i y_i \\ a_0 \sum x_i^N + a_1 \sum x_i^{N+1} + a_2 \sum x_i^{N+2} + \ldots + a_N \sum x_i^{2N} &= \sum x_i^N y_i \end{aligned} \qquad (57)$$

All the summations are from i=1 to i=n. Equation (57) can be written in a matrix form:

$$\begin{bmatrix} n & \sum x_i & \sum x_i^2 & \sum x_i^3 & \ldots & \sum x_i^N \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \ldots & \sum x_i^{N+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \ldots & \sum x_i^{N+2} \\ . & . & . & . & . & . \\ \sum x_i^N & \sum x_i^{N+1} & \sum x_i^{N+2} & \sum x_i^{N+3} & \ldots & \sum x_i^{N+3} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ a_N \end{bmatrix} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ . \\ \sum x_i^N y_i \end{bmatrix} \qquad (58)$$

When the summations that appear in the coefficient matrix and the right-hand vector have been evaluated, the equations can be solved using a method for solving simultaneous equations [31].

4.4.2 Polynomial Least Squares Curve Fitting Function

Function **poly_leastsqr()** determines the best-fit polynomial of the form $y= a_0 + a_1x + a_2x^2 + \ldots + a^N x^{N-1}$ and computes the coefficients a_0, a_1, a_N of the best-fit polynomial of degree *N-1* for a set of observations (x_1, y_1), (x_2, y_2), ...,(x_m, y_n). The computations are performed by function, *poly_leastsqr().* The function first assembles the square coefficient matrix and the right-hand vector given in Eq. (58) in the previous section. The coefficient matrix is saved in the two-dimensional array **c[][]**, and the right-hand

vector is saved in the one-dimensional array **a[]**. The function then calls the *triangular factorisation with pivoting* elimination routine to solve the system of equations and obtain the coefficients of the best-fit polynomial. The same function also calculates the value of y (predicted value) for a given *x_value*. The elimination function is discussed in the next section. The function expects six arguments, described next:

- **x[]** array containing observed values of x_i
- **y[]** array containing observed values of y_i
- **num_points** number of data points, n
- **num_poly** degree of polynomial, which is equal to $N-1$
- **x_value** x value of polynomial

and returns:

- **a[]** coefficients of best-fit polynomial, $a_0, a_1, \ldots, a_n$

The function creates two local arrays, a one-dimensional array **s[]** and a two-dimensional array **c[][]**. The array **s[]** is used to store various sums that used to create the coefficient matrix. The elements of the array **s[]** are obtained from:

$$s[k] = \sum_{i=1}^{n} x_i^k \tag{59}$$

On the other hand the elements of the coefficient matrix **c[][]** are obtained from the array **s[],** knowing that the relation between them is, **c[i][j]=s[i+j]**.

The right-hand vector is saved in the array a[], the elements of which are obtained from:

$$a[k] = \sum_{i=1}^{n} y_i x_i^k \tag{60}$$

The function first creates the arrays **s[]** and **a[]**. Next it creates the array **c[][]** by placing the elements of **s[]** in their appropriate positions in **c[]** [31]. It then calls the function *Triangular_Factorization()*, to solve the system of equations and returns the value of y (**fx**, predicted value) based on **x_value**.

4.4.3 Pseudo Code for Function, *poly_leastsqr()*

The pseudo code for the function is shown below.

Declare x, y as double arrays of size defined in variable MAXPOINTS
Declare a as double arrays of size defined in variable MAXSIZE

Declare c as 2x2 double array of size defined in variable MAXSIZE
Declare s as double array of size twice the one defined in variable MAXSIZE
Declare loop counters i, j as integers
Declare predicted value fx as double

Compute sums routine
Set s[0] equal to number of data points, num_points
Start incremental counter for i=1, to i<= to twice num_points
Set s[i] equal to 0.0
Start incremental counter for j= 0, to j<num_points

Create coefficient matrix routine
Start incremental counter for i=0, to i<= to degree of polynomial, num_poly
Start incremental counter for j=0, to i<= to num_poly
Set the relation between the elements of c[][] and s[], c[i][j]=s[i+j]

Create right-hand side vector routine
Set a[0] equal to 0.0
Start incremental counter for j=0 to j<num_points
Compute sums of y[j] and store in a[0]
Start incremental counter for i=0 to i<=num_poly
Set a[i] to 0.0
Start incremental counter for j=0 to j<num_points
Compute the product of y[j] with x[j] in power of I, add to previous result and store in a[i]

Call Triangular_factorisation() elimination faction, with input parameters,
c[], a[], num_poly+1, x_value, a[]

4.5 Triangular Factorisation

4.5.1 Triangular Factorisation Theory

A system of linear simultaneous equations is usually represented by the form:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_2 x_1 & + & \dots & + & a_{2n}x_n & = & b_2 \\ . & & . & & . & & . & . & . \\ . & & . & & . & & . & . & . \\ . & & . & & . & & . & . & . \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \dots & + & a_{nn}x_n & = & b_n \end{array} \quad (61)$$

In the above form, a_{ij} are known coefficients, b_i are known constants, and x_i are the unknowns for which the equations are to be solved. The unknown x_i's are raised only to the 1st power and do not multiply each other. Therefore, each equation is linear.

The system of equations above can be represented in a matrix form as:

$$\begin{bmatrix} a_{11} & a_{11} & \dots & \dots & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \dots & \dots & \dots & a_{2n} \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ a_{n1} & a_{n2} & \dots & \dots & \dots & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix} \qquad (62)$$

or simply using vectors as:

$$\mathbf{Ax=B} \qquad (63)$$

Where **A** represents the square array of coefficients a_{ij} and is known as the coefficient matrix, x represents the n component matrix of unknowns x_i, and **B** is the column matrix of the right-hand side constants b_i.

Assuming that the matrix A can be written as a product of three matrices, as:

$$\mathbf{PA=LU} \qquad (64)$$

Where **L** is the *lower triangular* matrix (has elements only on the diagonal and below), **U** is the *upper triangular* matrix (has elements only on the diagonal and above), and **P** is the permutation matrix.

Decomposition can be used to solve the linear set:

$$\mathbf{Ax=(LU)x=L(Ux)=PB} \qquad (65)$$

By first solving for the vector y such that:

$$\mathbf{Ly=PB} \qquad (66)$$

and then solving:

$$\mathbf{Ux=y} \tag{67}$$

Equation (66) can be solved by *forward substitution* as follows:

$$\begin{aligned} y_1 &= \frac{\beta_1}{a_{11}} \\ y_i &= \frac{1}{a_{ii}}\left[\beta_i - \sum_{j=1}^{i-1} a_{ij} y_j\right] \rightarrow i = 2,3,..., N \end{aligned} \tag{68}$$

while Eq.(67) can be solved by *backsubstitution* as follows:

$$\begin{aligned} x_N &= \frac{y_N}{b_{NN}} \\ y_i &= \frac{1}{b_{ii}}\left[y_i - \sum_{j=i+1}^{N} b_{ij} x_j\right] \rightarrow i = N-1, N-2,..., 1 \end{aligned} \tag{69}$$

4.5.2 Triangular Factorisation with Pivoting Function

Function **Triangular_factorisation()** (**PA=LU** Factorisation with Pivoting), solves the linear system **Ax = B** by performing the following steps:

- Computes **PB** and forms the equivalent linear system **LUx=PB**
- Solves the lower-triangular system **Ly=PB** for *y*.
- Solves the upper-triangular system **Ux=y** for *x*.

Then it performs forward and back substitution according to equations (68), (69) in order to find the solutions to system of equations. Finally the solution matrix ($a_0,…,a_n$) is used to determine the predicted value *y* (**fx**) according to *x* (**x_value**).

4.5.3 Pseudo Code for Function, Triangular_factorisation()

```
Declare loop counters i, ii, z, j, l as integers
Declare field with row number, Row, as double array of size defined in variable MAXSIZE
Declare temporary variable for storing intermediate values, temp, as integer
Declare y as double array of size defined in variable MAXSIZE
Declare adder variable, SUM, as double
Declare determinant of [A], DET, as variable and set equal to 1.0
Declare predicted value fx as double

Initialise pointer vector routine
Start incremental counter for l=1 to l<=degree of polynomial, num_poly
Set Row[l-1]=l-1
```

Start LU factorisation routine
Start incremental counter for z=1 to z<=n_poly-1
Start pivot element routine
Start incremental counter for i=z+1 to i<=n_poly
If absolute value of A[Row[i-1]][z-1] greater than absolute value of A[Row[z-1]][z-1]
Switch the index for the p-1 pivot row if necessary
Set temp=Row[z-1]
Set Row[z-1]=Row[i-1]
Set Row[i-1]=temp
Set DET=-DET
Simulated row interchange ends here

If A[Row[z-1][z-1] equal to 0 then
Prompt "The matrix is singular! Cannot use algorithm to solve the system of equations Ax=B"

Multiply the diagonal elements routine
Set DET=DET*A[Row[z-1][z-1]

Form multiplier routine
Start incremental counter for i=z+1 to i<=n_poly
Set A[Row[i-1]][z-1]=A[Row[i-1]][z-1]/A[Row[z-1]][z-1]

Eliminate p-1, routine
Start incremental counter for j=z+1 to j<=n_poly
Set A[Row[i-1]][j-1]-=A[Row[i-1]][z-1]*A[Row[z-1]][j-1]

LU factorisation routine ends here

Set DET=DET*A[Row[n_poly-1][n_poly-1]

Start forward substitution routine
Set y[0] equal to B[Row[0]]
Start incremental counter for i=2 to i<=n_poly
Set SUM equal to 0
Start incremental counter for j=1 to j<I-1
Compute product of A[Row[I-1]][j-1] and y[j-1] add to previous value and store in SUM
Set y[i-1] equal to B[Row[i-1]]-SUM

If A[Row[n_poly-]][[n_poly] equal to 0 then
Prompt "The matrix is singular! Cannot use algorithm to solve the system of equations Ax=B"
Forward substitution routine ends here

Start back substitution routine
Set a[n_poly-1] equal to y[n_poly-1]/A[Row[n_poly-1]][n_poly-1]
Start decrement counter for i=n_poly-1 to i>=1
Set SUM equal to 0
Start incremental counter for j=i+1 to j<=n_poly
Compute product of A[Row[i-1]][j-1] and a[j-1] add to the previous value and store in SUM

Set a[i-1] equal to (y[i-1]-sum)/A[Row[i-1]][i-1]
Back substitution routine ends here

Compute predicted value fx, routine
Start incremental counter for ii=0 to ii<=n_poly
Set fx equal to 0.0
Compute product of a[ii] and x_value to the power of ii, add to previous value and store in SUM
Return value of fx

4.6 Master Section

4.6.1 Master Theory

The theory behind the master was analytically discussed in chapter 3, where the proposed method was presented. The master function consists of four main routines namely as follows:

- Neural input routine
- Read data files routine
- Master algorithm
- Create data files routine

A detailed description of above routines is presented in the next sections.

4.6.2 Neural Input Routine

The neural input routine generates the neural input that drives the master. For simplicity reasons, neural input was selected to be a sine wave of constant frequency and unity amplitude [$\sin(2\pi ft)$]. The time duration of the sine wave was defined as the product of number of samples by the sampling frequency [$t=nTs$].

4.6.3 Read Data Files Routine

The purpose of read data file routine is to read the slave force and position as well as the sample number for every sample till the simulation is over, and pass the data to the master algorithm. The data files read by the routine are of the form *datan.dat*, where *n* is the sample number. The functions *strcpy* and *strcat* included in string.h include file are used in order to build the above data file form, by coping a string to a specified data location (strcpy) and adding more strings to it (strcat). Function *_itoa* included in stdlib.h include file was used to convert the current sample number (integer) to a

character string, in order to attach it to the data file form. Since it is not possible to save double numbers (limit to float numbers) in a data file, the numbers before written into the data file are converted and stored as strings. Then when read back, are converted back from string to double, without loss of information. This is achieved by using the function *_itof* (converts a string to double) included in stdlib.h.

4.6.4 Master Algorithm

The master-slave force and position data are computed for two cases. The first case is while the number of samples (*n*) is less or equal to half of the given time delay ($T_t/2$ expressed in msec). The computations are based on Eqs. (28), (29) for $T_t=0$ (or *Tp*). The second case arises when the number of samples is greater to $T_t/2$. Again the computation of master-slave force and position is based on Eqs. (28), (29) for the given T_t. To make things more clear, in the first case T_t is set to zero since there is not prediction (prediction model is discussed in slave section) of slave force and position (f_s, x_s) previous values, thought in the second case where $n>T_t$ the prediction has started, therefore previous values of (f_s, x_s) can be obtained.

In order to obtain master-slave velocity and acceleration from master-slave position, a differentiation method needs to be applied. An easy and simplified method to differentiate arises from the definition of differentiation and can be expressed as:

$$\dot{x} = \frac{x - x_0}{T_s} \text{ and } \ddot{x} = \frac{\dot{x} - \dot{x}_0}{T_s} \qquad (70)$$

where T_s needs to be small enough to reduce errors.

Euler method for solving a differential equation is applied in order to find master position, velocity, and acceleration. Equation (30) is solved for $\ddot{x}$ and becomes:

$$\ddot{x}_m = \frac{\tau_{m_0} + f_{m_0} - b_m \dot{x}_{m_0}}{m_m} \tag{71}$$

Then according to equation (70), $\dot{x}$, x can be found as well:

$$\dot{x}_m = \dot{x}_{m_0} + T_s \ddot{x}_m \tag{72}$$

$$x_m = x_{m_0} + T_s \dot{x}_m \tag{73}$$

Finally from the control law equation for master actuator driving force, Eq. (32), τ_m can be obtained.

4.6.5 Write Data Files Routine

The structure of *write data files* routine is more or less the same as of the *read data files* routine described in section 4.6.3. The specific routine is used for storing the master force and position as well as the sample number in the data file. The main difference is that function *_gcvt* (stdlib.h include file) is used for converting double number to string. Since the function requires the number of decimal points to be converted, double numbers considered having 15 decimal points (16 is the maximum for double by definition). Therefore the truncation error is negligible.

4.6.6 Pseudo Code for Function, Master()

Define time duration = no. of samples * sampling frequency

Neural input (master force) simulation
Define rad=2* π(pi[4])*frequency(fi[3])*time
Master force, fm=sin(rad)

If number of samples, n>1

Start read data files routine
Convert n+1 (integer) to character, no2
Copy path for slave data files[4] to filename2
Add slave filename for data files to filename2
Add current no. sample to filename2
Add slave extension for data files to filename2
Open filename2 (slave data file) for reading only
Read slave force information stored in data file, buffer_fs
Read slave position information stored in data file, buffer_xs
Read current sample number stored in data file, n
Convert string to double, buffer_fs to fs[n]
Convert string to double, buffer_xs to xs[n]
Close file

If no. of sample, n<=delay, Tt[4]
Start master algorithm routine
Master-slave position calculation, xms[n]=(xm[n]+xs[n])/2
Master-slave force calculation, fms[n]=(fm[n]+fs[n])/2

If n>Tt/2
Master-slave position calculation, xms[n]=(xm[n]+xs[n-Tt/2])/2
Master-slave force calculation, fms[n]=(fm[n]+fs[n-Tt/2])/2

Perform classic differentiation to find xxm (master velocity), xxxm (master acceleration)
xxms[n]=(xms[n]-xms[n-1])/Ts
xxxms[n]=(xxms[n]-xxms[n-1])/Ts

Control law equation for master actuator driving force calculation, Eq. (32)
tm[n]=mm[4](xxxms[n]+k1[4]*(xxms[n]-xxm[n])+k2[4](xms[n]-xm[n]))+
+bm[4]*xxm[n]-kmf[4](fms[n]-fm[n])-fms[n]

Euler method for finding xxxm, xxm, xm
xxxm[n+1]=(tm[n]+fm[n]-(bm*xxm[n]))/mm
xxm[n+1]=xxm[n]+(Ts[4]*xxxm[n+1])
xm[n+1]=xm[n]+(Ts*xxm[n+1])

Start create data files routine
Convert n (integer) to character, no1
Copy path for master data files[4] to filename1
Add master filename for data files to filename1
Add current no. sample to filename1
Add slave extension for data files to filename1
Open filename1 (master data file) for writing only

---

[4] User defined in script.h include file

Convert double to string, fm[n] to buffer_fm
Convert double to string, xm[n] to buffer_xm
Write master force information in data file, buffer_fm
Write master position information in data file, buffer_xm
Write master sample number in data file, n

4.7 Slave Section

4.7.1 Slave Theory

The theory behind the slave was analytically discussed in chapter 3, where the proposed method was presented. The slave function consists of four main routines namely as follows:

- Read data files routine
- Predictor model
- Slave algorithm/Slave dynamics
- Create data files routine

A detailed description of above routines is presented in the next sections.

4.7.2 Read Data Files Routine

The routine is the same as the one described in section 4.6.3, with the only difference that here it is used for reading the master force and position and pass the data to the slave algorithm.

4.7.3 Predictor Model

In sections 4.3 and 4.4 the two predictor methods used were discussed. After a number of tests it proved that the method belonging in section 4.4, gave the most correct results.

The horizon of prediction (*n_start*) or after how many samples the prediction starts, is user defined in script.h include file. If the no. of samples, n is less than *n_start* then the

estimated master force and position for *n+Tt/2* samples will be equal to master force and position of the 1st sample, until the initial condition changes. If *n* is greater than *n_start* then the prediction starts (with *x_value=n+Tt/2*) in order to get the estimated master force and position data (*efm[n+Tt/2], exm[n+Tt/2]*) . Summarising,

- If n<=n_start then there is no prediction (*xm[1], fm[1]*).
- If n>n_start
  - If n>(n_start+np) then perform prediction with data: *xm[n-np]…xm[n]*
  - Else perform prediction with data: *xm[1]…xm[n]*

4.7.4 Slave Algorithm

The master-slave force and position are based on Eqs. (28), (29), but the estimated master force and position data obtained from the predictor model are used instead. To avoid confusion with master algorithm, the master-slave force and position are renamed as *xmss, fmss*. The rest of algorithm follows exactly the same steps as in master algorithm (section 4.6.4), with the only difference that Eq. (33) is used for computing slave actuator driving force, τ_s. For simplicity reason slave force *fs* was set to zero, assuming no collision of slave robot with any object.

4.7.5 Write Data Files Routine

The routine is exactly the same as the one described in section 4.6.5, with the only difference that here it is used for writing the slave force and position, f_s, x_s to the data file.

4.7.6 Pseudo Code for Function, Slave()

```
Start read data files routine
Convert n (integer) to character, no1
Copy path for master data files to filename1
Add master filename for data files to filename1
Add current no. sample to filename1
Add master extension for data files to filename1
```

Open filename1 (master data file) for reading only
Read master force information stored in data file, buffer_fm
Read master position information stored in data file, buffer_xm
Read current sample number stored in data file, n
Convert string to double, buffer_fm to fm[n]
Convert string to double, buffer_xm to xm[n]
Close file

Start predictor model
Collect no. of samples for prediction function and save in st[n], st[n]=n
Collect current master position for prediction function and save in sx[n], sx[n]=xm[n]
Collect current master force for prediction function and save in sf[n], sf[n]=fm[n]

If n<=prediction horizon, n_start

Set predicted master position, exm[n+Tt/2]=xm[1]
Set predicted master force, efm[n+Tt/2]=fm[1]

If n>prediction horizon, n_start
Set x_value equal tp n+Tt/2
Call curve-fitting function for efm, fx1=poly_leastsqr(st,sf,np,2,x_value,a)
Save predicted value fx1 into efm[n+Tt/2]

Call curve-fitting function for exm, fx2=poly_leastsqr(st,sx,np,2,x_value,a)
Save predicted value fx2 into exm[n+Tt/2]

Start slave algorithm/slave dynamics
Master-slave position calculation, xmss[n]=(exm[n]+xs[n])/2
Master-slave force calculation, fmss[n]=(efm[n]+fs[n])/2

Perform classic differentiation to find xxs (slave velocity), xxxs (slave acceleration)
xxmss[n]=(xmss[n]-xmss[n-1])/Ts
xxxmss[n]=(xxmss[n]-xxmss[n-1])/Ts

Control law equation for slave actuator driving force calculation, Eq. (32)
ts[n]=bs[5]*xxs[n]-ksf[5]*(fs[n]-fmss[n])+fmss[n]+
+ms[5]*(xxxmss[n]+k1[5]*(xxmss[n]-xxs[n]+k2[5]*(xmss[n]-xs[n])

Euler method for finding xxxs, xxs, xs
xxxs[n+1]=(ts[n]+fs[n]-(bs*xxs[n]))/ms
xxs[n+1]=xxs[n]+(Ts*xxxs[n+1])
xs[n+1]=xs[n]+(Ts*xxs[n+1])

Start create data files routine
Convert n+1 (integer) to character, no2
Copy path for slave data files to filename2
Add slave filename for data files to filename2
Add current no. sample to filename2
Add slave extension for data files to filename2
Open filename2 (master data file) for writing only
Convert double to string, fs[n+1] to buffer_fs
Convert double to string, xs[n+1] to buffer_xs
Write slave force information in data file
Write slave position information in data file
Write slave sample number+1 in data file

[5] User defined in script.h include file

4.8 Simulation Logfile Section

4.8.1 Logfile Creation Function

After the simulation has finished a number of individual master and slave data files have been created, containing the simulation values of master and slave force and position during the simulation run. Function *logfile_create* reads those files one by one (read data files routine for master and slave, sections 4.63, 4.7.2, with no conversion from string to double), collects all data contained in them and saves them in a log file. Other interesting simulation results, such as the master-slave force and position (*fmss, xmss*), predicted master force and position (*efm, exm*), master and slave actuator driving forces (τ_m, t_s), remain in the computer memory after the simulation is over, and can be saved in the log file as well. The simulation elapsed time is saved as well and added at the bottom end of the logfile. If another simulation is run by the user, the new results are simply added in the logfile without replacing the old ones. This way the user can compare the old results with the new ones. The logfile structure was chosen to be as described before, since it can be directly manipulated by *Matlab* without modification, for further analysis of the results.

4.8.2 Pseudo Code for Function, *logfile_create()*

Open logfile. Perform a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the function.

Store heading description on first row of simulation data to be stored, “n”, “xm”, “fm”, “xs”, “fs”, “fmss”, “xmss”, “exm”,”efm”,“tm”, “ts”

Run loop for n=1 to n<=no. of samples, k[6]

Start read master data files routine
Convert n (integer) to character, no1
Copy path for master data files to filename1
Add master filename for data files to filename1
Add current no. sample to filename1

[6] User defined in script.h include file

Add master extension for data files to filename1
Open filename1 (master data file) for reading only
Read master force information stored in data file, buffer_fm
Read master position information stored in data file, buffer_xm
Read current sample number stored in data file, n
Close file

Start read slave data files routine
Convert n+1 (integer) to character, no2
Copy path for slave data files to filename2
Add slave filename for data files to filename2
Add current no. sample to filename2
Add slave extension for data files to filename2
Open filename2 (slave data file) for reading only
Read slave force information stored in data file, buffer_fs
Read slave position information stored in data file, buffer_xs
Read current sample number stored in data file, n
Close file

Convert double to string, fmss[n] to buffer_ fmss
Convert double to string, xmss[n] to buffer_ xmss
Convert double to string, exm[n] to buffer_ exm
Convert double to string, efm[n] to buffer_ efm
Convert double to string, tm[n] to buffer_ tm
Convert double to string, ts[n] to buffer_ ts
Write in log file, all values of n, buffer_xm, buffer_fm, buffer_xs, buffer_fs, buffer_fmss, buffer_xmss, buffer_efm, buffer_exm, buffer_tm, buffer_ts during the simulation

Write in logfile the simulation elapsed time, stored in variable duration

4.9 Simulation Elapsed Time Section

4.9.1 Elapsed Time Start and Finish Functions

Two functions were developed for measuring the duration of the simulation. The first function starts the system clock and the second function stops the system clock and estimates the elapsed time (duration). The functions are inserted before and after the task to be measured. Both of the functions use the *time.h* include file.

4.9.2 Pseudo Code for Function, *elapsed_time_start()*

Start system clock, start=clock()

4.9.3 Pseudo Code for Function, *elapsed_time_finish()*

Stop system clock, finish=clock()
Estimate elapsed time, duration=(double)(finish-start)/CLOCKS_PER_SEC
Prompt “Simulation elapsed time:”

4.10 Program Variable Initialisation Section

4.10.1 Initialisation Function

Function *init()* was developed to simply initialise the different variables used throughout the program code. The values used were obtained from feedback information based on similar simulation tests.

4.10.2 Pseudo Code for Function, *Init()*

Set master position at n=1, xm[1]=1.2217 rad or 7°
Set estimated master position at n=1, xm[1]=1.2217 rad or 7°
Set master velocity at n=1, xxm[1]=0.0
Set master acceleration at n=1, xxxm[1]=0.0
Set slave position at n=1, xm[1]=1.2217 rad or 7°
Set slave velocity at n=1, xxm[1]=0.0
Set slave acceleartion at n=1, xxm[1]=0.0
Set master-slave velocity at n=1, xxms[1]=0.0
Set master-slave acceleration at n=1, xxxms[1]=0.0
Set slave force at n=1, fs[1]=0.0assume no collision
Set master-slave position at n=0, xms[0]=(xm[1]+xs[1])/2
Set master-slave velocity at n=0, xxms[1]= xxms[0]

Set master-slave position at n=0, xmss[0]=(exm[1]+xs[1])/2
Set mster-slave velocity at n=1, xxmss[1]=0.0
Set master-slave velocity at n=0, xxmss[0]=xxmss[1]
Set master-slave acceleration at n=1, xxxmss[1]=0.0

4.11 Include Files and Declaration of Global Variables

The list of include files used in the program code are given below:

- stdio.h stdlib.h
- iostream.h, iomanip.h
- math.h
- string.h
- time.h
- script3.h

Include file *script3.h* holds user-defined parameters for the program simulation. It is further discussed in the next section.

All global variable definitions used, are described analytically in the table below.

Variable name		*Data Type*		*Explanation*
rad		double		Angle variable
Timing variables				
t=0		double		Time
n=0		integer		Program counter
start		clock_t	Defined in time.h	Start system clock
finish		clock_t		Stop system clock
duration		double		Simulation duration
Master variables				
tm[k]	k defined in script3.h	double		Master actuator driving force
xm[k]	k defined in script3.h	double		Master position
xxm[k]	k defined in script3.h	double		Master velocity
xxxm[k]	k defined in script3.h	double		Master acceleration
fm[k]	k defined in script3.h	double		Master force
Slave variables				
ts[k]	k defined in script3.h	double		Slave actuator driving force
xs[k]	k defined in script3.h	double		Slave position
xxs[k]	k defined in script3.h	double		Slave velocity
xxxs[k]	k defined in script3.h	double		Slave acceleration
fs[k]	k defined in script3.h	double		Slave force
Master-Slave variables				
xms[k]	k defined in script3.h	double		Master-slave position
xxms[k]	k defined in script3.h	double		Master-slave velocity
xxxms[k]	k defined in script3.h	double		Master-slave acceleration
fms[k]	k defined in script3.h	double		Master-slave acceleration
Slave dynamics variables				
xmss[k]	k defined in script3.h	double		Master-slave position
xxmss[k]	k defined in script3.h	double		Master-slave velocity
xxxmss[k]	k defined in script3.h	double		Master-slave acceleration
fmss[k]	k defined in script3.h	double		Master-slave force
Master data files variables				
no1[10]		char		Data file number
filename1[80]		char		Filename of data file
buffer_fm[50]		char		Master force converted to string
buffer_xm[50]		char		Master force converted to string
Slave data files variables				
no2[10]		char		Data file number
filename2[80]		char		Filename of data file
buffer_fs[50]		char		Slave force converted to string
buffer_xs[50]		char		Slave force converted to string
Log file Creation Variables				
buffer_fmss[50]		char	Master-slave force converted to string	
buffer_xmss[50]		char	Master-slave position converted to string	
buffer_efm[50]		char	Estimated master force converted to string	
buffer_exm[50]		char	Estimated master position converted to string	
buffer_tm[50]		char	Estimated master actuator driving force conv. to string	
buffer_ts[50]		char	Estimated slave actuator driving force conv. to string	

a[MAXSIZE]	MAXSIZE defined in script3.h	double	Polynomial coeffiecients
Curve-fitting variables			
fx1		double	Predicted value for master force
fx2		double	Predicted value for master position
x_value		double	x value of polynomial
Curve-fitting external variables			
st[k]	k defined in script3.h	double	same as n
sf[k]	k defined in script3.h	double	same as xf[n]
sx[k]	k defined in script3.h	double	same as xm[n]
efm[2*k]	k defined in script3.h	double	Predicted master force
exm[2*k]	k defined in script3.h	double	Predicted master position
np=50		integer	Number of data points for prediction

Table 4.1: Global variable declarations

4.12 Simulation Script File Section

A script file in the form of included file from the main program code was created in order to hold the simulation parameters. The user can alter the simulation parameters by simply editing the script file through any text editor. The following table, accounts the availability of user defined simulation parameters, contained in the script file.

Variable name	*Explanation and Units*
Timing Parameters	
Tt	Time delay (msec)
Ts	Sampling frequency (sec)
fi	Frequency of sinusoidal i/p signal (Hz)
k	Number of samples
Human arm parameters	
mo	Mass (kgr)
bo	Damping coefficient (N/m^2)
ko	Spring constant (N/m)
Master parameters	
mm	Mass (kgr)
bm	Damping coefficient (N/m^2)
km	Spring constant (N/m)
Slave parameters	
ms	Mass (kgr)
bs	Damping coefficient (N/m^2)
ks	Spring constant (N/m)
Control Parameters	
kmf	Spring constant (N/m)
ksf	Spring constant (N/m)
k1	Spring constant (N/m)
k2	Spring constant (N/m)
Definition of π (pi)	
pi	π (pi)

Curve Fitting Parameters (predictor)	
n_start	Prediction staring point (prediction horizon)
MAXSIZE	Maximum degree of polynomial
MAXPOINTS	Max number of data points for prediction
Parameters for Master data files	
path_master	Master data file path
file_name_master	Master datafile filename
extension_master	Master datafile extension
Parameters for Slave data files	
path_slave	Slave data file path
file_name_slave	Slave datafile filename
extension_slave	Slave datafile extension
Logfile Parameters	
log_file	Logfile path, name, and extension

Table 4.2: Script file parameters

4.13 Main Section

4.13.1 Main Function

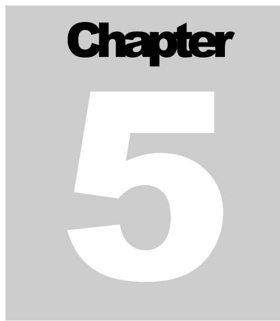
The main function simply calls the functions described in the sections above. Initially the function *init()* is called to initialise the program variables. Next, the function *elapsed_time_start()* is called to measure the simulation time. An external counter loop containing the master and slave functions is set for the simulation purposes. The function *elapsed_time_finish()* is called after the simulation has finished to return the simulation elapsed time. Finally, function *logfile_create()* is called to create a logfile containing the simulation results.

4.13.2 Pseudo Code for Function, *main()*

Call function init() to initialise variables
Call function elapsed_time_start() to start the system clock

Start external counter loop for n=1 to n<=k (defined in script.h)
Call function master() for master model
Call function slave() for slave model

Call function elapsed_time_finish() to stop the system clock and return the simulation elapsed time
Call function logfile_create to save simulation results.



SOURCE CODE

5.1 Introduction

In this chapter the source code of the programs is presented. The source code was developed and debugged using *Microsoft Visual C++ 6.0* part of *Microsoft Visual Studio 6.0.* Section 5.2 shows the source code of the main program, while the following section shows the source code for the script file. Finally in section 5.3 the source code for the interpolation function is given.

5.2 Source Code of Main Program

```
#include <stdio.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "script.h"            //Holds required simulation parameters



//______________global variable declarations______________

        //angle variable
        double rad;                             //angle variable

        //timing variables
        double t=0;                 //time or simulation duration
        int n=0;                    //master counter

        clock_t start, finish;      //start & stop system clock
        double duration;            //simulation duration

        //master variables
        double tm[k];               //master actuator driving force
        double xm[k];               //master position
        double xxm[k];              //master velocity
        double xxxm[k];             //master acceleration
        double fm[k];               //master force

        //slave variables
        double ts[k];               //slave actuator driving force
        double xs[k];               //slave potition
        double xxs[k];              //slave velocity
        double xxxs[k];             //slave acceleration
        double fs[k];               //slave force

        //master-slave variables
        double xms[k];              //master-slave position
        double xxms[k];             //master-slave velocity
        double xxxms[k]; //master-slave acceleration
        double fms[k];              //master-slave force

        //slave dynamics variables
        double xmss[k];             //master-slave position
        double xxmss[k]; //master-slave velocity
        double xxxmss[k];           //master-slave acceleration
        double fmss[k];             //master-slave force

        //variables for master data files
        char no1[10];               //data file number
        char filename1[80];         //filename of data file
        char buffer_fm[50];         //master force converted to string
        char buffer_xm[50];         //master position converted to string
```

```
	//variables for slave data files
	char no2[10];			//data file number
	char filename2[80];		//filename of data file
	char buffer_fs[50];		//slave force converted to string
	char buffer_xs[50];		//slave position converted to string


	//variables for logfile_create function
	char buffer_fmss[50];		//master-slave force converted to string
	char buffer_xmss[50];		//master-slave position converted to string
	char buffer_efm[50];		//estimated master force converted to string
	char buffer_exm[50];		//estimated slave position converted to string
	char buffer_tm[50];		//estimated master actuator driving force conv. to string
	char buffer_ts[50];		//estimated slave actuator driving force conv. to string

	//variables for curve-fitting function
	double a[MAXSIZE];		//polynomial coefficients
	double fx1;			//predicted value for master force
	double fx2;			//predicted value for master position
	double x_value;			//x value of polynomial

	//external variables for curve-fitting function
	double st[k];			//same as k
	double sx[k];			//same as xf[k]
	double sf[k];			//same as xm[k]
	double efm[2*k];		//estimated master force
	double exm[2*k];		//estimated mster position
	int np=50;			//number of data points for prediction


void init(void);
void main(void);
void master(void);
void slave(void);
void logfile_create(void);
void elapsed_time_start(void);
void elapsed_time_finish(void);

double poly_leastsqr(double x[],double y[],int num_points,int num_poly,double x_value,double a[]);
double Triangular_Factorization(double A[][MAXSIZE],double B[],int n,double x_value,double a[]);

/*----------------------------------------------------------------------*/
/* Function: init()							*/
/*									*/
/* Purpose:								*/
/*									*/
/* Initialises variables						*/
/*									*/
/* Returns:								*/
/*	void								*/
/*----------------------------------------------------------------------*/

void init(void)
{

	//initial values (n=1) both master-slave
		xm[1]=1.221700000000000;
		exm[1]=1.221700000000000;
		xxm[1]=0.000000000000000;
		xxxm[1]=0.000000000000000;
```

```
		xs[1]=1.221700000000000;
		xxs[1]=0.000000000000000;
		xxxs[1]=0.000000000000000;
		xxms[1]=0.000000000000000;
		xxxms[1]=0.000000000000000;
		xms[0]=(xm[1]+xs[1])/2;
		xxms[0]=xxms[1];
		fs[1]=0.000000000000000;
		ts[0]=0.000000000000000;

		xmss[0]=(exm[1]+xs[1])/2;
		xxmss[1]=0.000000000000000;
		xxmss[0]=xxmss[1];
		xxxmss[1]=0.000000000000000;
}


/*----------------------------------------------------------------------------------------------*/
/* Function: master()                                                                           */
/*                                                                                              */
/* Purpose:                                                                                     */
/*                                                                                              */
/* Performs simulation of the master model. Reads the data file containing                      */
/* the values of xs and fs and calculates the values for xm and fm and saves                    */
/* them in a data file(for the slave function to read).                                         */
/*                                                                                              */
/* Returns:                                                                                     */
/*          void                                                                                */
/*----------------------------------------------------------------------------------------------*/

void master(void)
{
		//pointers
		t=n*Ts;			//time duration = no. of samples * sampl. freq

		//neural input
		rad=(2*pi*fi*t);	//rad definition
		fm[n]=sin(rad);		//neural input signal

		if (n>1)
		{
			//Read data files routine
			_itoa(n+1,no2,10);		//convert int to char

			strcpy(filename2,path_slave);		//copy string to specified location
			strcat(filename2,file_name_slave);	//add string to specified location
			strcat(filename2,no2);			//add string to specified location
			strcat(filename2,extension_slave);	//add string to specified location

			ifstream slave_data (filename2);	//open file for reading

			slave_data >> buffer_fs;	//read string from file
			slave_data >> buffer_xs;	//read string from file
			slave_data >> n;		//read integer from file

			fs[n]= atof(buffer_fs);		//convert string to double
			xs[n]= atof(buffer_xs);		//convert string to double

			slave_data.close();		//close file
		}
```

```
        if (n<=Tt/2)
        {
                //master-slave position calculation
                xms[n]=(xm[n]+xs[n])/2;

                //master-slave force calculation
                fms[n]=(fm[n]+fs[n])/2;
        }

        if (n>Tt/2)
        {
                //master-slave position calculation
                xms[n]=(xm[n]+xs[n-Tt/2])/2;

                //master-slave force calculation
                fms[n]=(fm[n]+fs[n-Tt/2])/2;
        }

        //Perform classic Differentiation to find xxm, xxxm
        xxms[n]=(xms[n]-xms[n-1])/Ts;
        xxxms[n]=(xxms[n]-xxms[n-1])/Ts;


        //control law eq. for master actuator driving force calculation
        tm[n]=mm*(xxxms[n]+k1*(xxms[n]-xxm[n])+k2*(xms[n]-xm[n]))
                +bm*xxm[n]-kmf*(fms[n]-fm[n])-fms[n];

        //Euler method for finding xxxm,xxm,xm
        xxxm[n+1]=(tm[n]+fm[n]-(bm*xxm[n]))/mm;
        xxm[n+1]=xxm[n]+(Ts*xxxm[n+1]);
        xm[n+1]=xm[n]+(Ts*xxm[n+1]);


        //Create data files routine

        _itoa(n,no1,10); //convert int to char

        strcpy(filename1,path_master);      //copy string to specified location
        strcat(filename1,file_name_master);//add string to specified location
        strcat(filename1,no1);              //add string to specified location
        strcat(filename1,extension_master);//add string to specified location


        ofstream master_data (filename1);   //open file for writing

        _gcvt( fm[n], 15, buffer_fm );      //convert double to string
        _gcvt( xm[n], 15, buffer_xm );      //convert double to string


        master_data << buffer_fm << endl; //write string to file
        master_data << buffer_xm << endl;//write string to file
        master_data << n << endl;         //write integer to file

        master_data.close();        //close file
}
```

```
/*--------------------------------------------------------------------------------------------*/
/* Function: slave()                                                                          */
/*                                                                                            */
/* Purpose:                                                                                   */
/*                                                                                            */
/* Performs simulation of the slave model. Reads the data file containing                    */
/* the values of xm and fm and calculates the values for xs and fs and saves                 */
/* them in a data file(for the master function to read).                                     */
/*                                                                                            */
/* Returns:                                                                                   */
/*          void                                                                              */
/*--------------------------------------------------------------------------------------------*/

void slave(void)
{

        fs[n]=0.000000000000000; //assume no collision

        //Read data files routine

        _itoa(n,no1,10);                        //convert int to char

        strcpy(filename1,path_master);                  //copy string to specified location
        strcat(filename1,file_name_master);             //add string to specified location
        strcat(filename1,no1);                          //add string to specified location
        strcat(filename1,extension_master);             //add string to specified location

        ifstream master_data (filename1);   //open file for reading

        master_data >> buffer_fm;//read string from file
        master_data >> buffer_xm;               //read string from file
        master_data >> n;                       //read string from file

        fm[n]=atof(buffer_fm);                  //convert string to double
        xm[n]=atof(buffer_xm);                  //convert string to double

        master_data.close();          //close file


        //Predictor
        st[n]=n;
        sx[n]=xm[n];
        sf[n]=fm[n];

        if (n<=n_start)
        {
                exm[n+Tt/2]=xm[1];
                efm[n+Tt/2]=fm[1];
        }

        if (n>n_start)
        {
                if (n>(n_start+np))
                {
                        x_value= (n+Tt/2);

                        //Call curve-fitting function for efm

                        fx1 =poly_leastsqr(st,sf,np,2,x_value,a);
                        efm[n+Tt/2]=fx1;
```

```
                //Call curve-fitting function for exm
                fx2 =poly_leastsqr(st,sx,np,2,x_value,a);
                exm[n+Tt/2]=fx2;
        }
        else
        {
                x_value= (n+Tt/2);

                //Call curve-fitting function for efm
                fx1 =poly_leastsqr(st,sf,n,2,x_value,a);
                efm[n+Tt/2]=fx1;

                //Call curve-fitting function for exm
                fx2 =poly_leastsqr(st,sx,n,2,x_value,a);
                exm[n+Tt/2]=fx2;
        }
}

//Slave Dynamics
xmss[n]=(exm[n+Tt/2]+xs[n])/2;
fmss[n]=(efm[n+Tt/2]+fs[n])/2;


//Perform classic Differentiation to find xxmss, xxxmss
xxmss[n]=(xmss[n]-xmss[n-1])/Ts;
xxxmss[n]=(xxmss[n]-xxmss[n-1])/Ts;

//control law eq. for slave actuator driving force calculation
ts[n]=bs*xxs[n]-ksf*(fs[n]-fmss[n])+fmss[n]
                +ms*(xxxmss[n]+k1*(xxmss[n]
                -xxs[n])+k2*(xmss[n]-xs[n]));

//Euler method for finding xxxs,xxs,xs
xxxs[n+1]=(ts[n]-fs[n]-(bs*xxs[n]))/ms;
//xxxs[n+1]=((ts[n]+ts[n-1])/2-fs[n]-(bs*xxs[n]))/ms;
xxs[n+1]=xxs[n]+(Ts*xxxs[n+1]);
xs[n+1]=xs[n]+(Ts*xxs[n+1]);

if (n==1)
{
        //Create data files routine
        //for n
        _itoa(n,no2,10); //convert int to char

        strcpy(filename2,path_slave);         //copy string to specified location
        strcat(filename2,file_name_slave);    //add string to specified location
        strcat(filename2,no2);                //add string to specified location
        strcat(filename2,extension_slave);    //add string to specified location

        ofstream slave_data (filename2);      //open file for writing

        _gcvt( fs[n], 16, buffer_fs );        //convert double to string
        _gcvt( xs[n], 16, buffer_xs );        //convert double to string

        slave_data << buffer_fs << endl;      //write string to file
        slave_data << buffer_xs << endl;      //write string to file
        slave_data << n << endl;              //write string to file
        slave_data.close();                   //close file
}
```

```
        //Create data files routine
        //for n+1
        _itoa((n+1),no2,10); //convert int to char

        strcpy(filename2,path_slave);             //copy string to specified location
        strcat(filename2,file_name_slave);        //add string to specified location
        strcat(filename2,no2);                    //add string to specified location
        strcat(filename2,extension_slave);        //add string to specified location


        ofstream slave_data (filename2);     //open file for writing

        _gcvt( fs[n+1], 15, buffer_fs );     //convert double to string
        _gcvt( xs[n+1], 15, buffer_xs );     //convert double to string

        slave_data << buffer_fs << endl;     //write string to file
        slave_data << buffer_xs << endl;     //write string to file
        slave_data << n+1 << endl;           //write string to file

        slave_data.close();                  //close file

}

/*-------------------------------------------------------------------------------------------*/
/* Function: logfile_create()                                                                */
/*                                                                                           */
/* Purpose:                                                                                  */
/*                                                                                           */
/* The purpose of the function is to collect individual step data and to create              */
/* a Log File containing all sinulation data for the master and slave models                 */
/* when simulation is over                                                                   */
/*                                                                                           */
/* Returns:                                                                                  */
/*      void                                                                                 */
/*-------------------------------------------------------------------------------------------*/

void logfile_create(void)
{

    //Read data files routine

    /*The function performs a seek to the end of file.
      When new bytes are written to the file,
      they are always appended to the end,
      even if the position is moved with the function.*/

    ofstream logfile (log_file, ios::app ); //open log file

    //create heading info. pointers
    logfile << "n" << setw(30) << "xm" << setw(35) << "fm" ;
    logfile << setw(45)<< "xs" << setw(45) << "fs";
    logfile << setw(25)<< "fmss" << setw(50) << "xmss";
    logfile << setw(40)<< "exm" << setw(35) << "efm";
    logfile << setw(35)<< "tm" << setw(40) << "ts" <<endl;

    for (n=1;n<=k;n++)
    {

        _itoa(n,no1,10); //convert int to char
```

```
}

            //put path and filename together and
            //store final string in filename1

            strcpy(filename1,path_master);           //copy string to specified location
            strcat(filename1,file_name_master);      //add string to specified location
            strcat(filename1,no1);                   //add string to specified location
            strcat(filename1,extension_master);      //add string to specified location

            ifstream master_data (filename1);        //read master simulation files

            master_data >> buffer_fm;     //read string from file and store in buffer_fm
            master_data >> buffer_xm;     //read string from file and store in buffer_xm
            master_data >> n;             //read integer from file

            master_data.close();     //close file


            _itoa(n,no2,10);  //convert int to char


            //put path and filename together and
            //store final string in filename2

            strcpy(filename2,path_slave);          //copy string to specified location
            strcat(filename2,file_name_slave);     //add string to specified location
            strcat(filename2,no2);                 //add string to specified location
            strcat(filename2,extension_slave);     //add string to specified location

            ifstream slave_data (filename2);       //read slave simulation files

            slave_data >> buffer_fs;   //read string from file and store in buffer_fs
            slave_data >> buffer_xs;   //read string from file and store in buffer_xs

            slave_data.close();        //close file

            //write in log file, all values of  n, xm, fm, xs, fs,
            //fmss, xmss, exm, efm, tm, ts during the simulation

            _gcvt( fmss[n], 15, buffer_fmss );   //convert double to string
            _gcvt( xmss[n], 15, buffer_xmss );   //convert double to string
            _gcvt( exm[n], 15, buffer_exm );     //convert double to string
            _gcvt( efm[n], 15, buffer_efm );     //convert double to string
            _gcvt( tm[n], 15, buffer_tm );       //convert double to string
            _gcvt( ts[n], 15, buffer_ts );       //convert double to string


            logfile << n << setw(30)<< buffer_xm << setw(30)
                    << buffer_fm<< setw(30) << buffer_xs;
            logfile << setw(30) << buffer_fs << setw(30)<<buffer_fmss<<setw(30)
                    <<buffer_xmss<<setw(30)<<buffer_exm;
            logfile <<setw(30)<<buffer_efm<<setw(30)<<buffer_tm<<setw(30)
                    <<buffer_ts<< endl;
        }

        //write in log file the duration
        logfile <<endl;
        logfile <<"Simulation elapsed time: "<<duration<<" sec"<<endl;
}
```

```
/*-------------------------------------------------------------------------------*/
/* Function: elapsed_time_start()                                                */
/*                                                                               */
/* Purpose:                                                                      */
/*                                                                               */
/* It activates (starts) the system clock                                        */
/*                                                                               */
/* Returns:                                                                      */
/*        void                                                                   */
/*-------------------------------------------------------------------------------*/

void elapsed_time_start(void)
{
        start=clock(); //start system clock
}


/*-------------------------------------------------------------------------------*/
/* Function: elapsed_time_start()                                                */
/*                                                                               */
/* Purpose:                                                                      */
/*                                                                               */
/* It stops the system clock and estimates the elapsed time (duration)           */
/*                                                                               */
/* Returns:                                                                      */
/*        void                                                                   */
/*-------------------------------------------------------------------------------*/

void elapsed_time_finish(void)
{
        finish=clock();    //stop system clock

        duration=(double)(finish-start)/CLOCKS_PER_SEC; //compute duration

        cout<<"Simulation elapsed time: "<<duration<<" sec"<<endl; //display duration
}
```

```
/*---------------------------------------------------------------------------------------------*/
/*      Function: Poly_lsrsqr()                                                                */
/*                                                                                             */
/*      Determines the best-fit polynomial of the form                                         */
/*      y= a0 + a1*x + a2 * x * x +.....+ aN * x^(N-1)                                         */
/*      and computes the coefficients a0, a1, aN of the best-fit                               */
/*      polynomial of degree N-1 for a set of observations (x1,y1),                            */
/*      (x2,y2),...(xm,yn)                                                                     */
/*                                                                                             */
/*      Input Parameters:                                                                      */
/*              x[]         - array containing observed values of x                            */
/*              y[]         - array containing observed values of y                            */
/*              num_points      - number of observations                                       */
/*              num_poly        - degree of polynomial - 1                                     */
/*                                                                                             */
/*      Output Parameters:                                                                     */
/*              a[] - coefficients of best-fit polynomial                                      */
/*                                                                                             */
/*      Returns:                                                                               */
/*              results    -        status of computation                                      */
/*                                          0 - computation was successful                     */
/*                                          1 - coefficient matrix is singular                 */
/*                                                                                             */
/*      Calls:                                                                                 */
/*              Triangular_Factorization() -                                                   */
/*              -for the solution of simultaneous equations [A]{x}={B}                         */
/*---------------------------------------------------------------------------------------------*/

double poly_leastsqr(double x[],double y[],int num_points,
                                        int num_poly,double x_value,double a[])
{
        double c[MAXSIZE][MAXSIZE];  //coefficient matrix
        double s[2 * MAXSIZE];       //matrix containing sums of products
        double fx;
        int i,j;                     //loop counters


        // compute sums
        s[0] = num_points;
        for (i=1; i<= 2*num_poly;++i)
                {
                s[i]=0.0;
                for(j=1; j<= num_points; ++j)
                        s[i] += pow(x[j],i);
                }
                // create coefficient matrix
                for (i=0; i<= num_poly;++i)
                        for (j=0; j<= num_poly;++j)
                                c[i][j]=s[i+j];
        //create right-hand side vector
        a[0]=0.0;
        for (j=1; j<= num_points;++j)
                a[0] +=y[j];
        for(i=1; i<= num_poly;++i)
                {
                a[i]=0.0;
                for(j=1; j<= num_points;++j)
                        a[i] +=y[j]*pow(x[j],i);
                }
```

```
        //estimate predicted value for a given x value
        fx=Triangular_Factorization(c,a,num_poly+1,x_value,a);
        return(fx);
}

/*-------------------------------------------------------------------------------------------*/
/*      Function: Triangular_Factorization()                                              */
/*                                                                                         */
/*      ( PA = LU Factorization with Pivoting)                                            */
/*      Solves the linear system  [A]x = [B]                                              */
/*      by performing the steps :                                                         */
/*                                                                                         */
/* 1. Find a permutation matrix  P, lower-triangular matrix  L,                            */
/*    and upper-triangular matrix  U  that satisfy:                                        */
/*      PA = LU.                                                                           */
/*                                                                                         */
/* 2. Computer PB and form the equivalent linear system                                    */
/*      LUx = PB.                                                                          */
/*                                                                                         */
/* 3. Solve the lower-triangular system                                                    */
/*      Ly = PB   for  y.                                                                  */
/*                                                                                         */
/* 4. Solve the upper-triangular system                                                    */
/*      Ux = y    for  x.                                                                  */
/*                                                                                         */
/*      Parameters:                                                                        */
/*              n               - number of equations                                     */
/*              A[n][n]         - coefficient matrix                                      */
/*              B[n]            - right-hand side vector                                  */
/*              a[n]            - system solutions                                        */
/*              x_value         - x value                                                 */
/*                                                                                         */
/*      Returns:                                                                           */
/*              fx              -       predicted y for a given x value                   */
/*                                                                                         */
/*      Local Variables                                                                    */
/*              SUM             - Adder                                                   */
/*              temp            - stores intermediate results                             */
/*              i,j,z,l         - loop counters                                           */
/*              DET             - Determinant of [A]                                      */
/*              Row[]           - Field with row-number                                   */
/*              y[]     - See description above                                           */
/*              fx                      - predicted y value                               */
/*-------------------------------------------------------------------------------------------*/

double Triangular_Factorization(double A[][MAXSIZE],double B[],int n_poly,double x_value,double
a[])
{
        int i, ii, z, j, l;                     //Loop counters
        int Row[MAXSIZE];                       //Field with row-number
        int temp;                               //Stores intermidiate values
        double y[MAXSIZE];                      //See description above
        double SUM;                             //Adder
        double DET = 1.0;                       //Determinant of [A]
        double fx;

        // Initialize the pointer vector
        for (l = 1; l<= n_poly; l++) Row[l-1] = l - 1;
```

```
//Start LU factorization
for (z = 1; z <= n_poly - 1; z++)
{

//Find pivot element
for (i = z + 1; i <= n_poly; i++)
{
        if ( fabs(A[Row[i-1]][z-1]) > fabs(A[Row[z-1]][z-1]) )
        {
        //Switch the index for the p-1 th pivot row if necessary
                temp     = Row[z-1];
                Row[z-1] = Row[i-1];
                Row[i-1] = temp;
                DET      = - DET;
        }
} //End of simulated row interchange

        if (A[Row[z-1]][z-1] == 0)
        {
        printf("The matrix is singular !\n");
        printf("Cannot use algorithm to solve the system of equations [A]{x}={B}\n");
        }

//Multiply the diagonal elements
DET = DET * A[Row[z-1]][z-1];

//Form multiplier
for (i = z + 1; i <= n_poly; i++)
{
        A[Row[i-1]][z-1]= A[Row[i-1]][z-1] / A[Row[z-1]][z-1];

//Eliminate X_(p-1)

        for (j = z + 1; j <= n_poly + 1; j++)
        {
                A[Row[i-1]][j-1] -= A[Row[i-1]][z-1] * A[Row[z-1]][j-1];
        }
}

} //End of  L*U  factorization routine

DET = DET * A[Row[n_poly-1]][n_poly-1];

//Start of forward substitution
y[0] = B[Row[0]];
for ( i = 2; i <= n_poly; i++)
{
        SUM =0;
        for ( j = 1; j <= i -1; j++) SUM += A[Row[i-1]][j-1] * y[j-1];
        y[i-1] = B[Row[i-1]] - SUM;
}

if( A[Row[n_poly-1]][n_poly-1] == 0)
{
        printf("The matrix is singular !\n");
        printf("Cannot use algorithm to solve the system of equations [A]{x}={B}\n");
}

//Start of back substitution
a[n_poly-1] = y[n_poly-1] / A[Row[n_poly-1]][n_poly-1];
```

```
        for (i = n_poly - 1; i >= 1; i--)
        {
                SUM = 0;
                for (j = i + 1; j <= n_poly; j++)
                {
                        SUM += A[Row[i-1]][j-1] * a[j-1];
                }

        a[i-1] = ( y[i-1] - SUM) / A[Row[i-1]][i-1];

        } //End of back substitution

        fx = 0;
        for (ii=0; ii<=n_poly;ii++)
                fx += a[ii]*pow(x_value,ii);

        //fx=fx1[0]+fx1[1]+fx1[2];
        return(fx);
}

/*----------------------------------------------------------------------------------*/
/* Function: main()                                                                 */
/*                                                                                  */
/* Purpose:                                                                         */
/*                                                                                  */
/* Main body of simulation                                                          */
/*                                                                                  */
/* Returns:                                                                         */
/*        void                                                                      */
/*----------------------------------------------------------------------------------*/

void main(void)
{

//___________Start Simulation___________

        init();    //initialise variables

        elapsed_time_start();        //start system clock


        for (n=1;n<=k;n++)
        {

                master(); //master model

                slave();  //slave model
        }

        elapsed_time_finish();       //stop system clock

        logfile_create(); //create logfile
}
```

5.3 Source Code of Script.h Include File

```
//_____________simulation parameters_____________

//define timing parameters
#define Tt              20          //time delay (msec)
#define Ts              0.001       //sampling frequency (sec)
#define fi              2           //frequency of sinusoidal i/p (Hz)
#define k               100         //no. of samples

//define human arm parameters
#define mo              1.75        //mass (kgr)
#define bo              0.4         //damping coefficient (N/m^2)
#define ko              5           //spring constant (N/m)

//define master parameters
#define mm              6           //mass (kgr)
#define bm              0.1         //damping coefficient (N/m^2)
#define km              0           //spring constant (N/m)

//define slave parameters
#define ms              (mm+mo)         //mass (kgr)
#define bs              (bm+bo)         //damping coefficient (N/m^2)
#define ks              (km+ko)         //spring constant (N/m)

//define control parameters
#define kmf             0               //spring constant (N/m)
#define ksf             0               //spring constant (N/m)
#define k1              8               //spring constant (N/m)
#define k2              70              //spring constant (N/m)

//define pi
#define pi              3.1415926535    //pi

//define parameters for curve-fitting
#define  n_start    5               //estimation starting point
#define MAXSIZE         5           //degree of polynomial
#define MAXPOINTS       100         //max number of data points for prediction


//define info for master data accumulation
#define path_master         "C:\\simulation\\master\\"          //master data file path
#define file_name_master  "data"                                //master datafile filename
#define extension_master  ".dat"                                //master datafile extension

//define info for slave data accumulation
#define path_slave              "C:\\simulation\\slave\\"       //slave data file path
#define file_name_slave         "data"                          //slave datafile filename
#define extension_slave         ".dat"                          //slave datafile extension

//define filename for master & slave log file
#define log_file    "C:\\simulation\\master.log"                //Logfile path, name, and extension
```

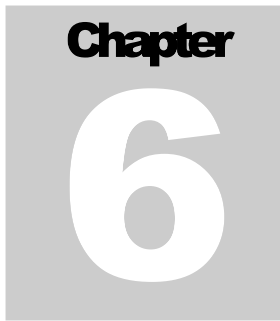
5.4 Source Code of Interpolation Function

```
/*-------------------------------------------------------------------------------------------*/
/* Function: lagrange_poly()                                                                  */
/*                                                                                            */
/* Purpose:                                                                                   */
/*                                                                                            */
/* Performs interpolation using an nth order Lagrange interpolating polynomial.               */
/* It determines a polynomial that passes through a set of n+1 data points,                  */
/* (x0,f(x0)),...(xn,f(xn)) and then computes the value of the dependent                      */
/* variable at a given x value.                                                               */
/*                                                                                            */
/*          x[] - array containing values of independent variables x                          */
/*          y[] - array containing values of dependent variable f(x)                          */
/*          n          - number of data points                                                */
/*          x_value  - x-value for interpolation                                              */
/*                                                                                            */
/* Returns:                                                                                   */
/*          fx  - value of dependent variable f(x) at x=x_value                               */
/*-------------------------------------------------------------------------------------------*/

double lagrange_poly(double x[],double f[],int n,double x_value)
{
	int i,j;			// loop counters
	double fx=0.0;			// dependent variable at x_value
	double li=1.0;

	for (i=0;i<n;++i)
	{
		li=1.0;

		for (j=0;j<n;++j)
			if (j !=i)
				li *=( (x_value - x[j])/(x[i] - x[j]) );
	fx +=li * f[i];
	}
return(fx);
}
```



SIMULATION RESULTS

6.1 Introduction

This chapter presents the simulation results obtained for the proposed method, discussed previously. All simulations were performed for the 1 DOF case and assumed predefined neural input (sinusoidal input). The source code was developed, compiled and executed, using Microsoft Visual C++ 6.0 (© Microsoft Corporation, 1994-98). The simulation results obtained for different simulation parameters (script file) from the executable code were manipulated using Matlab 5.2 (© The MathWorks Inc, 1984-98), to get the simulation graphs.

6.2 Simulation Results and Evaluation

Out of the two, predictor models discussed in sections 4.3 and 4.4 the second one (polynomial least squares curve fitting predictor model), turned out to give the most correct results. The inability of the first method is concentrated on the fact that the accuracy of the approximation is likely to be better if x lies between x_i and x_{i+1} (interpolation) rather than beyond either of them (extrapolation), which is the actual case anyway. On the other hand the advantage is the reduced code complexity (less processing power) compared to the second method.

The simulations parameters chosen for the simulation of the proposed method are summarised in the next table, where the parameter of interest is the time delay *Tt* that is set to 20 msec.

Parameters defined in script file			
Timing Parameters		**Definition of π (pi)**	
Tt	20 msec	pi (π)	3.1415926535
Ts	0.001 sec	**Curve Fitting Parameters (predictor)**	
fi	2 Hz	n_start	5
k	1000	MAXSIZE	5
Human arm parameters		MAXPOINTS	100
mo	1.75 kgr	**Parameters for Master data files**	
bo	0.4 N/m^2	path_master	C:\simulation\master\
ko	5 N/m	file_name_master	data
Master parameters		extension_master	.dat
mm	6 kgr	**Parameters for Slave data files**	
bm	0.1 N/m^2	path_slave	C:\simulation\slave\
km	0 N/m	file_name_slave	data
Slave parameters		extension_slave	.dat
ms	m_m+m_o (kgr)	**Logfile Parameters**	
bs	b_m+b_o (N/m^2)	log_file	C:\simulation\results.log
ks	k_m+k_o (N/m)		
Control Parameters		*Parameters defined in source code*	
kmf	0 (N/m)	**Curve Fitting Parameters (predictor)**	
ksf	0 (N/m)	no. of data points (np)	50
k1	14 (N/m)	Degree of polynomial (num_poly)	2
k2	140 (N/m)		

Table 6.1: Simulation Parameters

A small sample of the produced logfile containing the simulation results is shown on the next page:

n	xm	fm	xs	fs	fmss	xmss	exm	efm	tm	ts
1	1.2217	1.25660398829935e-002	1.2217	0.	6.28301994149675e-003	1.2217	1.2217	0.	-6.28301994149673e-003	6.28301994149675e-003
2	1.22170000104717	2.51300954426194e-002	1.22170000081071	0.	6.28301994149675e-003	1.22170000040536	0.	0.	-7.0013347686364e-003	9.38050612686121e-003
3	1.22170000511578	3.76901826688578e-002	1.22170000283176	0.	6.28301994149675e-003	1.22170000141588	0.	0.	-6.23629127476689e-003	1.08629193428731e-002
4	1.22170001442664	5.02443181783347e-002	1.22170000625434	0.	6.28301994149675e-003	1.22170000312717	0.	0.	-5.44063688072722e-003	1.15265898229252e-002
5	1.22170003120463	6.27905195275211e-002	1.22170001116401	0.	6.28301994149675e-003	1.221700005582	0.	0.	-5.03782416363797e-003	1.17755446803803e-002
6	1.22170005760778	7.53268055257838e-002	1.22170001759278	0.	0.100166300797943	1.22170033033391	0.	0.	-5.08368305060154e-003	2.63284917695262
7	1.22170009571768	8.78511965482387e-002	1.22170036374361	0.	0.106350191773565	1.22170060185095	0.	0.	1.0234149234221	-0.313886299007522
8	1.22170031903797	0.100361714848356	1.22170066937065	0.	0.112519488464629	1.22170160705558	0.	0.	0.387505282082731	5.87567259425628
9	1.2217006236657	0.11285638487027	1.22170173312928	0.	0.118673259746619	1.22170310546083	0.	0.	2.49426884701135	3.99015817686722
10	1.22170136280923	0.125333233560741	1.2217033116784	0.	0.124810576332715	1.22170581257354	0.	0.	2.82139971032713	9.61824516985173
11	1.22170259306259	0.137790290680725	1.22170613118957	0.	0.130930510900992	1.22171007926925	1.2217	1.25660398829935e-002	-13.4579753840853	12.3804117698859
12	1.2217016032646	0.150225589116496	1.22171054799132	0.	0.137032138221345	1.22171067852083	1.2217	1.25660398829935e-002	3.243034029134	-28.6975140387838
13	1.22170117902638	0.162637165190277	1.22171126160308	0.	0.143114535282165	1.2217090530632	1.2217	1.25660398829935e-002	1.63635820050466	-17.354210313144
14	1.22170105462779	0.175023058970325	1.22170973591586	0.	0.149176781416726	1.22170684657858	1.2217	1.25660398829935e-002	0.821127688382978	-4.43154670445873
15	1.22170109625639	0.187381314580432	1.22170763851459	0.	0.155217958429291	1.22170494652933	1.2217	1.25660398829935e-002	0.406858436104846	2.54753526166637
16	1.22170123692426	0.199709980508776	1.22170586996286	0.	0.161237150720905	1.22170366917838	1.22170064307504	0.200332601595885	0.195684007966689	5.03717215056746
17	1.22170144348879	0.212007109916088	1.22170475148293	0.	0.167233445414826	1.22170303570957	1.2217008399583	0.212700383547131	1.11628237452721	5.20752286591133
18	1.22170187143145	0.224270760943081	1.22170430501359	0.	0.173205932481647	1.22170310244161	1.22170254474051	0.225038976929258	0.424828335929748	5.6539169116442
19	1.22170240755017	0.236498997017094	1.22170458811072	0.	0.179153704864032	1.22170379703706	1.22170447779239	0.237346519493238	2.50260465396359	5.08902581828697
20	1.22170340017722	0.248689887157897	1.22170552783808	0.	0.185075858601074	1.22170526902486	1.22170831346867	0.249621152665429	2.81422236721124	6.26780597190277
21	1.2217049032731	0.260841506282615	1.22170727625397	0.	0.190971492952258	1.22170769508266	1.22171402734892	0.261861021801985	5.17982736695721	7.6598667521193
22	1.22170731312208	0.272951935509723	1.22171001292697	0.	0.196839710520996	1.22171143051299	1.22171080905033	0.274064276442691	7.46154670252021	10.4557483622839
23	1.221711012014	0.285019262462058	1.22171409855223	0.	0.202679617377748	1.22171697674737	1.22170684452331	0.286229070564329	-7.50985782553217	14.400052769841
24	1.22171350670452	0.297041581568804	1.22172004198522	0.	0.208490323182675	1.22172340569685	1.22170395724129	0.298353562833452	-10.6492120842338	7.10883230118904
25	1.2217142759917	0.309016994366408	1.22172690230345	0.	0.214270941307838	1.2217293152573	1.22170225454406	0.310435916858582	-7.16897235951779	-3.90810316248081
26	1.22171390194017	0.320943609798365	1.2217332579077	0.	0.220020588958908	1.22173375041688	1.22170146839389	0.322474301441809	-2.66591917582492	-11.4112421103847
27	1.22171313706561	0.332819544513842	1.22173814068358	0.	0.22573838729638	1.22173644737036	1.2217013199362	0.334466890829653	0.592886690659986	-13.4819165729659
28	1.22171252648817	0.344642923165076	1.2217412835423	0.	0.231423461556267	1.22173766310134	1.22170189986964	0.346411864963293	2.30998013224877	-11.4595009478975
29	1.22171235835808	0.356411878703519	1.22174294755297	0.	0.237074941170263	1.22173790418562	1.22170300596339	0.358307409728063	3.35350675279702	-7.47546462034292
30	1.22171280855057	0.368124552674659	1.22174364688021	0.	0.242691959885362	1.22173774075646	1.22170501021164	0.370151717202149	3.65834371959958	-2.99195469867749
31	1.2217139298136	0.379779095511501	1.22174396010366	0.	0.248273655882896	1.22173768436074	1.22170811391135	0.381942985904515	4.27304572144311	1.03102658698558
32	1.22171582652875	0.391373666826626	1.22174440634259	0.	0.253819171897018	1.2217381704681	1.221712848099	0.393679421041992	5.12846702991212	4.45600092270145
33	1.22171864318573	0.402906435702815	1.22174542752059	0.	0.259327655332554	1.22173957455318	1.2217198549425	0.405359234755496	6.65689866187923	7.40936052317358
34	1.22172263642995	0.41437558098217	1.22174740467923	0.	0.264798258382273	1.22174225757392	1.22172676940849	0.416980646365351	8.97721470264412	10.2485396702106
35	1.22172819487267	0.425779291553698	1.22175070410252	0.	0.270230138143503	1.22174661754456	1.22173172821114	0.428541882615676	7.28805329071509	13.3788769687533
36	1.2217350388615	0.437115766639303	1.22175572961965	0.	0.275622456734114	1.22175291836777	1.22173424292607	0.440041177917817	2.10336279112183	15.4550632063625
37	1.22174230614937	0.448383216078154	1.22176274901426	0.	0.280974381407839	1.2217611023353	1.22173475405713	0.45147677459276	-3.47395190985824	15.0034220534712
38	1.22174906905466	0.459579860609366	1.22177170388143	0.	0.286285084668905	1.22177076224006	1.22173404266038	0.462846923112534	-7.11732470123264	11.804750934088
39	1.2217547222231	0.470703932152974	1.22178218136454	0.	0.291553744385991	1.22178126670046	1.22173286081826	0.474149882340525	-8.17303135493974	6.84403332910028
40	1.22175909157608	0.481753674089125	1.22179354127276	0.	0.29677954390546	1.22179194227266	1.22173183463271	0.485383919770724	-7.14656441240936	1.55259054591318
.										
.										
.										
.										
990	1.28264439650246	-0.125333233917082	1.2847091925695	0.	-5.81451320623497	1.28846099115747	1.29079223232534	-11.2682700612161	1.61211409205276	0.823736655707004
991	1.28275476046321	-0.112856385227148	1.28482311115595	0.	-5.8326868506526	1.28858937582951	1.29093363760004	-11.3041234342152	1.60207584981786	0.823777951908909
992	1.28286537078782	-0.100361715205715	1.28493712868674	0.	-5.85088519086204	1.28871788224786	1.29107518742303	-11.3400261987979	1.59188048216363	0.823819434965417
993	1.28297622785537	-8.78511969060232e-002	1.28505124516084	0.	-5.86910822686329	1.28884651041201	1.29121688179431	-11.3759783549643	1.58152956862831	0.823861137182606
994	1.28308733202171	-7.5326805883937e-002	1.28516546057725	0.	-5.88735595865634	1.28897526032146	1.29135872071388	-11.4119799027143	1.57102478052578	0.823903047234199
995	1.28319868361931	-6.27905198859847e-002	1.285279774935	0.	-5.90562838624119	1.28910413197572	1.29150070418175	-11.4480308420478	1.56036775095	0.823945176475453
996	1.28331028295725	-5.02443185370538e-002	1.28539418823313	0.	-5.92392550961786	1.28923312537432	1.29164283219791	-11.484131172965	1.54956019338601	0.823987440295348
997	1.28342213032118	-3.76901830277758e-002	1.28550870047072	0.	-5.94224732878633	1.2893622405168	1.29178510476235	-11.5202808954658	1.53860375144405	0.824029961731878
998	1.28353422597325	-2.51300958016796e-002	1.28562331164687	0.	-5.96059384374661	1.2894914774027	1.29192752187509	-11.5564800095503	1.52750020258744	0.82407264569008
999	1.28364657015208	-1.25660402421374e-002	1.28573802176071	0.	-5.97896505449869	1.2896208360316	1.29207008353613	-11.5927285152183	1.51625130785997	0.824115548893853
1000	1.28375916307271	-3.59172737341915e-010	1.28585283081138	0.	-5.99736096104259	8.17168554541762e-043	1.29221278974545	-11.6290264124699	1.50485880213354	0.824158581490778

Simulation elapsed time: 6.669 sec

The logfile containing the simulations results was uploaded in Matlab in order to obtain the simulation graphs of interest. The next few lines below point the commands required by Matlab's editor in order to plot a graph, for example master-slave positions, xm, xs:

- load logfile with simulation results, *load c:\simulation\results.log*
- specify name of 1st variable to be plot, and number of column, *xm=results(:,2);*
- specify name of 2nd variable to be plot, and number of column, *xs=results(:,4);*
- plot *xm* and *xs* and mark last with red colour, *plot(xm);hold on;plot(xs,'r');hold off;*

The plot graph of *xm, xs* is shown in figure 6.1 below.

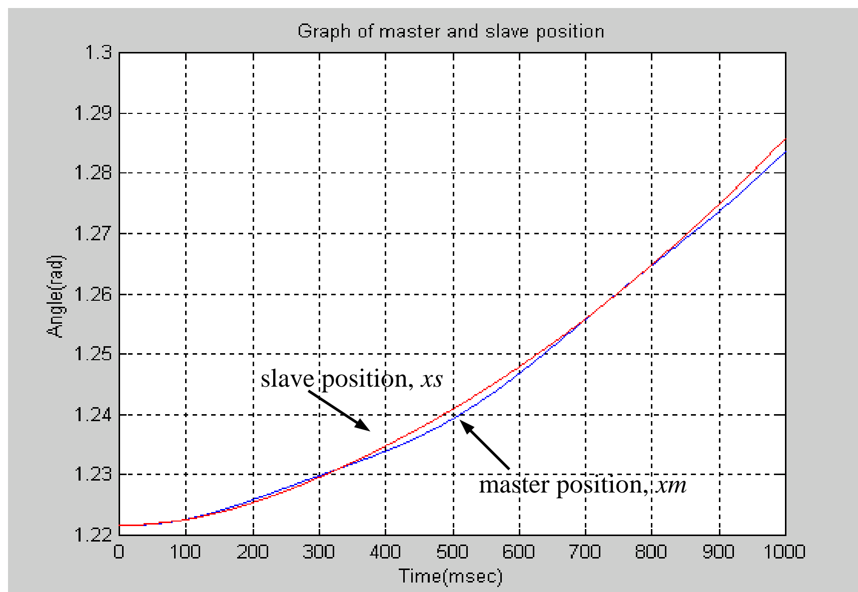


Fig 6.1: Master and slave position graph

In the graph above it is shown that the slave robot follows as closely as possible the master robot. The small variations of the slave are due to small prediction inaccuracies, which can be improved if the prediction horizon (*n_start*) is increased appropriately. Ideally the slave should lead the master.

The second graph (Fig. 6.2) shows the predicted master position, *exm* with the slave position, *xs*.

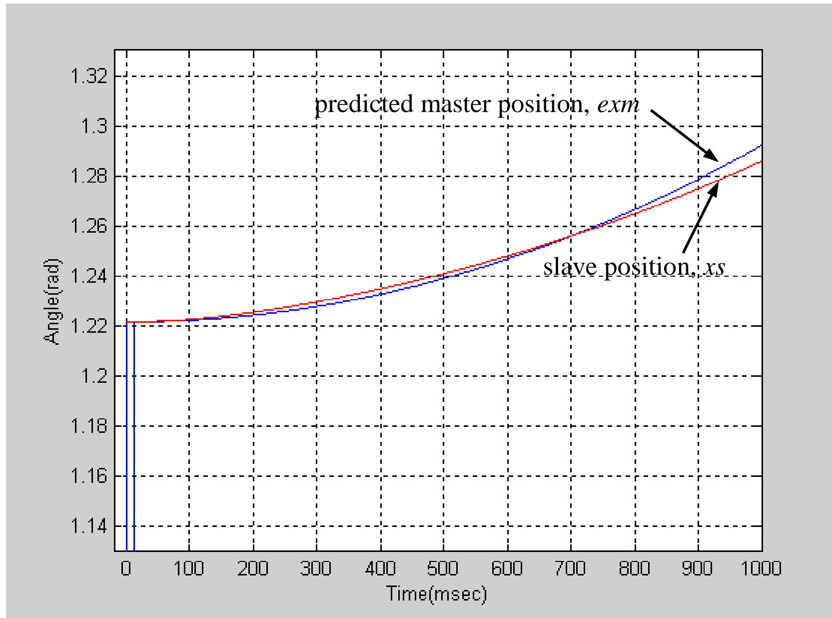


Fig 6.2: Predicted master position and slave position graph

It can be observed from the graph that, the predicted master position, *exm* drops to zero after the first sample (which has been set to 1.2217) for *Tt/2-1 msec*. This is absolutely normal since the prediction is always *n+Tt/2* ahead. As it can be seen *xs* follows as close as possible *exm*. The next graph (Fig. 6.3) shows the predicted master force, *efm*.

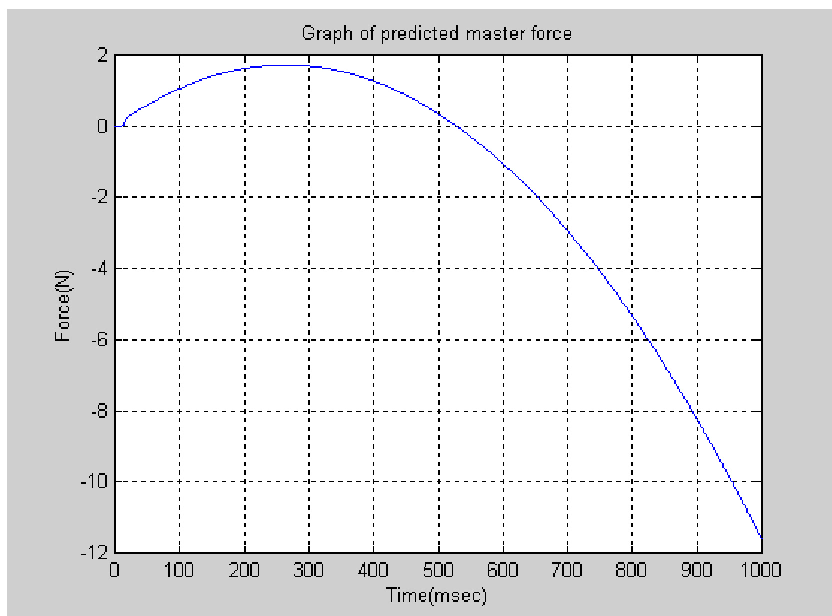


Fig 6.3: Predicted master force graph

The following graphs (Fig. 6.4, Fig 6.5) illustrate the master actuator driving force, *tm*, and slave actuator driving force, *tm* respectively.

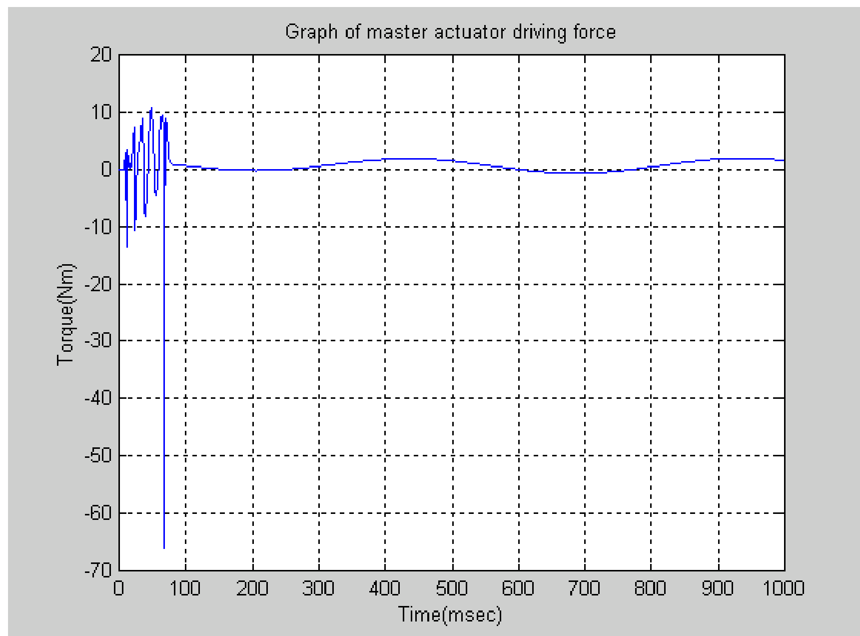


Fig 6.4: Master actuator driving force graph

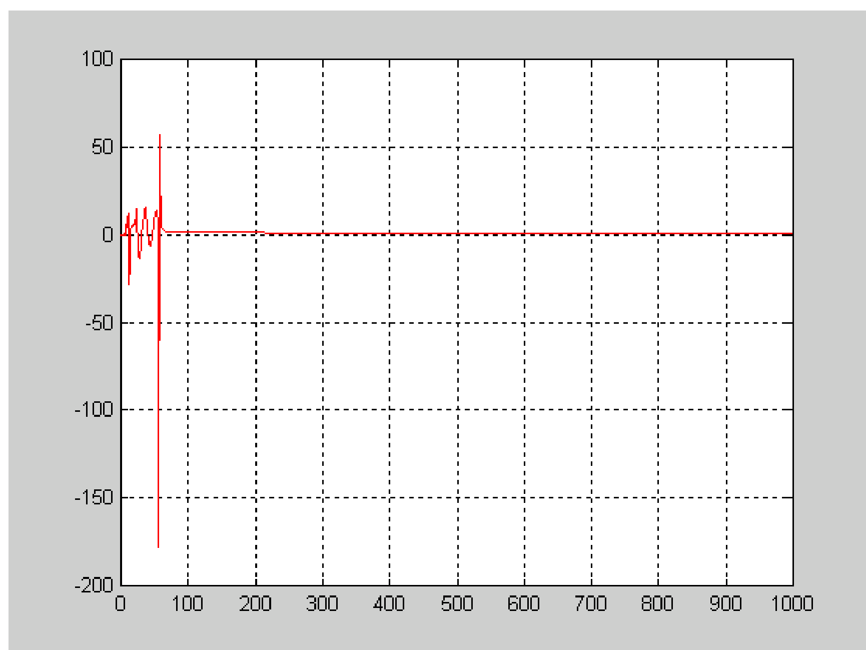


Fig 6.5: Slave actuator driving force graph

Another simulation of the proposed method, with increased time delay, *Tt* of 50 msec was performed. The rest of parameters, except the prediction starting point (prediction horizon), *n_start* that was increased to 20, and degree of polynomial for the predictor model, which was set to 3, kept the same as in table 6.1. The same procedure followed for the first simulation was carried out here again. The graph of master and slave position occurred from the simulation is shown below. It can be observed from the graph that the slave leads the master.

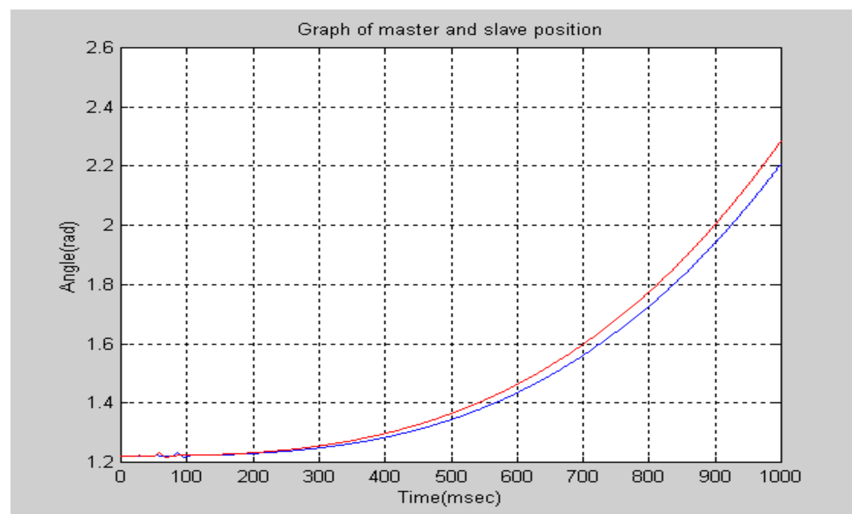


Fig 6.6: Master and slave position graph

The following graph illustrates the predicted master position, *exm* and slave position, *xs*.

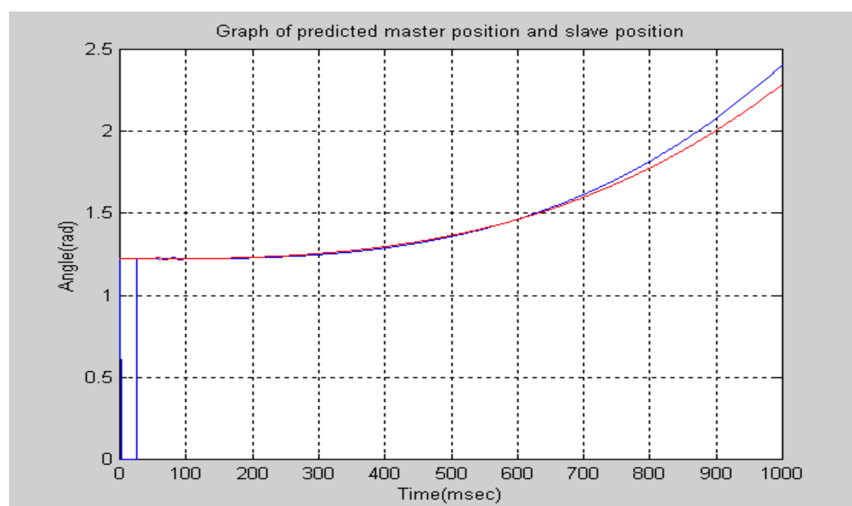


Fig 6.7: Predicted Master position and slave position graph

The next graph (Fig. 6.3) shows the predicted master force, *efm*.

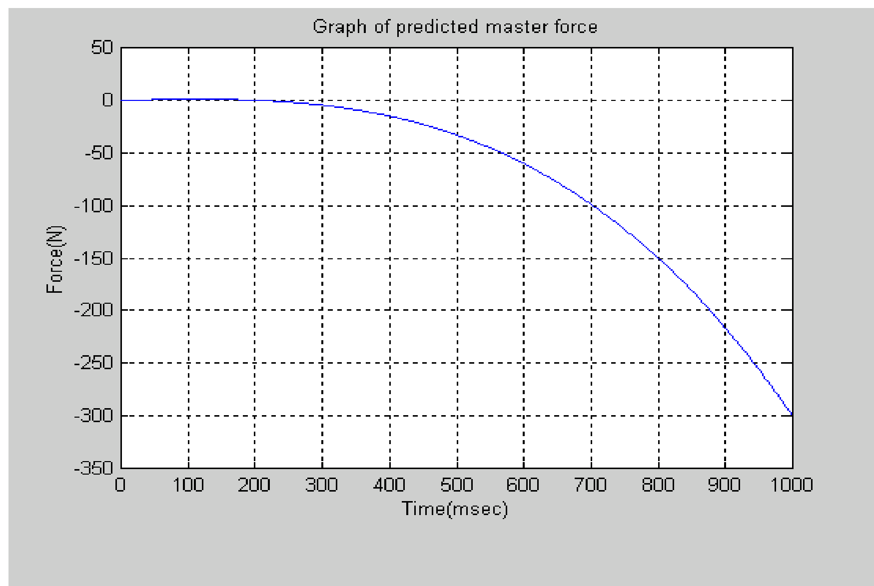


Fig 6.8: Predicted master force graph

The following graph (Fig. 6.7) illustrates the master actuator driving force, *tm*, and slave actuator driving force, *tm* respectively.

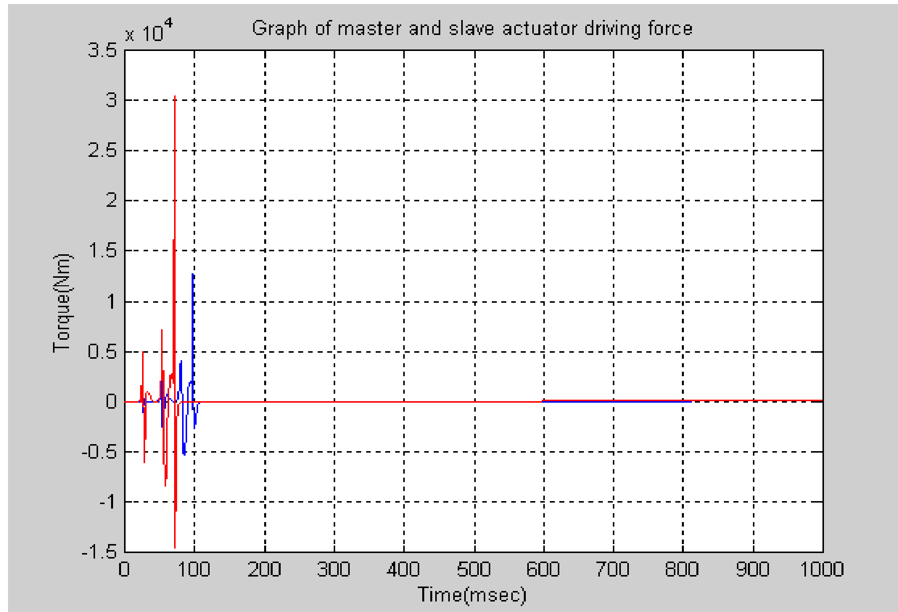


Fig 6.9: Master and slave actuator driving force graph

The final simulation was performed for a time delay, *Tt* of 100 msec and neural input frequency, *fi* of 1 Hz. Moreover the prediction horizon, *n_start* was increased to 60. All other parameters remained the same according to table 6.1. The graph, showing the master and slave position is shown below.

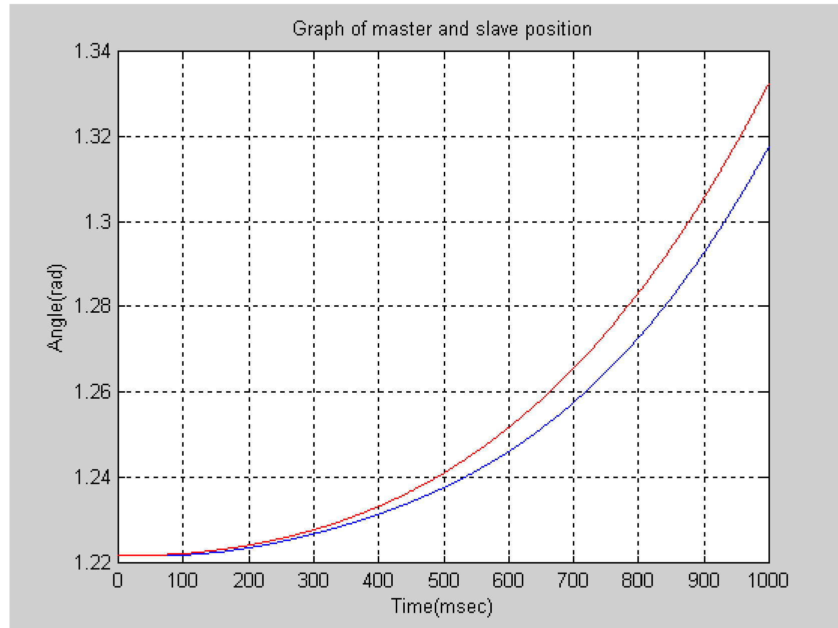


Fig 6.10: Master and slave position graph

Similarly, the graph of predicted master position and slave position is shown next.

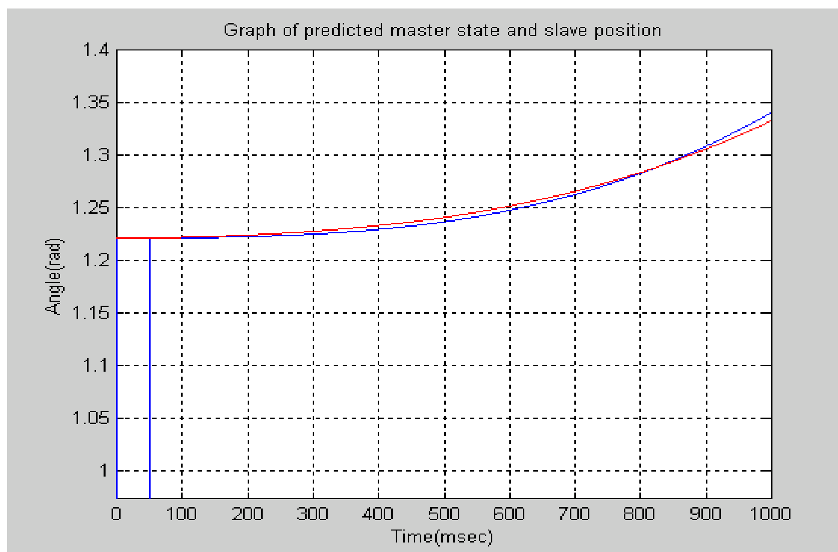


Fig 6.11: Predicted Master position and slave position graph

The next graph (Fig. 6.11) shows the predicted master force, *efm*.

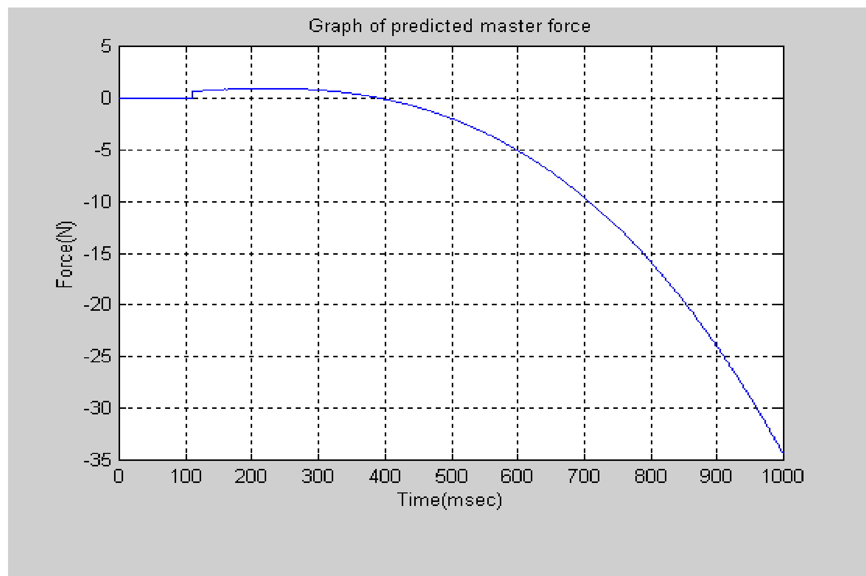


Fig 6.12: Predicted master force graph

The final graph of the simulation (Fig. 6.13) illustrates the master actuator driving force, *tm*, and slave actuator driving force, *tm* respectively

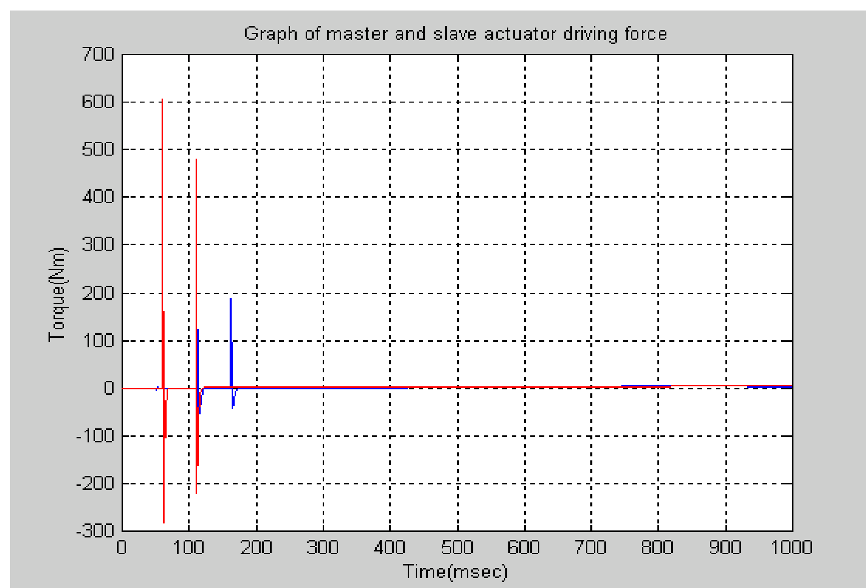
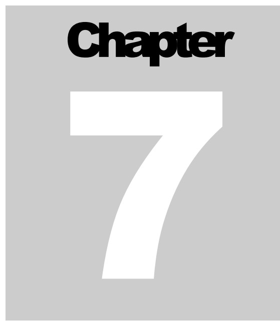


Fig 6.13: Master and slave actuator driving force graph

From the above simulations of the proposed method, it can be concluded that using sinusoidal inputs, good performance can be achieved for time delays up to 1/10 of the input frequency, f_i.



CONCLUSION

The conceptual framework and simulation results on a general technique for time delay compensation in teleoperation, which is based on predicting the human arm position and force (effectively the master state) was presented. It was shown that the proposed method, tends to be significantly less complex and more intuitive than predicting the slave and environment dynamics. Simple polynomial predictors, employing no knowledge of the human arm dynamics, were shown to produce good performance for small time delays when the master force and position are smooth. On the other hand for real life force profiles, better performance could be achieved by employing a human arm model and predicting the neural input to it.

Chapter 2 attempted a brief presentation of the most important matters concerning the control of teleoperator systems. Several methods were discussed and analysed.

Chapter 3 presents the proposed method of variable-time-delays-robust telemanipulation through master state prediction.

Chapter 4 illustrated the methods carried out in order to implement the proposed method, previously described in chapter 3. Initially to predictor models were investigated and developed. A method for solving simultaneous equations was developed and presented as well. All the sections of the implemented method were presented in a methodical manner one by one. The Pseudo Code for all the developed was included on order to ease the explanations of the source code. Due to time limitations and programming difficulties (using *Microsoft Direct X* drivers) at the present time and after a common agreement, with the supervisor P. A. Prokopiou, it was decided to omit the programming of the force feedback joystick for the master robot. Instead it was decided to use simple sinusoidal inputs as neural inputs.

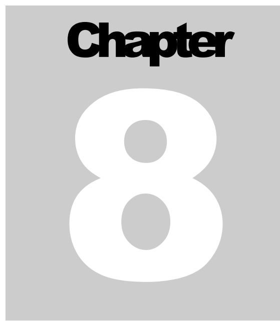
Finally chapter 5 presented various simulation results. The aims set for chapter 5 were successfully completed, since the proposed method proved to function satisfactorily.

Obviously in order to understand the concepts of teleoperation and delay elimination techniques took a great deal of time. Without any prior knowledge in this field, background reading was also essential. The correct results were not gained immediately and in many cases several attempts at understanding concepts required as necessary to achieve that. Regular meetings with the project supervisor were proven to be invaluable. Knowledge of the subject area was gained mostly by periodicals, books, and Internet resources. On the other hand previous knowledge in C++ programming and use of *Matlab*, helped a lot in implementing the proposed

method. Throughout the period of this project many software packages such as, *Microsoft Visual C++ 6.0, Matlab 5.2, Microsoft office 2000, Visio 3.0, Adobe Photoshop 5.0* were used.

As time is important in industry and deadlines need to be met, following the Gantt Chart (Appendix B) helped in achieving the completion of the required task within the available time.

Having the opportunity to undertake a project of this nature has proven to be an invaluable source of knowledge. It has allowed discovering previously unfamiliar areas of expertise, which will be beneficial when considering future career opportunities. Also by allowing the student to use appropriate tools such as, *Microsoft Visual C++ 6.0, Matlab 5.2*, from previous areas of study enabled him to plan and undertake investigations both theoretical and practical.



RECOMMENDATIONS FOR FURTHER WORK

In future work the program code for the force feedback joystick could be implemented. Through the master (force feedback joystick), the human operator gives an order to the system (effectively to the slave robot) and feels back the response of his actions. For example if the slave robot hits a wall, then the master must feel on his hand the reaction force absorbed by the slave. Force feedback, also known as *haptic feedback* or *force reflection*, refers to the technique of emulating “feel” sensations to computer software by imparting real physical forces upon the user hand. These forces are imposed by actuators, usually motors, incorporated in the interface hardware. The interface hardware in this case is the joystick. Through the force feedback joystick the human operator can feel, rigid surfaces, viscous liquids, compliant springs, jarring vibrations, grating textures, heavy masses, and just about any other physical phenomenon that can be represented mathematically.

The joystick could be very well programmed in C++ but it requires the use of *Direct X 5* or higher from *Microsoft* in order to make the programming easier. There is a number of available force feedback joysticks in the market from different manufacturers, like Logitech Inc., ThrustMaster, ACT Labs, Advanced Gravis, and Nuby manufacturing.

Furthermore a graphic routine that will provide the human operator with visualization of the master position, slave position, predicted master position could be implemented. This could be simply achieved by creating a function to display three bars in a separate window that will accept as inputs, x_m, x_s, $\hat{x}_m$ (exm) and rotate according to the input values, in order to indicate the current position of each.

REFERENCES

[1] P. A. Prokopiou, A. G. Tzafestas, W. S. Harwin, "Towards Variable-Time-Delays-Robust Telemanipulation Trough Master State Prediction", submitted to AIM'99: *1999 IEEE/ASME Int. Conf. on Adv. Intel. Mechatr.*, Atlanta, U.S.A. Sept. 19-22, 1999.

[2] J. Vertut and P. Coiffet, "*Robot Technology", Volume 3A: Teleoperations and Robotics: Evolution and Development.* (Englewood Cliffs, NJ: Prentice-Hall, 1986).

[3] A. K. Bejczy and M. Handlykken, "Generalization of bilateral force reflecting cintrol of manipulators", in Proc. *4th Rom-An-Sy*, Warsaw, Poland, 1981, pp. 242-255.

[4] J. Vertut, R. Fournier, B. Espiau, and G. Andre, "Advances in a computer aided bilateral manipulator system", in *Proc. 1984 Nat. Topical Meet Robotics and Remote Handling in Hostile Environ.*, 1984, pp. 367-372.

[5] R. M. Satava, "The Modern Medical Battlefield: Sequitur on Advanced Medical Technology", *IEEE Robotics and Automation Magazine*, Sept. 1994, pp. 21-25.

[6] T. L. Brooks, "Telerobotic Response Requirements", *IEEE International Conference on System, Man and Cybernetics*, pp 113-120.

[7] Y. Strassberg, A. A. Goldenberg, J. K. Mills, "A New Control Scheme for Bilateral Teleoperating Systems: Lyapunov Stability Analysis", *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, pp 837-842.

[8] G. J. Raju, "Design Issues in 2-port Network Models of Bilateral Remote Manipulation", in *Proc. IEEE International Conference on Robotics and Automation*, pp. 1316-1321, 1989.

[9] G. J. Raju, "An Experiment in Bilateral Manipulation with Adjustable Impedance", in *Proc. Japan-USA Symposium on Flexible Automation*, pp. 395-339, 1990.

[10] Y. Yokokohji, T. Yoshikawa, "Bilateral Control of Master-Slave Manipulators for Ideal Kinesthetic Coupling: Formulation and Experiment", *IEEE Transactions on Robotics and Automation*, Vol. 10, No. 5, Oct. 1994.

[11] P. A. Drakatos, N. P. Mattheos, ”Man-Machine System Design”, Accepted for Publication.

[12] P. A. Prokopiou, S. G. Tzafestas, W. S. Harwin, “A Novel Scheme for Human-Friendly and Time-Delays-Robust Neuropredictive Teleoperation”, Accepted for Publication.

[13] B. Hannaford, “A Design Framework for Teleoperators with Kinesthetic Feedback”, *IEEE Transactions on Robotics and Automation*, Vol. 5, No. 4, Aug. 1989, pp. 426-434.

[14] K. Funaya, N. Takanasi, “Predictive Bilateral Master-Slave Manipulation with Statistical Environment Model”, *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, Georgia, USA, Vol. III, pp. 755-760.

[15] Y. Strassberg, A. A. Goldenberg, J. K. Mills, “A New Control Scheme for Bilateral Teleoperating Systems: Lyapunov Stability Analysis”, *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, pp 837-842.

[16] C. A. Lawn, B. Hannaford, “Performance testing of passive Communication and Control in teleoperation with Time delay”, *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, Atlanta, Georgia, USA, Vol. III..

[17] T. B. Sheridan, “Space Teleoperation Trough time Delay: Review and Programs”, IEEE Transactions on Robotics and Automation, Vol. 9, No. 5, Oct.1993.

[18] G. Niemeyer, J.-J. E. Slotine, “Stable Adaptive Teleoperation”, *IEEE Journal of Oceanic Engineering*, Vol. 16, No. 1, Jan. 1991.

[19] D. A. Lawrence, “Stability and Transparency in Bilateral Teleoperation”, IEEE Transactions on Robotics and Automation, Vol. 9, No.1, pp. 152-162, Jan. 1991.

[20] R. J. Anderson, M. W. Spong, “Bilateral Control of Teleoperators with Time Delay”, *IEEE Transactions on Automatic Control*, Vol. 34, No. 5, pp. 494-501, May 1989.

[21] P. A. Prokopiou, W. S. Harwin, S. G. Tzafestas, “Exploiting A Human Arm Model for Fast, Intuitive, and Time-Delays-Robust Telemanipulation”, *in S. G. Tzafestas and G. Schmidt (Eds): Progress in System and Robot Analysis & Control Design*, Springer-Verlag, UK, 1999, pp. 255-266.

[22] P. A. Prokopiou, W. S. Harwin, S. G. Tzafestas, “Enhancement of a Telemanipulator Design with a Human Arm Model”, *in S. G. Tzafestas (Ed): Advances in Manufacturing-Decision, Control and Information technology,* Springer-Verlag, U.K., 1999, pp. 445-456.

[23] G. Niemeyer, J.-J. E. Slotine, “Towards Force Reflecting teleoperation Through the Internet”, *Proceedings IEEE International Conference on Robotics and Automation*, Leuven, Belgium, pp. 1909-1915, May 1998.

[24] A. Sano, H. Fujimoto, M. Tanaka, "Gain-Scheduled Compensation for Time Delays of Bilateral teleoperation", *Proceedings IEEE International Conference on Robotics and Automation*, Leuven, Belgium, pp. 1916-1923, May 1998.

[25] K. Brady, T.-J. Tarn, "Internet Based Remote Teleoperation", *Proceedings IEEE International Conference on Robotics and Automation*, Leuven, Belgium, pp. 65-70, May 1998.

[26] T. B . Sheridan, "Telerobotics", *Automatica*, Vol. 25, No. 4, pp. 487-507, 1989

[27] J. C. Houk, J. T. Buckingham and A. G. Barto, "Models of the Cerebellum and Motor Learning", *Behavioral and Brain Sciences*, Vol. 19, pp. 368-383, 1996

[28] M. K. Haugland and T. Sinkjaer, "Cutaneous Whole nerve recordings Used for Correction of Footdrop in Hemiplegic Man", *IEEE Transactions on rehabilitation Engineering*, Vol. 3, No. 4, pp. 307-317, 1995.

[29] J. M. E. Van de vegte, P. Milgram and R. H. Kwong, "Teleoperator Control Models: Effects of Time Delay and Imperfect System Knowledge", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 20, No. 6, pp1258-1272, 1990.

[30] W. H. Zangemeister, S. Lehman, L. W. Stark, "Simulation of Head Movement Trajectories: Model and Fit to Main Sequence", *Biological Cybernetics*, Vol. 41, pp. 19-32, 1981.

[31] K. B. Rojiani, "Programming in C with Numerical Methods for Engineers", (Prentice Hall Int. (UK), Ltd. 1996).

[32] C. H. Pappas, W. H. Murray, III, "Visual C++ Handbook", (McGraw-Hill, Inc, 1994).

[33] G. C. Burdea, "Force and Touch Feedback for Virtual Reality", (John Wiley & Sons, Inc., 1996).

[34] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, "Numerical Recipes in C", 3rd Edition, (Cambridge University Press, 1988).

[35] L. B. Rosenberg, "A Force Feedback Programming Primer", (Immersion Corporation, 1997).

[36] "MATLAB Reference Guide", (The MathWorks, Inc., Oct. 1992).

APPENDIX A MODIFIED STARK MODEL FOR THE HUMAN ARM - STATE EQUATIONS

$$\dot{X}_L = \begin{cases} \dfrac{B_h(HTL - Fsl)}{0.25 * HTL + Fsl} * \sigma_L + \left(v + \dfrac{H\dot{T}L}{k_2(HTL+1)} \right) * (1 - \sigma_L) & \text{if } HTL \geq thres_HTL \\ \dfrac{f_{xl}(\theta) - X_L}{\tau_L} & \text{if } HTL < thres_HTL \end{cases} \quad (A.1a)$$

$$\dot{X}_R = \begin{cases} \dfrac{B_h(HTR - Fsr)}{0.25 * HTR + Fsr} * \sigma_R + \left(v + \dfrac{H\dot{T}R}{k_2(HTR+1)} \right) * (1 - \sigma_R) & \text{if } HTR \geq thres_HTR \\ \dfrac{f_{xl}(\theta) - X_R}{\tau_R} & \text{if } HTR < thres_HTR \end{cases} \quad (A.1b)$$

$$\dot{v} = J_p^{-1}(-B_p v - K_p\theta + F_e + F_{sl} - F_{sr}) \quad (A.2)$$

$$\dot{\theta} = v \quad (A.3)$$

$$\text{with: } \sigma_L = \left(1 + e^{-1.5*(\frac{B_h(HTL-Fsl)}{0.25*HTL+Fsl} - 4.5*dXlo)} \right)^{-1}, \sigma_R = \left(1 + e^{-1.5*(\frac{B_h(HTR-Fsr)}{0.25*HTR+Fsr} - 4.5*dXro)} \right)^{-1} \quad (A.4a,b)$$

$$Fsl = \max(0, k1(e^{k_2(X_L - \theta)} - 1)), \quad Fsr = \max(0, k1(e^{k_2(X_R - \theta)} - 1)), \quad (A.5a,b)$$

$$f_{XL}(\theta) = f_{XR}(\theta) = \theta - 0.1(\theta - X_{lo})^2 \quad (A.6)$$

$$H\dot{T}L = \frac{N_L - HTL}{T}, \quad H\dot{T}R = \frac{N_R - HTR}{T} \quad (A.7)$$

<u>where:</u> subscripts l, r denote left, right muscle, θ,v position and velocity of the arm, X_L, X_R internal model variables, K_p, J_p, B_p, passive parameters of the arm (muscles' load), B_h, T, k_1, k_2, X_{lo}, dX_{lo}, X_{ro}, dX_{ro}, thres_HTL, thres_HTR const-ants, N_L, N_R the neural input, HTL, HTR activation levels, F_{sl}/F_{sr} the left/right muscle's force, F_e an external force.

APPENDIX B GANTT CHART

No	Activity	Duration																						
		Month	April				May				June				July				August				September	
		Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	Project Allocation																							
2	Understand Concepts																							
3	Implementation																							
4	Simulation																							
5	Verification & Testing																							
6	Possible Corrections																							
7	Write Thesis																							

Predicted Time: **Actual Time**:

View publication stats