LB/DON/53/08

UNIVERSAL DYNAMIC SIMULATOR FOR ROBOTIC MANIPULATORS: KINEMATIC MODELING

A dissertation submitted to the
Department of Electrical Engineering, University of Moratuwa
in partial fulfillment of the requirements for the
degree of Master of Science

by

LASANTHA KURUKULARACHCHI

621.3 "08"
621.3 (043)

Supervised by: Dr. Rohan Munasinghe

Department of Electrical Engineering
University of Moratuwa, Sri Lanka

January 2008

91208

I

DECLARATION

The work submitted in this dissertation is the result of my own investigation, except where otherwise stated.

It has not already been accepted for any degree, and is also not being concurrently submitted for any other degree.

UOM Verified Signature

[illegible]

Lasantha Kurukularachchi

Date- 19/2/2008

I endorse the declaration by the candidate.

UOM Verified Signature

[illegible]

Dr. Rohan Munasinghe

DEPARTMENT OF ELECTRONIC & TELECOMMUNICATION ENGINEERING
OFFICE
21 FEB 2008
UNIVERSITY OF MORATUWA
SRI LANKA

Table of Contents

Abstract

This project highly focuses on a total simulating solution to the robotic manipulator users. The existing simulators are narrow with limited applications. Therefore the simulator users do not have an adequate solution for the universal manipulators. The simulating solution developed through this project is the combination of kinematic, dynamic, trajectory planning and frictional model on a one interface. This project has been divided into four different research components because of the vast extent of the research areas.

This thesis is based on the kinematic behavior of this robotic simulator. Under the kinematic behavior, the forward kinematic and the reverse kinematic have been focus on. In the forward kinematic bases, the systematic analytic approaches are used to develop the algorithm. This algorithm describes the spatial relationship between links & link parameters of the manipulator and it supports to find the end-effector position and orientation with respect to the joint space parameters in a graphical way. On the other hand the forward kinematic supports to visualize the manipulator in the 3D environment.

The reverse kinematic is required to find a set of joint variables that would bring the end-effector in the specified position and orientation. In general this solution is non-unique for the universal model, but solving the inverse kinematic is most important to design the practical manipulators. Therefore the inverse kinematic algorithm is the combination of Jacobian transformation and the Taylor series expansion. This combination is ideal to solve the inverse kinematic in this simulator.

The software tool is the final output of this project. The kinematics module supports to find the manipulator geometry and the joint angles. But the software tool is the combination of kinematics, dynamics and the trajectory planning.

The object-oriented program is well adapted to this application since OOP can describe each part of the robot as one object with its own properties and behavior. Even if C++ is not a perfect OO language, a lot of very useful libraries are available, and maintains very good efficiency for

intensive computations. The robotic applications will be highly popular in the future. Therefore this software tool may be most important to develop the manipulator application because it provides a total solution for designing the application. Still nobody has developed this type of an application tool to manipulator designers. This software application operates with out any hindrance and the major advantage is that this simulator can be used for universal serial link manipulator for N-degree of freedom.

Acknowledgements

I would like to thank my supervisor, Dr Rohan Munasighe for his guidance though out my project to achieve its goals and his valuable suggestions to direct this project in the correct direction.

My sincere thanks are extended to Mr. Hiran Perera Automation Engineer in ANSELL LANKA (PVT) LTD for his support to test this simulator in the practical environment and also I thank my colleagues Rajeeve, Bannayake and Mahinda for giving full support to develop a dynamic simulator tool.

I am grateful to my family for their never ending love and support. I would also like to thank my friends both in the university and factory for their friendship. Finally I thank all the members of the staff in the Electrical Department of University of Moratuwa.

List of Figures

List of Tables

List of Principal Symbols

q_i	Joint Variable
θ	Joint Angle
a_i	Link Length
α_i	Link Twist
d_i	Link Offset
$^{i-1}A_i$	Homogeneous Transformation Matrix.
R_i	Rotation Matrix
P_i	Translation Vector
T	Forward Kinematics Equation
J	Jacobian Matrix
$\dot{x}$	6x1 Cartesian Velocity Vector
$\dot{q}$	nx1 Vector of n Joint Velocity
ω_i	Angular Velocity of ith Link
δq	Small Displacement of Joint Variable
δx	Small Displacement of Cartesian Variable
I	Identity Matrix

List of Acronyms

IK	Inverse kinematic
RP	Revolute & prismatic
DOF	Degree of freedom
OOP	Object oriented programming
D-H	Denavit-Hartenberg parameters
TP	Trajectory planning
TPA	Trajectory planning algorithm

CHAPTER 1

Introduction of the Simulator Design

The simulation is a powerful visualizing, planning, and strategic tool in different areas of research and development. And it plays a very important role in robotic manipulator designing. The simulator facilitates the study of the structure, characteristics, motion and the behavior of robot manipulators at different levels of details each posing requirement for different simulation tools. As the complexity of the motional behavior increases, the role of simulation becomes more and more important. The robotic kinematic algorithms and their numerical solutions are quite complex to understand the characteristic and behavior of the robotic manipulators in the actual environment. The easiest way to understand the motional behavior is to visualize the exact model of robotic manipulator and its characteristics. Hence, the simulation tools can certainly enhance the design, development, and even the operation of robotic manipulators.

Augmenting the simulation with visualization tools and interfaces, one can simulate the operation of the robotic systems in a very realistic way. Depending on the type of application different structural attributes and functional parameters have to be modeled. Therefore, a variety of simulation tools have been developed for the robotic manipulators that are used in mechanical design.

A robotic manipulator is designed to perform a task in the 3D space. The tool or end-effector is required to follow a planned trajectory to manipulate an object or carry out the task in the working space. This requires control of position of each link and joint of the manipulator to and orientation of the tool. To program the tool motion and joint –link motions, a mathematical model of the manipulator is required to refer to all its geometrical and time based properties in the motion.

Kinematics is the study of motion without regard to the force which causes it; within the kinematics one studies the position, velocity and acceleration and all higher order derivatives of the position and variables. The kinematics of manipulator involves the study of the geometrical and time based properties of the motion and in particular how various links move with respect to one another.

1.1 Background of the requirement

In order to perform tasks on different manipulator platforms, a kinematics model has to be developed for each one of them. Because the equations can become too cumbersome to deal with manually when the robots have more than just several joints, a method is needed to automate the formation of the kinematics model. Industrial robots are usually developed for specified pre-determined tasks. Therefore the requirement of arms combination and configuration are different. The manipulator arm configurations, along with equations needed for the manipulator arm motion, are determined and solved during robot development stage. This limits the robot to the prescribed tasks and to no other. However, for robots that must adapt to their environment or perform a wide range of tasks, a method is manipulator arm to adapt to changes in joint space & Cartesian space Changes to the equations (Jacobian and kinematics expressions). They are required when something changes the geometry of the manipulator arm such as when a tool is added to the end-effector. In order to accommodate to different manipulator platforms and to provide for tool acquisition, a method is needed to automate the formation of the kinematic model that eliminates manual calculation processes.

1.2 Problem statement

Robots are an integral part of today's industrial scenario. As a result, simulation has evolved as a major tool in the programming and the designing of robots. Simulation in this sense includes actually having a computer draw conclusions about the workings of a system. These conclusions are derived from the knowledge already available about the system in question. It is, in effect, a knowledge based process that utilizes the information stored about a particular system in the database to predict its response to various situations. Robot simulator is the collection of computer programs and related information that is developed, marketed, manufactured and sustained for industrial robots. Consequently, the software for industrial robots is best seen from three points of view: operation; application; and manufacturing.

The existing simulators are developed with certain limitations and no simulator designer has given a total solution (i.e. robotic motion, dynamic characteristic changes , trajectory planning and controlling) for the different manipulator designing platforms. And also lots of accurate simulators are already developed with the pre-defined commercial manipulators. (E.g. Robware for the ABB manipulators) They are very expensive and they can not simulate with the other manipulators. For example if any user who wants to add an extra link or change the link parameters (i.e. Joint type, Joint variables or maximum limits) they do not have facilities to change it on this simulator.

The other simulators are study versions (e.g. Robotica) and they can not be used for industrial application. Major weaknesses of these simulators are, they disregard the dynamic constraints and the manipulator controlling. Then the simulators can not be used for the actual manipulator platform. The table1.1 is explains some of the simulators and their features.

Almost all simulators are developed for the non- redundant manipulators. The reason for this is most of the practical manipulators are developed with the non-redundant base. Therefore the simulator designers are interested in doing their products on the non-redundant base. But now a days lots of researchers are interested to do a research on the redundant base manipulators. Therefore today the requirements of the simulator are strongly feel for the universal manipulator.

The MATLAB robotic tool box [6] by Peter I. Corke has given some kind of solution on this matter. But it is not a simulator. He has given a tool that can support to develop robotic manipulator in the Cartesian space and the joint space both. If any one wants to develop the robotic manipulator on this platform he should study all the tool and functions that have been provided in this tool box. In the MATLAB robotic tool box, there are fifty seven tools and hundreds of functions available.[7] Then the manipulator designers should study all these tools and functions on his manipulator. And the other problem is, there is no logical way to array these tools. Therefore lots of bugs and deficiencies come in the programming process. As a result of this the manipulator designer has to waste a lot of time to meet with these challenges.

1.3 Available simulators

The most famous robot simulators and their features are as follows. This resource has been provided by the university of Essex U.K.[11]. The aim of this documentation attached to this thesis is to understand the available simulators and their features. These features are most important to develop the simulator for the universal manipulator.

1. EASY-ROB by Stefan Anton is a commercial Robot Simulation Tool with 3D graphic and animation [11]. The user can design a robot kinematic, move the robot in joint and Cartesian space, write a motion program, grab and release some thing, etc. A simple 3D-CAD System allows creation of basic elements such as block, cylinder, pyramid, cone and sphere in order to model a robot, tool and bodies. The user can rotate and translate the world view, zoom in and out and do a lot more. No additional graphic power is required. But this simulator is based on kinematic. Dynamic behavior of the robotic manipulator is not taken into account.

2. Encarnaco Robot Simulator by Luiz Felipe Rudge Encarnacao is a robot simulation that provides a full 3-Dimensional environment (wire frame graphics) with one fully moveable robot (5 axis)[11]. Control can be exercised via high level control mechanisms (i.e. grab, move and placing objects) or manually directed from the keyboard. There are several possible views (2 display areas and 15 possible virtual cameras). A camera can be placed in one "aeroplane" (i.e. allowing the user to "fly it" and use the resulting

perspective as their view). With a mouse you point & click on an object and to make the robot grab it. Another click on some possible local will cause the robot to place the robot there. With keyboard you can control all parts of the robot and fly the "aeroplane". It runs on MS Windows 3 or above.

3. MATLAB Robotic Toolbox by Peter I. Corke [6]. The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators. But there is no direct simulating facility. There is tools to develop the robotic manipulator. These parameters are encapsulated in MATLAB objects and it provides many functions that are useful in robotics including such things as kinematics, dynamics, and trajectory generation. The Toolbox is useful for simulation as well as analyzing results from experiments with real robots. But the reverse kinematic can not solve all type of non-redundant manipulators. The tool box designer (Peter I. Corke) has provided the close from reverse kinematic solution for standard manipulators like PUMA 560 , Stanford arm.

4. Melbourne-Robots is a robot simulator written by undergraduates Andrew Conway and Craig Dillon on a Silicon Graphics workstation for their electrical engineering project at the University of Melbourne [11]. There is a latest user manual (inc. the mathematics) but not much in the way of installation instructions (ftp address no longer valid).

5. Robotica is a collection of robotics problem solving functions for the Mathematica package [12]. This is the study pack for robotic student. It has the capability of reading external simulation (e.g., SIMNON) output files and displaying the motion of the robot when subjected to the sequence of joint variables. It requires Mathematica and X-windows.

Table. 1.1 The comparison between the developed simulator and the existing simulators

	Simulator	DOF(n)	forward kinematic	reverse kinematics	designing cost	trajectory planning	deigning easiness	dynamic modeling	data visualizing	controlling
1	EASY-ROB	n>6	available	not available	high	pt to pt	medium	not available	3D viewer	not available
2	Encarnaco Robot Simulator	n=6	available	available n=6	high	–	medium	not available	3D viewer	not available
3	MATLAB Robotic too box	n =N	available	available n<6	low	pt topt	low	not available	3D viewer	not available
4	Melbourne-Robots	n>6	available	not available	low	–	–	not available	2D viewer	not available
5	Robotica	n=N	available	not available	low	pt to pt	high	not available	3D viewer	not available
6	Simderella	n>6	available	available n>6	high	–	low	not available	3D viewer	not available
7	***Developed simulator***	***n =N***	***available***	***available***	***low***	***pt to pt***	***high***	***available***	***3D viewer***	***available***

6. Simderella is a popular simulator that was released to the world in 1993 by Patrick van der Smagt of the University of Amsterdam [11]. It came out of his research into neural networks and robot control. But this is developed for 6 axis manipulator. The original software consisted of three programs, connel: the controller, simmel: the simulator, bemmel: the X-windows oriented graphics back-end. Simmel is the part which actually simulates the robot. It performed matrix multiplications, based on the Denavit Hartenberg method & calculated velocities with the Newton-Euler scheme. But this simulator is designed for the non-redundant manipulator. These theories are not valid for the universal manipulators.

To compare the above simulating tool, no one has given a unique solution to the manipulator designers. Among these software tools, MATLAB tool box is one of the most used platforms for the modeling and simulation of various systems. But it is not a direct simulator. There are tools and functions to develop the manipulator simulation. It is a time wasting method and lots of bugs and deficiencies should be solved during the programming process. And also it is unable to give a total solution for the reverse kinematic and also the controlling of the system has been disregarded.

1.5. Aims and objectives

The Overall aim of the project is to give a complete solution to simulator users for rectifying the above weaknesses of the existing manipulators. In this scenario the developed simulator is the total solution for the universal serial manipulators. The final solution is combination of forward & reverse kinematic, dynamic solution, trajectory planning and manipulator controlling. But in this thesis only the kinematic behavior of this simulator is discussed. The 3D graphical interface is ideal to avoid the difficulties of programming and overcoming the bug fixing while in the programming. The graphical way is the easiest method to visualize the manipulator geometric in the Cartesian space and the joint space and behavior of the manipulator in the actual environment. To improve the functionality of the program and correct sequence is important to reduce the processing time. Designing the correct sequence to process the program is one of the objectives in this project.

Major objective of this project is designing the correct simulating tool for universal serial link manipulator. The new concept of this is to design any degree of freedom for serial link manipulator with any link combination and features. And the other aim of this thesis is to find a correct kinematic module supportive to the simulating requirement for the universal simulator. In these phenomena, the kinematic modeling of the project supports the study of the geometric and time based properties of the motion, and in particular how the various links move with respect to one another. Under this aspect, solving the forward and the inverse kinematic is essential to develop the kinematic algorithm to software coding.

1.6 Literature survey

Valuable support was provided by the Literature survey to direct this project in the correct direction. There are so many techniques used to solve the robotic manipulator kinematic and depending on the applicability, their capabilities are different form the different platform. Therefore the literature survey played a vital role to find the adaptive method for the required simulator application with great efficiency. Hence the literature survey is done to find the algorithms for the forward kinematics and the reverse kinematics. They are as follow,

1.6.1 Forward kinematics modeling for the universal manipulator

The forward kinematic directly supports to find the position and orientation of the link of the design manipulator. The Potential difference in the simultaneous links positions are used to draw the link in the 3D interface. There are several methods available for forward kinematic modeling to understand the motion without respect to force. They are as follows,

1. Artificial intelligent application
2. Matrix method of the systematically assigning the co-ordinate systems.(D-H Parameter)

The Artificial intelligent application like Genetic algorithm & Artificial Neural network [17] is ideal and accurate for the fixed manipulator. When it is used for the universal model different RP combination are provided. Then modeling is very difficult and mathematically complex. Then it is more time consuming while in the simulating operations. Therefore it is not suitable for the on-line application.

Solvability matrix method of systematically assigning the co-ordinate system is the mathematically elegant and also it is the fastest. Therefore this method is used to solve the forward kinematics.

For forward kinematic modeling, frames are assigned to each link of the manipulator starting from the base to the end-effector. The homogeneous transformation matrices relating the frame attached to successive links describe the spatial relationship between adjacent links[4]. The composition of these individual transform matrices determines the overall transform matrix, describing tool frame with respect to base frame.

The task to be performed by a manipulator is stated in terms of the end-effector location in space. The values of joint variables required to accomplish the task are computed using the inverse kinematic model. To find the location in space, at any time, the joint variable values are substituted in the forward kinematic model. This chapter 4 describes the problem of formulation of forward kinematic model. The inverse kinematic model formulation will be discussed in the chapter 5.

1.6.2 Alternative Methods of Inverse kinematic solutions

There are several possibilities to approach the inverse kinematic problem IK. Most of these approaches are based on some sort of search technique [9].

Algebraic approach: this approach is necessary, if one wants to calculate all possible solutions of the IK problem [5]. Given a specified linkage, one solves the forward kinematics problem explicitly. This in general leads to a set of nonlinear algebraic expressions in the state variables of the linkage. Given an end-effector position, the problem now is to algebraically solve this system of equations for the unknown state variables. Although it is theoretically possible to solve these using symbolic computation techniques, it is generally beyond today's computing power. Nevertheless, there exist some successful algebraic approaches for a few types of linkages.

Neural networks: recently, [10] an increasing number of neural network approaches have been suggested in the area of robotics. Since this requires teaching the linkage and thus highly depends on its actual geometry, these approaches are not very useful for simulation purposes.

Genetic programming and *genetic algorithms:*[10] due to steadily increasing computing power these techniques get more and more interesting in the area of Computer Animation especially when it comes to highly complex virtual environments. Genetic algorithms are directed search algorithms which try to find the solutions by mimizing the natural process of evolution. Currently, however, this approach is useless in VRML since today's browsers are not capable of providing the computing power necessary to achieve anything near real-time.

Graphical analyzing method: In the closed form solution, [9] joint displacement are determined as explicit functions of the position and orientation of the end-effectors and also the solution method based on the analytical algebraic or kinematics approaches, giving expressions for solving unknown joint displacements. In this project, the requirement of inverse kinematics solving is to find joint variable by using the user defined trajectory Cartesian space. The inverse kinematics approach of this project is to find the joint variable for the universal manipulator model. In the universal manipulator model inverse kinematic techniques should be defined for any kind of degree of freedom (DOF) with any joint combinations (revolute, prismatic joint combination).

Jacobian based approach: [30] one idea which proves to have the highest efficiency for most industrial applications is based on the so-called *manipulator Jacobian.* Roughly speaking the manipulator Jacobian is a matrix which describes the relationship between joint and end-effector velocities. Given the velocity of the end-effector, the Jacobian allows to recalculate the corresponding joint velocities simply by solving a system of linear equations. This can be used to deduce a gradient-based iterative technique for solving IK.

The fist four solutions may not be possible to solve all type of manipulators for the online simulation bases. A sufficient condition for non-redundant manipulator to possess close form solution is that both its three consecutive joint axes interest and its three consecutive joint axes are parallel and also the solving method & techniques are different from manipulator to manipulator. Therefore the first four methods of solution techniques are very difficult to model for these types of application.

The Jacobian based approach is the interesting and efficient method and it can easily model by using the computer application. But generally, there are four methods to solve the inverse kinematics. Then the next challenge is to find the most suitable method to solve the inverse kinematic of this project. Each and every method of Jacobian base solutions has different kinds of advantages and disadvantages. They are as follows.

1.6.3 Jacobian based numerical solution

The interactive algorithms are used to reverse kinematic in the Jacobian based solutions. Various types of Jacobian methods can be used to solve the inverse kinematics in different manipulators with different limitations [11]. And also there is no limitation for non redundant and it can be used for redundant and non-redundant both. The Jacobian base solutions are not actuated compared to the close form solutions but they can solve for the universal type by using various techniques and it is very efficient. The most common methods are,

1- Jacobian Transpose
2- Jacobian Pseudo inverse
3- Damped Least Squares methods

Jacobian transpose is the fastest method and it can be applied for any kind of degree of freedom but the results suffer in the singularity area and the degenerate case. The Jacobian transposes method and the optimization-based Newton- Raphson technique can stop in local minima. Therefore the interactive solution may not converge in those areas. But the Taylor series expression can be used to solve the inverse kinematic on this region [13]. Pseudo inverse have also the same problem on the singularity and degenerate case and it is more accurate. But there are lots of arguments to solve the inverse kinematics in non-redundant manipulator and also it is a very difficult model for different RP (revolute, prismatic) combinations.

The damping least square method is well behaved near the singularity. But it is very difficult to model for the different RP combination and the convergence rate is too slow. Therefore this is not suited for the on-line application.

The Jacobin transpose is the best method to solve this simulator inverse kinematics because it is faster and efficient than the others. But the direct result does not guarantee in near singular region and degenerate case. Therefore it should converge by using Newton-Raphson techniques [30]. In the singular region Jacobin transpose is not valid. Then the Taylor series can be used to solve the reverse kinematics

CHAPTER 2

Design Methodology

There are two research components used in this simulating tool. They are forward kinematics and the reverse kinematics. The supportive way of kinematic modeling is to develop this simulating tool as shown in the figure 2.1. Using the kinematic control equation, algorithms was developed for the forward and inverse kinematics to solve the Cartesian space and the joint space parameters that are related to simulating process.

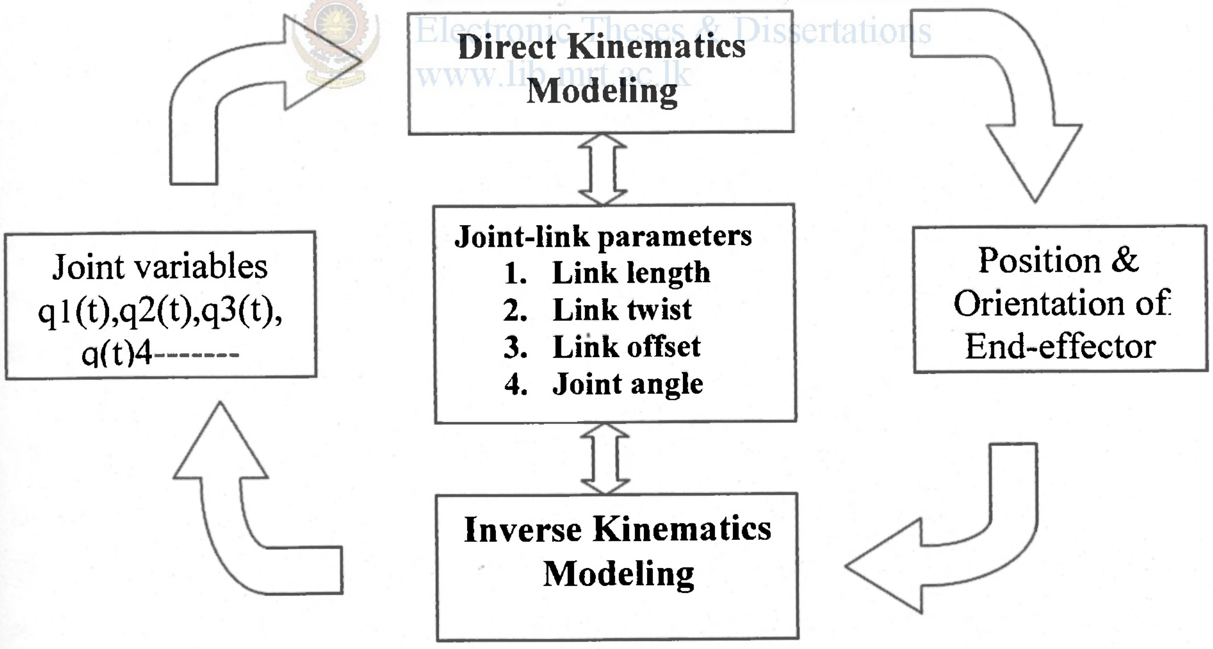

Direct Kinematics Modeling
Joint-link parameters
1. Link length
2. Link twist
3. Link offset
4. Joint angle
Joint variables q1(t),q2(t),q3(t), q(t)4-------
Position & Orientation of End-effector
Inverse Kinematics Modeling

Fig. 2.1 The direct and inverse kinematics model

2.1 Manipulator modeling fundamentals for the dynamic simulator

Robot manipulator modeling consists of a geometrical definition along with a kinematical description of the linkages. The geometrical definition can be accomplished by creating representative three-dimensional (3D) computer-aided graphic models whereas the kinematic entities describing the relations between links, velocities, accelerations and other characteristics of the manipulator can be obtained from robot kinematic theory. The structural design of the components of a robot manipulator uses the optimized geometric entities resulting from the previous stages. Though there are numerous software packages available for 3D modeling and motion simulation,[10] there is no single all inclusive packages that could produce the exact physical (3D characteristics) and functional description (motion planning) of the robot. The simulation itself consists of the kinematic and dynamic parts depending upon whether or not the actuator forces and torques are considered when generating motion trajectories. Once the functional description of the manipulator is finalized, any of the widely available solid modeling tools could be used for the accurate description of the robot's 3D geometry and an extensive structural analysis could be performed. A detailed discussion of structural modeling is shown in figure 2.1 and the highlighted areas of this figures is the scope of this thesis. It comes sunder the kinematic. It should be solved to develop the dynamic constraint and the trajectory planning.

The Visual C++ (version 6.0) is used for this simulator designing because it is the base language and processing time is less compared with the other programming tools [15]. The final interface is the total solution for universal manipulator simulations. The software tool is the final output of this project. The kinematics module helps to find the manipulator geometry and the joint angles. But the software tool is the combination of kinematics, dynamics and the trajectory planning.

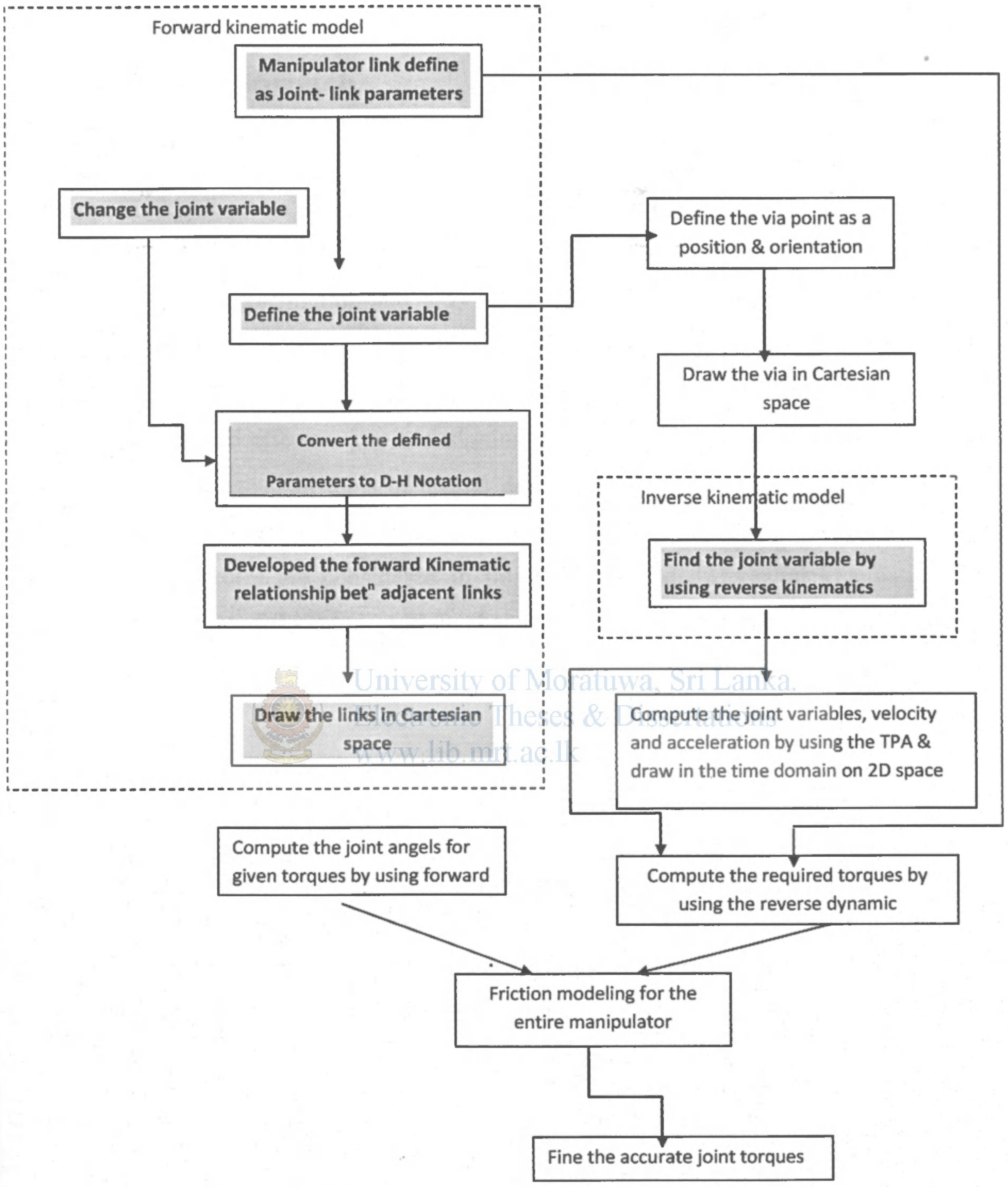

Forward kinematic model
Manipulator link define as Joint- link parameters
Change the joint variable
Define the joint variable
Convert the defined Parameters to D-H Notation
Developed the forward Kinematic relationship betn adjacent links
Draw the links in Cartesian space
Define the via point as a position & orientation
Draw the via in Cartesian space
Inverse kinematic model
Find the joint variable by using reverse kinematics
Compute the joint variables, velocity and acceleration by using the TPA & draw in the time domain on 2D space
Compute the joint angles for given torques by using forward
Compute the required torques by using the reverse dynamic
Friction modeling for the entire manipulator
Fine the accurate joint torques

Fig.2.2 structural designs for the proposed simulator

2.2 Designing approach

The supportive way of the kinematic behavior is described in this chapter. It is helps to find the manipulator geometric parameters in the Cartesian space and the joint space. To program the end-effector motion and link- joint motions, a mathematical model of the manipulator is required to refer to all geometrical and/or time base properties of it motion. Kinematic model describes the spatial position of the joint and links, and position and orientation of the end-effector. [5] In designing a robot manipulator, kinematic plays a vital role. The mathematical tools of spatial descriptions will be discussed in the next chapter are used in the modeling of robotic manipulator. And the forward kinematic and inverse kinematic roles are considered in this chapter. The differential kinematic of manipulators refer to differential motion, that is, velocity, acceleration, and all higher order derivative of joint-links.

2.2.1 Forward kinematic approach

The Forward kinematics is the problem of finding the unique location (position and orientations) of the links and the end-effector of the robot for the given set of joint values. The position and orientation of link end effector position are used to draw the link in the Cartesian space and draw the via point in trajectory.

2.3.2 Inverse kinematic approach

The inverse kinematics is used to calculate the joint variable for given via points that have to be defined in the Cartesian space. Solving inverse kinematic is the real challenge for the simulator manufacturers. Specially for the universal model. The solving of the problem of kinematic equation of a manipulator is a nonlinear one. By using numerical techniques two unique solution have been obtained for inverse kinematic equation for universal manipulators. They are Jacobian transformation base Newton-Rhapson techniques and the Taylor expression.

In addition to dealing with static positioning problems, author may wish to analyze manipulators in motion. Often in performing velocity analysis of a mechanism it is convenient to define a matrix quantity called the Jacobian of the manipulator. The Jacobian specifies a mapping from velocities in joint space to velocities in Cartesian space. Manipulator Jacobian is the most important to find the inverse kinematic solution in this manipulator. The nature of these mapping changes as the configuration of the manipulator varies. At the singularities the existing mapping is not invertible. Then another mapping system like Taylor expression should be considered to solve the joint space parameters. An understanding of the phenomenon is important to designers and the users of the manipulator.

CHAPTER 3

Forward Kinematics Modeling Theories

Robotic manipulator modeling consists of a geometrical definition along with a kinematical description of the linkages. The geometrical definition can be accomplished by creating representative three-dimensional (3D) graphic models whereas the kinematic entities describing the relations between links, velocities, accelerations and other characteristics of the manipulator can be obtained from robot kinematic theory based on Denavit-Hartenberg(D-H) parameters[12]

3.1 Mechanical structure and notation of the manipulator

Typical robots are serial-link manipulators comprising a set of bodies, called links, in a chain, connected by joints which allow linear or revolute motion between connected links each of which exhibits just one degree of freedom (DOF) are not common. For a manipulator with *n* joints numbered from 1 to *n*, there are *n*+ 1 links, numbered from 0 to *n*. Link 0 is the base of the manipulator, generally fixed, and link *n* carries the end-effector. Joint *i* connects links *i* and *i*- 1.[5]

A link may be considered as a rigid body defining the relationship between two neighboring joint axes. A link can be specified by two numbers, the *link length* and *link twist*, which define the relative location of the two axes in space. The link parameters for the first and last links are meaningless, but are arbitrarily chosen to be 0. Joints may be described by two parameters. The *link offset* is the distance from one link to the next along the axis of the joint. The *joint angle* is the rotation of one link with respect to the next about the joint axis.[7]

3.2 Denavit and Hartenberg notation for manipulator configuration

Denavit and Hartenberg have proposed a matrix method of systematically assigning coordinate systems to each link of an articulated chain. This method is mathematically elegant and it is used to develop the forward kinematics algorithm for this project. Another advantage of this method is, it is very logical and easy to model the any kind of degree of freedom serial link manipulator through this method.[5]

To facilitate describing the location of each link we affix a coordinate frame to it. The frame i is attached to link i. The axis of revolute joint i is aligned with z_{i-1}. The x_{i-1} axis is directed along the normal from z_{i-1} to z_i and for intersecting axes is parallel to $z_{i-1} \times z_i$. The link and joint parameters may be summarized as follows (see figure 3.2):

Link length (a_i) - the offset distance between the z_{i-1} and z_i axes along the x_i axis;
Link twist (α_i) - the angle from the z_{i-1} axis to the z_i axis about the x_i axis;
Link offset (d_i) - the distance from the origin of frame i- 1 to the x_i axis along the z_{i-1} axis;
Joint angle (θ_i) -the angle between the x_{i-1} and x_i axes about the z_{i-1} axis.

For a revolute axis θ_i is the joint variable and d_i is constant, while for a prismatic joint d_i is variable, and θ_i is constant. In many of the formulations that follow we use generalized coordinates, q_i, where

$$q_i = \theta_i \text{ for a revolute joint}$$
$$q_i = d_i \text{ for a prismatic joint}$$

The Denavit-Hartenberg (DH) representation results in a 4x4 homogeneous transformation matrix.

$$
{}^{i-1}A_i = \left[\begin{array}{ccc|c}
\cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\
\sin\theta_i & \cos\theta_i\sin\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\
0 & \sin\alpha_i & \cos\alpha_i & d_i \\
\hline
0 & 0 & 0 & 1
\end{array}\right]
$$

(3.1)

Translation is qualitatively different form rotation in one important respect. In rotation, the origins of two coordinate frames are same. This invariance of the original characteristic allows the representation of rotation in 3-D space as a 3x3 rotation matrix $\boldsymbol{R_i}$. However, in translation, the origins of translated frame and original frame are not coincident then the origins have to be translated by a 3x1 translation vector $\boldsymbol{P_i}$ [5].

Then,

$$
{}^{i-1}A_i = \left[\begin{array}{c|c}
\text{Rotation matrix, } \boldsymbol{R} & \text{Translation vector, } \boldsymbol{P} \\
(3X3) & (3X1) \\
\hline
\text{Prospective} & \text{Scale factor} \\
\text{Transformation matrix} & (1X1) \\
(1X3) &
\end{array}\right]
$$

(3.2)

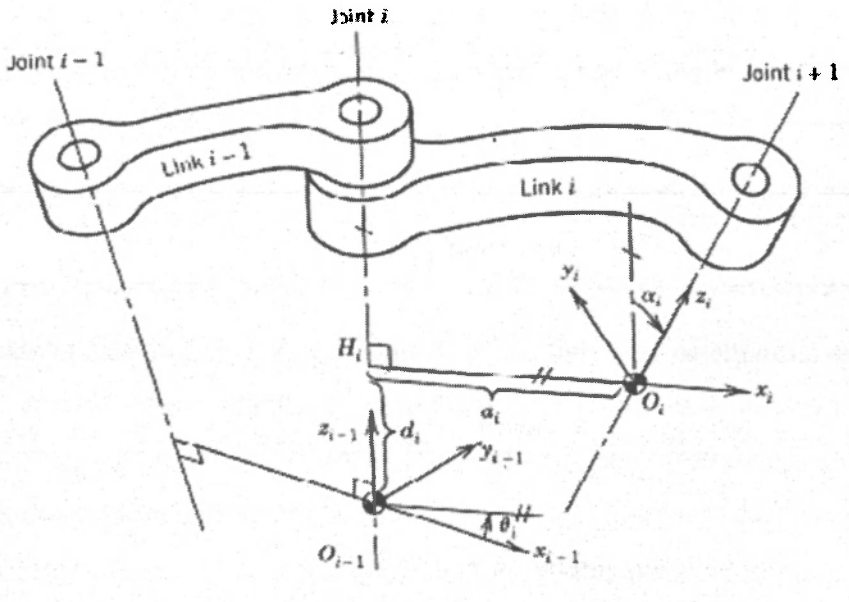


Fig. 3.1-Denavit –Hartenberg parameters

The Rotation matrix, $\boldsymbol{R_i}$ represents the orientation of the entire link and the translation vector, $\boldsymbol{P_i}$ represents the position of same link of the manipulator. These 3D valves can be used to draw the link position and orientation with respect to the previous frame in the Cartesian space. [5]

For the next link rotation matrix $\boldsymbol{R}_{i+1}$ can be written as follows,

$${}^{0}R_{i+1} = {}^{0}R_i \, {}^{i}R_{i+1} \tag{3.3}$$

And the translation matrix can be written as follows,

$${}^{0}P_{i+1} = {}^{0}P_i + {}^{0}R_i \, {}^{i}P_{i+1} \tag{3.4}$$

Where,

$${}^{0}T_{I} = \begin{vmatrix} {}^{0}R_{i+1} & {}^{0}P_{i+1} \\ 0 & 1 \end{vmatrix}$$

The $^{i-1}\mathbf{A}_i$ matrices can be used to develop a matrix of expressions for the forward kinematics equation, $\boldsymbol{T}$, for the manipulator arm. $\boldsymbol{T}$ is a 4 × 4 matrix which gives the position and orientation of the end-effector with respect to the base frame as a function of each of the joint variables q_i.

$$T = A_1^0(q_1)A_2^1(q_2)\ldots\ldots\ldots\ldots A_n^{n-1}(q_n) \tag{3.5}$$

This method can be used to calculate the position and orientation of the serial link manipulator [5]. These calculated values can be used to draw the links in the Cartesian space on the user graphical interface. The used algorithm to draw the manipulator on the Cartesian space is as follows.

3.3 Algorithm for the manipulator link frame assignment

This algorithm assigns frame and determine the D-H –parameters for each link of an n-DOF serial link manipulator.[5] Both, the first link 0 and the last link ***n,*** are connected to only one other link and thus, have more arbitrariness in frame assignment.[29] For this reason, the first (frame {0}) and the last (frame {n}) frames are assigned after assigning frames to intermediate links, link *1* to link ***(n-1).***

The displacement of the each joint-link is measured with respect to a frame and therefore the initial position of each link needs to be clearly defined. The initial position of a revolute joint is when the joint angle is ***θ*** **,** while for the prismatic it is when the joint displacement ***d*** is between working range. (i.e. maximum and minimum of the variable displacement).

Because of mechanical constraints, the range of the joint motion possible is restricted and, in some cases, this may result in a home position that is unreachable. In such cases, the home position is redefined by changing the initial manipulator joint and /or frame assignments. The new home position can be obtained by adding a constant value to the joint angles in case of revolute joint and to the joint displacement in the case of prismatic joint. This shifting of the home position is illustrated in figure 3.3.

Normally the manipulators, the initial position as zero position is designed in the joint variable. But this simulator has been designed to take any position of the joint variables in the working range as home position. Therefore the user has a good flexibility to select home position including zero position. Assigning frame of the manipulator designer should decide the home position of his manipulator.

The algorithm is divided into four parts. The first segment gives steps for labeling scheme and the second one describes the steps for frame assignment to intermediate links ***1*** to ***(n-1).*** The third and fourth segments give steps for frame {0} and frame {n} assignment, respectively.

Step 0- Identify and number the joints starting with base and ending with end-effector. Number the links from ***0*** to ***n*** starting with immobile base as 0 and ending with last link as n.

Step 1 – Align axis Z_i with axis of joint $(i+1)$ for $i = 0,1,\ldots,n-1$

Assigning frames to intermediate links-link *1* to link *(n-1)* for each link *i* repeat step 2 and 3.

Steps 2- The x_i axis is fixed perpendicular to both Z_{i-1} and Z_i axes and points away from Z_{i-1} . The origin of frame **{i}** is located at the intersection of Z_i and X_i axes. Three situations are possible.

Case 1- If Z_{i-1} and Z_i axes intersect. Choose the origin at the point of their intersection. The xi axis will be perpendicular to the plane containing Z_{i-1} and Z_i axes. This will give a_i to be zero.

Case2- if the Z_{i-1} and the Z_i axes are parallel or lie in parallel planes then their common normal are not uniquely defined. If joints i is revolute then X_i –axis is chosen along that common normal, which passes through origin of frame **{i-1}**. This will fix the origin and make d_i zero. If joint i is prismatic, X_i axis is arbitrarily chosen as any convenient common normal and the origin is located at the distal end of the link *i*.

Case 3- if Z_{i-1} and Z_i axes coincide, the origin lies on the common axis. If joint i is revolute, origin is located to coincide with origin of frame **{i-1}** and X_i axis coincides with X_{i-1} axis to cause d_i to be zero. If joint i is prismatic , X_i axis is chosen parallel to X_{i-1} axis to make a_i to be zero. The origin is located at distal end of link *i*.

Step3 – The Y_i axis has no choice and is fixed to complete the right-handed orthonormal coordinate frame **{i}.**

Assigning frame to link 0, the immobile base-frame **{0}**

Step 4 – The frame {0} location is arbitrary. Its choice is made based on simplification of the model and some convenient reference in workspace. The X_0 axis, which is perpendicular to Z_0 axis, is chosen to parallel to X1 axis in the home position to make $\theta_1=0$ or any where in working range. The origin of the frame{0} can be chosen at a convenient reference such as, floor, working table, and so on ,giving a constant value for the parameter d_1 zero. If joint 1 is prismatic, parallel X_0 –and X_1 axes will make θ_1 to be zero and origin of frame {0} is placed arbitrarily.

Step 5 - The Y_0 axis completes the right –handed orthonormal coordinate frame {0}.

Link n, the end effector, frame assignment-frame **{n}**

Step 6 -The origin of frame **{n}** is chosen at the tip of the manipulator, that is, a convenient point on the last link (the end-effector). This point is called the "tool point" and the frame {n} is the tool frame.

Step 7 – the Z_n axis is fixed along the direction of Z_{n-1} axis and pointing away from the link ***n***. It is the direction of "approach."

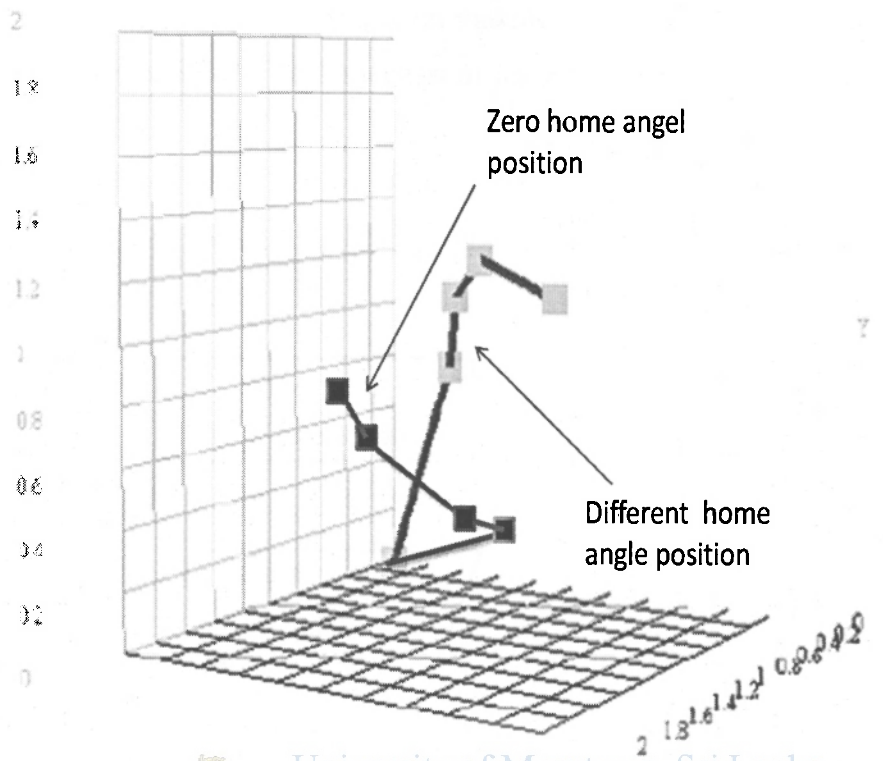


Fig.3.2-End-effector location in the different home position

Step 8- if joint ***n*** is prismatic; take X_n parallel to X_{n-1} axis. If joint ***n*** is revolute, the choice of the X_n is similar to step 4, that is, X_n is perpendicular to both Z_{n-1} and Z_n axes. Xn direction is the "normal" direction. The Y_n axis is chosen to complete the right-handed orthonormal frame {n}. The Y_n axis is the "orientation" or "sliding" direction.

Once the frames are assigned to each link, the joint parameter $(\theta_i\ d_i\ \alpha_i\ a_i)$ can be easily identified for each link, using which the direct kinematic model developed in designing the universal manipulator on this robotic simulator. In fixing the frames, it is desirable to make as many of the joint-link parameters zero as possible because the amount of the computation necessary in later analysis is dependent on these. Hence, where there is a choice in frame assignment, emphasis is on making a choice, which results in as many zero parameters as possible. The flow chart of forward kinematic modeling is shown in the figure 3.4.

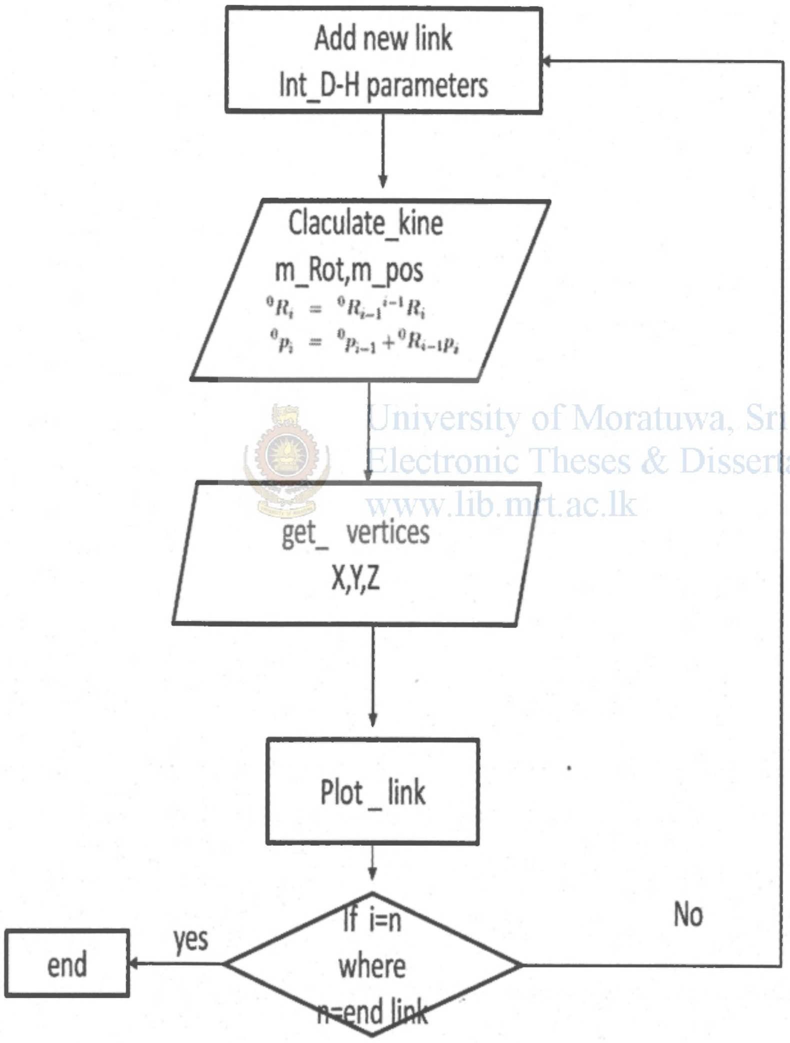


Fig. 3.3 - Flow Chart of Forward Kinematic

3.4 Manipulator Jacobian which relates to build the inverse kinematic algorithm

The manipulator Jacobian is a support to find the joint variable for the given end-effector position and orientation when it is moved in the desired trajectory path. Even though the position and orientation equations are non-linear, the relationship between the velocity of the distal end and the velocities of the joint angles is linear. If the forward kinematic problem is stated by $x = f(q)$, then numerical solutions to the inverse kinematic problem typically involve differentiating the constraint equations to obtain a Jacobian matrix[5]

$$J=\frac{\partial f}{\partial q} \tag{3.6}$$

And solving the linear matrix system

$$\dot{x}=J\dot{q} \tag{3.7}$$

Where,

$\dot{x}$ = 6x1 Cartesian velocity (desired angular and liner velocity) vector

$$\dot{x}=[\ v_x\ v_y\ v_z\ \omega_x\ \omega_y\ \omega_z\ ]^{\mathrm{T}} \tag{3.8}$$

$\dot{q}$= nx1 vector of n joint velocities

$$\dot{q} = [\theta_1\ \theta_2\ \theta_3\text{----------}\theta_n]^T \tag{3.9}$$

J= 6x n Manipulator Jacobian

The Jacobian is a function of the joint variable. The Jacobian function supports to develop the reverse kinematic modeling of this project. To solve the reverse kinematic is the key feature of this project and it will be explained in chapter 5 in detail.

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_x}{\partial \theta_2} & \cdots & \frac{\partial p_x}{\partial \theta_n} \\ \frac{\partial p_y}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_2} & \cdots & \frac{\partial p_y}{\partial \theta_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial a_z}{\partial \theta_1} & \frac{\partial a_z}{\partial \theta_2} & \cdots & \frac{\partial a_z}{\partial \theta_n} \end{bmatrix} \tag{3.10}$$

The manipulator *J*acobian, J, is a 6x *n* matrix (e.q.4.5) where *n* is the number of joints in the manipulator arm. The *i*th column of *J* can be thought of as two 3x1 vectors, J_{Li} and J_{Ai} , which are associated with the linear and angular velocities, respectively, of the tip of the robot arm due to the *i*th joint velocity. So we can partition J as follows:[18]

$$J_i = \begin{bmatrix} J_{Li} \\ J_{Ai} \end{bmatrix} \tag{3.11}$$

The Jacobian deals with small motions of the end-effector about its current position and arm Configuration, so each of the elements of J is a function of the joint variables, q_i . The first three rows of **J** (**J**$_L$) deal with the linear velocity of the end-effector with respect to the base coordinate system. Each column of **J**$_L$, vector **J**$_{Li}$, is formed by differentiating the expression for the position of the end-effector, which is given as the last column in **T,** as follows[18].

$$J_{Li} = \begin{bmatrix} \frac{\partial x}{\partial q_i} \\ \frac{\partial y}{\partial q_i} \\ \frac{\partial z}{\partial q_i} \end{bmatrix} \tag{3.12}$$

The last three rows of **J** ($\mathbf{J}_A$) deal with the angular velocity of the end-effector and are due to the angular velocity of the end-effector generated by each joint. There is no contribution to the angular velocity at the end-effector for prismatic joints, so:

$$J_{Ai}\,\dot{q}_i = 0 \text{ for prismatic joint } i$$

However a revolute joint i rotates the links i to n at the angular velocity ω_i as follows:

$$J_{Ai}\,\dot{q}_i = \omega_i = b_{i-1}\dot{\theta}_i \qquad (3.13)$$

Where

$\mathbf{b}_{i-1}$ is the unit vector pointing along the direction of the joint axis i. For revolute joint i, the rotation is about the z_{i-1} axis, by convention. In terms of coordinate frame i, $\mathbf{b}_{i-1}$ is represented as $[0, 0, 1]^T$.

The rotation matrix **R** can transform a vector in the i^{th} frame to one in the previous (i-1) frame. To determine $\mathbf{b}_{i-1}$ for Eq. (3.13), the rotation matrices to express the z_{i-1} axis with respect to the base frame as follows:

$$b_{i-1} = R_1^0(q_1) \text{--} R_{i-1}^{i-2}(q_{i-1}) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad (3.14)$$

To summarize above equations,

The manipulator Jacobian defines the relation between the velocities in joint space $\dot{q}$ and in the Cartesian space $\dot{\chi}$ expressed in frame i:

$$\dot{x}_i = j(q)_i\,\dot{q} \qquad (3.15)$$

or the relation between small variations in joint space δq and small displacements in the Cartesian space $\delta\chi$:

$$\delta\chi_i \approx J(q)_i\,\delta q \qquad (3.16)$$

The manipulation Jacobian expressed in the base frame is given by

$$^{0}J(q) = [^{0}J_1(q)\ ^{0}J_2(q)\ \ldots\ ^{0}J_n(q)] \qquad (3.17)$$

With,

$^{0}J_i(q) = \begin{bmatrix} z_{i-1} \times {}^{i-1}p_n \\ z_{i-1} \end{bmatrix}$ for a revolute joint

$^{0}J_i(q) = \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix}$ for a prismatic joint

Where,

z_{i-1} and $^{i-1}p_n$ are expressed in the base frame and × is the vector cross product [20].
Expressed in the i^{th} frame, the Jacobian is given by

$$^{i}J(q) = \begin{bmatrix} ({}^{0}_{i}R)^T & 0 \\ 0 & ({}^{0}_{i}R)^T \end{bmatrix} {}^{0}J(q) \qquad (3.18)$$

This function returns $^{i}J_{(q)}$ (i = 0 when not specified) for the end-link.

3.6. Jacobian singularities

Those manipulator configurations at which J become noninvertible are termed as Jacobian singularities and the configuration is itself called singular. At the singularities, the Jacobian matrix loses its rank and becomes ill conditioned at values of joint variables q at which its determinant vanishes.

The study of manipulator singularities is of great significance for the following reasons:

1. It is not possible to give an arbitrary motion to end-effector; that is, singularities represent configurations at which structural mobility of the manipulator is reduced.
2. At a singularity where no solution any exists for the inverse Jacobian problem.
3. In the neighborhood of a singularity, small velocities in the Cartesian space require very high velocities in the joining space. This causes problems when the manipulator is required to track a trajectory that passes close to the singularity.

To solve this problem in the inverse kinematics of this simulator, the Taylor series is used to solve the joint space parameters [7]. Used method of calculating the inverse kinematics will be explained in next chapter.

CHAPTER 4

Inverse kinematic Approach on the Simulator

Inverse kinematics plays a key role in this simulating tool. Inverse kinematics is more difficult than the forward because there is no unique solution for it. The inverse equations are non-linear simultaneous equations, involving transcendental functions. The number of simultaneous equations is also generally more than the number of unknowns, making some of the equations mutually dependent.

4.1 Used methods for inverse kinematic solving techniques

The key role of this project is the configuration of a joint space for a continuous function of one or more real scalars; or a rotational joint, the scalar is the angle of the revolute joint or length of the prismatic joint.[11].

The complete configuration of the manipulator is specified by the scalars $q_1, q_2, q3$----qn describing the joints configurations. Assuming there are n joints and each q_i value is called a *joint variable*. Certain points on the links are identified as end-effector. The end effector current position s is a function of the joint variable [17]. The target position of the end-effector is t. *(see figure 5.1)*

The desired change in position of end-effector is δx. Then,

$$\delta x = t\text{-}s \qquad (4.1)$$

Let,

$$\overline{\delta x}=\bar{t}\text{-}\bar{s} \qquad (4.2)$$

The joint angles are written as a column vector as $q = (q_1, q_2 \ldots \ldots, q_n)^T$. The end effector positions are function of the joint variable; this fact can be expressed as $\bar{s}=\bar{x}(\theta)$. The inverse kinematic problem is to find values for the θ_i so that,

$$t_i = s_i(\theta) \quad (4.3)$$

For this, the functions $\delta\chi$ are linearly approximated using the Jacobian matrix. The Jacobian matrix J is a function of the q values and is defined by

$$J = \frac{\delta x}{\delta q} \quad (4.4)$$

The basic equation for forward dynamic that describe the velocities of the end effector can be written as follows (using dot notation for first derivatives)

$$\dot{x} = J(q)q \quad (4.5)$$

The Jacobian leads to a method for solving equation (5.1) , suppose the current values for the q, $\bar{s}$ and $\bar{t}$. From these, the Jacobian $J=J(q)$ is computed.
Then the update valve Δq for the purpose of incrementing the joint variable q by Δq,

$$q = q + \Delta q \quad (4.6)$$

The change in end effector position caused by this change in joint angles can be estimated as,

$$\Delta\bar{x} \approx J\Delta q \quad (4.7)$$

4.2 Inverse kinematics algorithm

Selected method of the inverse kinematic calculation is shown in the figure 5.2. The solution is the combination of the Newton –Raphson base Jacobian transformation method and the Taylor expression. The Taylor expression is used to calculate the inverse kinematic near the singularity region. Used method is explained as follows.

4.2.1 Computed inverse kinematic model by using Newton-Raphson technique

Let mj=0, it is based to solve the inverse kinematic by using the Newton –Raphson techniques. [30]

$$^{0}T_{n}(q^{*}) = {}^{0}T_{n}(q + \delta q) \approx {}^{0}Tn(q)\delta T(\delta q) = T_{obj} \qquad (4.8)$$

Where, T_{obj} is the end-effector transformation matrix which is selected by user in his trajectory Cartesian point.

Then we can write the eq^{n} 5.9 as follows and the $\delta T(\delta q)$ assumes the form of I and Δ

$$\delta T(\delta q) = ({}^{0}T_{n}(q))^{-1}T_{obj} = I + \Delta \qquad (4.9)$$

Where the I is the identity matrix & Δ can assume as follows. [7]

$$\Delta = \begin{bmatrix} 0 & -\delta_x & \delta_y & d_x \\ \delta_z & 0 & -\delta_x & d_y \\ -\delta_y & \delta_x & 0 & d_z \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad (4.10)$$

From the eq^{n} 4. 10 we can write the end-effector differential motion as follows,

$$^{n}\delta\chi = [d_x\ d_y\ d_z\ \delta_x\ \delta_y\ \delta_z]^{T} \qquad (4.11)$$

Then jointing variable δq can be found as follows

$$^{n}\delta\chi \approx {}^{n}J(q)\delta q \qquad (4.12)$$

If δq is "small enough" the interaction stops; otherwise above procedure is repeated with new estimate value,

$$q_{i+1} = q_i + \delta q \tag{4.13}$$

The numerical procedure finds only *one* solution, i.e., the one to which the iteration converges.

4.2.2 Computed inverse kinematic model by using Taylor series

If mj = 1, it is based on the following Taylor series and it can be written as follows.

$$^{0}T_n(q^*) = {}^{0}T_n(q + \delta q) \approx {}^{0}T_n(q) + \sum_{i=0}^{n} \frac{\partial^{0}Tn}{\partial qi} \delta qi \tag{4.13}$$

The partial derivatives of $\frac{\partial^{0}Tn}{\partial qi}$ can be calculated as follows,

$$\frac{\partial^{0}Tn}{\partial qi} = {}^{0}T_{i-1}\, Q_i\, {}^{i-1}T_n \tag{4.14}$$

where the Q_i is

$Q_i = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ for a revolute joint

$Q_i = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$ for a prismatic joint

Given the desired position represented by the homogeneous transform T_{obj}, this function returns the column vector of joint variables that is corresponding to this position [23]. On return, the value converge is true if the Procedure has converge to values that give the correct position or else it is false.

4.3 Calculate the singularities and test the convergence

The NEWMAT 11 Mathematic library in c++ is used to calculate the inverse kinematic convergence in simulator and the NEWMAT 11 is commonly used for Object oriented programming in C++ [23]. In this case, it is used to check the convergence of Newton-Raphson base Jacobian inverse calculation if it fails then the end-effector trajectory point in the desired manipulator may be in the singularities region or out of the working space.

Then the inverse kinematics solves on the Taylor expansion and check the convergence. If it is convergence then it is in the singularity region or else then the selected trajectory point is out of boundary in working space [13]. It may be difficult to completely eliminate the possibility of unreachable positions and still get the desired motion. Second, if target positions are barely reachable and can be reached only with full extension of the links, then the situation is very similar to having unreachable targets. Unfortunately, the situation of target positions in unreachable positions is difficult to handle robustly. Many methods, such as the pseudo inverse or Jacobian transpose methods, will oscillate badly in this situation; however, Taylor expansion methods can still perform well with unreachable target positions. But The Newton-Raphson base Jacobian is more accurate than the Taylor expansion. Therefore the priority of the inverse calculation has been given to the Newton- Raphson base solution.

The convergence theory is the superior method to achieve correct value for the inverse kinematic solution. In order to perform the Taylor expansion and Newton-Raphson techniques of reverse kinematics, the real-time computation of any order derivatives of inverse kinematics of any serial manipulators can be used (n joints, rotary or prismatic) [30]. Set of tests measured how well the different methods converged accurately to fixed target positions.

For these tests, the target position is moved discontinuously and the joint angles are updated repeatedly until either the end effector reaches it position or fails to continue moving towards the end effector. In the both cases, thousand time tests for the convergence are separately done to find whether the target positions are reachable or not.

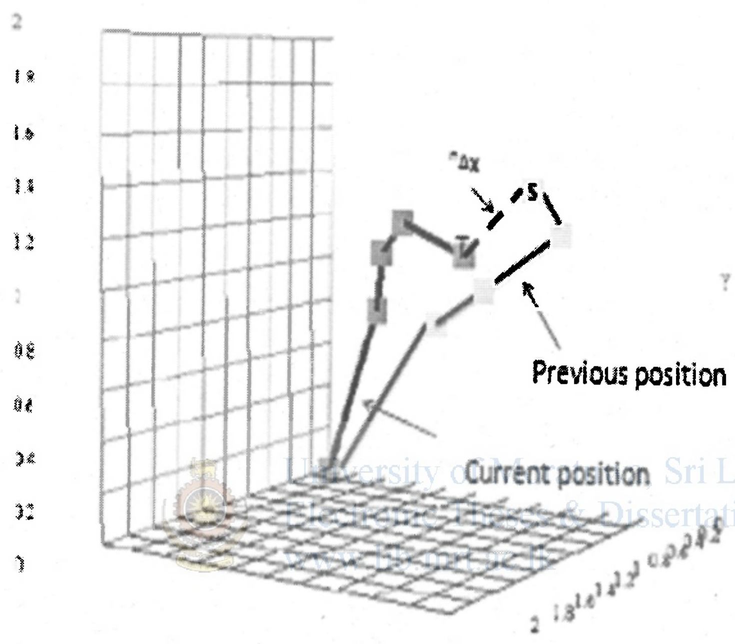


Fig. 4.1- Manipulator moves in the Cartesian space.

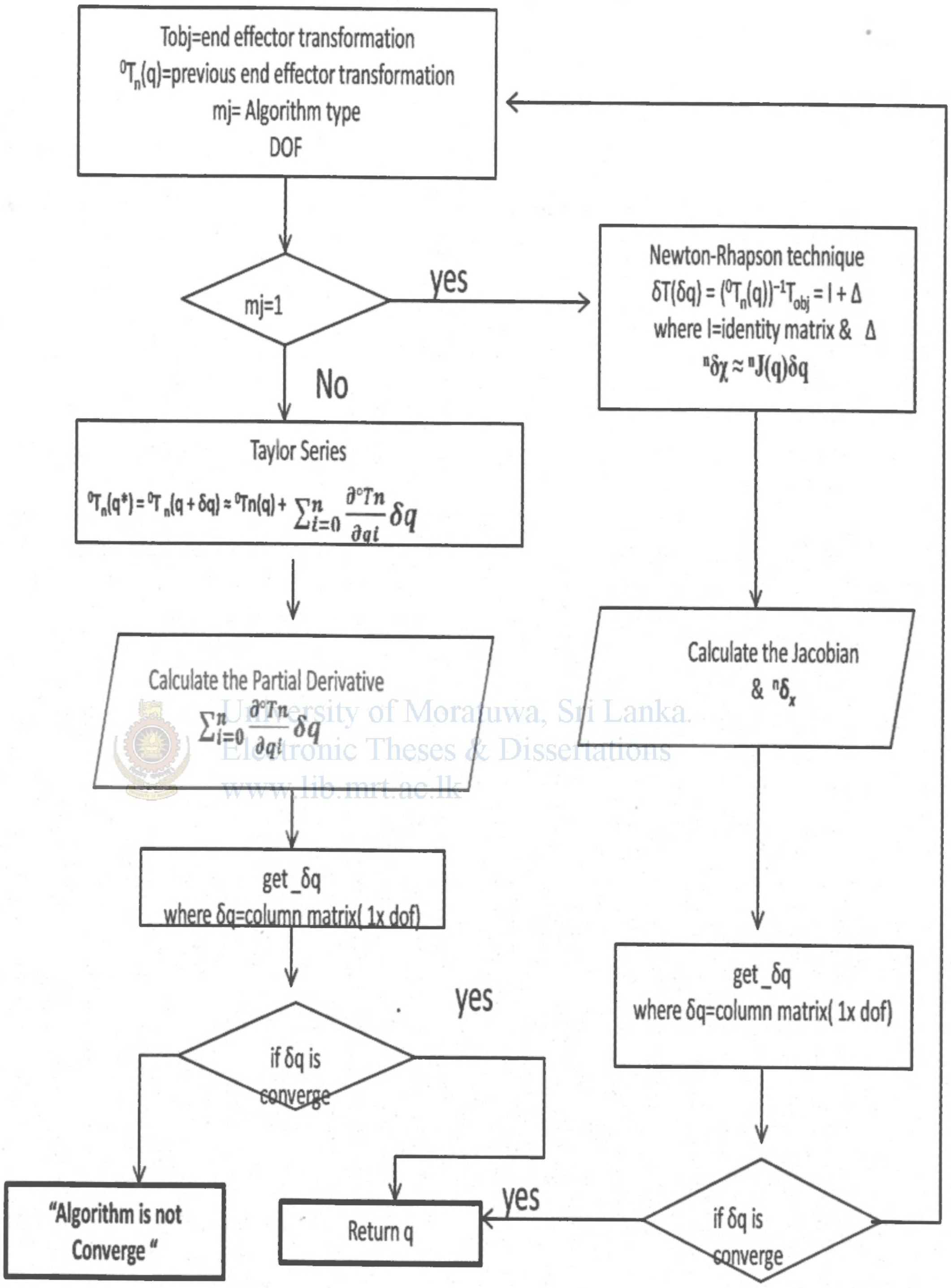

Tobj=end effector transformation
$^0T_n(q)$=previous end effector transformation
mj= Algorithm type
DOF
mj=1
yes
No
Newton-Rhapson technique
$\delta T(\delta q) = (^0T_n(q))^{-1}T_{obj} = I + \Delta$
where I=identity matrix & Δ
$^n\delta\chi \approx {}^nJ(q)\delta q$
Taylor Series
$^0T_n(q^*) = {}^0T_n(q+\delta q) \approx {}^0Tn(q) + \sum_{i=0}^{n} \frac{\partial^0 Tn}{\partial qi}\delta q$
Calculate the Partial Derivative
$\sum_{i=0}^{n} \frac{\partial^0 Tn}{\partial qi}\delta q$
Calculate the Jacobian
& $^n\delta_x$
get _δq
where δq=column matrix(1x dof)
get _δq
where δq=column matrix(1x dof)
yes
if δq is converge
"Algorithm is not Converge "
Return q
yes
if δq is converge

Fig. 4.2.The flow chart of Inverse Kinematic calculation

CHAPTER 5

Design the Software Tool

The developed dynamic simulator is an interactive 3D environment for viewing robot manipulator module as described in the tabulation dialog mode. In the developed robot manipulator, the jointing variables can be changed and it can be viewed on dialogs. And also the user can view the end effector orientation, visualizing dynamic stability and trajectories through time-based animation.

5.1 Object oriented programming

The object-oriented program is used to develop this simulating tool since OOP can describe each part of the robot as one object with its own properties and behavior. Even if C++ is not a perfect OO language, a lot of very useful libraries are available, and maintains very good efficiency for intensive computations. The selected software tool for this robotic simulator is Visual C++ 6. The visual C++ developing environment is well adaptive and more flexible for this type of application [15]. The object oriented programming is also considered to be better at modeling the real world than in procedural programming. It allows for more complicated and flexible interactions. The object orient programming systems are also easier for non-technical personnel to understand and easier for them to participate in the maintenance and enhancement of a system because it appeals to natural human cognition patterns.

For this application, an object orient programming approach is faster since many objects are standard across systems and can be reused for different kinds of modeling like kinematic, dynamic and trajectory planning.

5.2 Supporting c++ libraries

All kind of supportive tools and functions are not developed in the software tool programming process. There are many kinds of tools and techniques that can help in the developed software tool and they are freely available in web. This can be used in the programming process to save the time. C++ directly supports a variety of programming styles. In this, C++ deliberately differs from languages designed to support a single way of writing programs. For this software tool authors have used to two supportive libraries for the mathematical operation and developing the user interface

5.2.1 NEWMAT 11 mathematic library

To develop a tool for mathematical operation of the project is one of the major objectives of this program and this tool should support to the Visual C++ operation and it should be faster and mathematically elegant [16]. Among the various tools and libraries the NEWMAT11 is the selected tool to solve the mathematical operation in this project.

NEWMAT11 developed by Robert Davies [13] is used for mathematical operation in this project. The simulation of robotic manipulator models in an environment that provides "MATLAB like" features for the treatment of matrices. NEWMAT11 is a portable tool which uses the professional C++ programming.

The package is intended for scientists and engineers who need to manipulate a variety of types of matrices using standard matrix operations. Emphasis is on the kind of operations needed in statistical calculations such as least squares, linear equation solve and eigen values.

It supports matrix types

Matrix	rectangular matrix
SquareMatrix	square matrix

nricMatrix	for use with *Numerical Recipes in C* programs
UpperTriangularMatrix	
LowerTriangularMatrix	
DiagonalMatrix	
SymmetricMatrix	
BandMatrix	
UpperBandMatrix	upper triangular band matrix
LowerBandMatrix	lower triangular band matrix
SymmetricBandMatrix	
RowVector	derived from Matrix
ColumnVector	derived from Matrix
Identity Matrix	diagonal matrix, elements have same value

Only one element type (float or double) is supported.

The package includes the operations *, +, -, Kronecker product, Schur product, concatenation, inverse, transpose, conversion between types, sub matrix, determinant, Cholesky decomposition, QR decomposition, singular value decomposition, eigenvalues of a symmetric matrix, sorting, fast Fourier transform, printing and an interface with *Numerical Recipes in C.*

5.2.2 NT Graph3D graph library

3D Data visualizing on the 2D environment is one of the challenges of the project. After testing and referring to different methods, tools and libraries NT Graph3D was selected as a reference resource to develop the user Graphical interface. It was a supportive library to develop a graphical tool but it does not well match to the required interface. Therefore the concept of development of this tool has been used and the new interface for these 3D data visualization[27] has been developed.

NT Graph3D is the graphical library developed by Nikolai Teofilov. This is the key source which is use to develop the 3D interface. The NT Graph3D directly supports the visual C++. In the figure 2.2 the developed interface is shown.

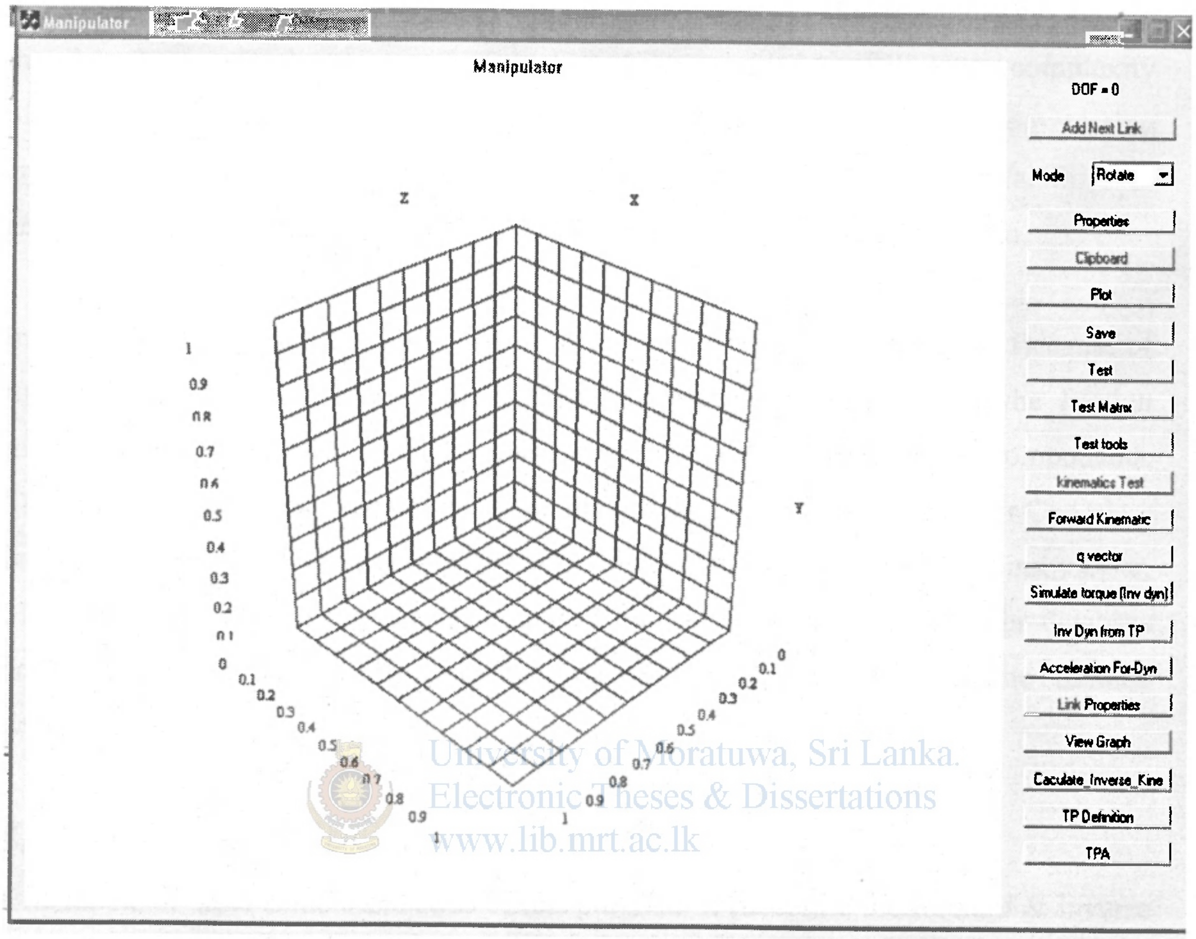


Fig. 5.1- Proposed Interface

5.3 Software architecture

This simulation architecture provides developers with the opportunity to easily extend the current simulation engine's abilities, without requiring modifications to the simulation engine code itself. In the event that developers need to add substantial new abilities to the simulator, the simple design of the simulation engine's architecture will make it easy to modify the simulation engine to incorporate these new abilities. This is the art of object oriented programming. In the design of this architecture, the object oriented programming has responded in three ways of this software to meet these programming requirements. They are, providing techniques for managing enormous complexity, achieving reuse of software components, and coupling data with the tasks that manipulate that data. In the visual c++ enough tools have been provided to affiliate the complexity of the program and the adaptability of the system tool. The designed programming architecture is shown in the figure 6.1. It has to be provided with the facilities to enhancing the simulation tool and the adaptability to the controlling application.

In this architecture repeatability of element and the tool has been eliminated. Because of four different modules (i.e. kinematic, dynamic, trajectory planning and the friction module developing) this simulator has been developed as individual components. Therefore the simulator executing is faster and efficient. In the CManipulator.Cpp class has seven major tools. They are serialization, calculation tool box, matrix output viewer, 3D graphics viewer, link class, 2D graphical output viewer and manipulator database access object. In this thesis only the related classes for the kinematic module development is discussed..

5.3.1 CManipulator Class

CManipulator class is the basic class of this program. It processes the forward & inverse kinematic algorithms.

5.3.2 Serialization Tool.

The Serialization is the class of data storing. This tool can be used to store the data and it can re-call the data. The different manipulator designing applications can be stored as a set of data. The stored data can be re-called and edit according to requirement of users.

5.3.3 Calculation Tool Box

The Calculation tool box is used to calculate all kinds of kinematic applications in this software. Not only the kinematic but also the dynamic and trajectory planning application can be calculated.

5.3.4 Matrix output viewer

Matrix output viewer tool is used to view the matrix element which is related to the calculation process. Basically this tool is used to view the link transformation matrix and the Jacobian matrix in the forward kinematic. And in the reverse kinematic, the joint variable column vector for the reverse kinematics.

5.3.5 Link Tool

The link tool is used to enter link properties and there are two sub-tools. They are link list tool and link properties tool. The link list is used to enter link parameters and link properties tool is used to visualize link data.

5.3.6 3D Graph Viewer

The 3D Graph Viewer is used for visualizing the entered parameters on the 3D environment. This graphical data can be rotating, zooming and panning to analyze the designed manipulator.

5.3.7 2D Graphic Output Viewer

This tool is used to display trajectory and dynamic graphical data in a time domain.

5.3.8 Manipulator Database Access Object

This tool is the same as the serialization tool. It is developed for the future enhancing of the this software tool. This database can be used to develop the controlling part of this software.

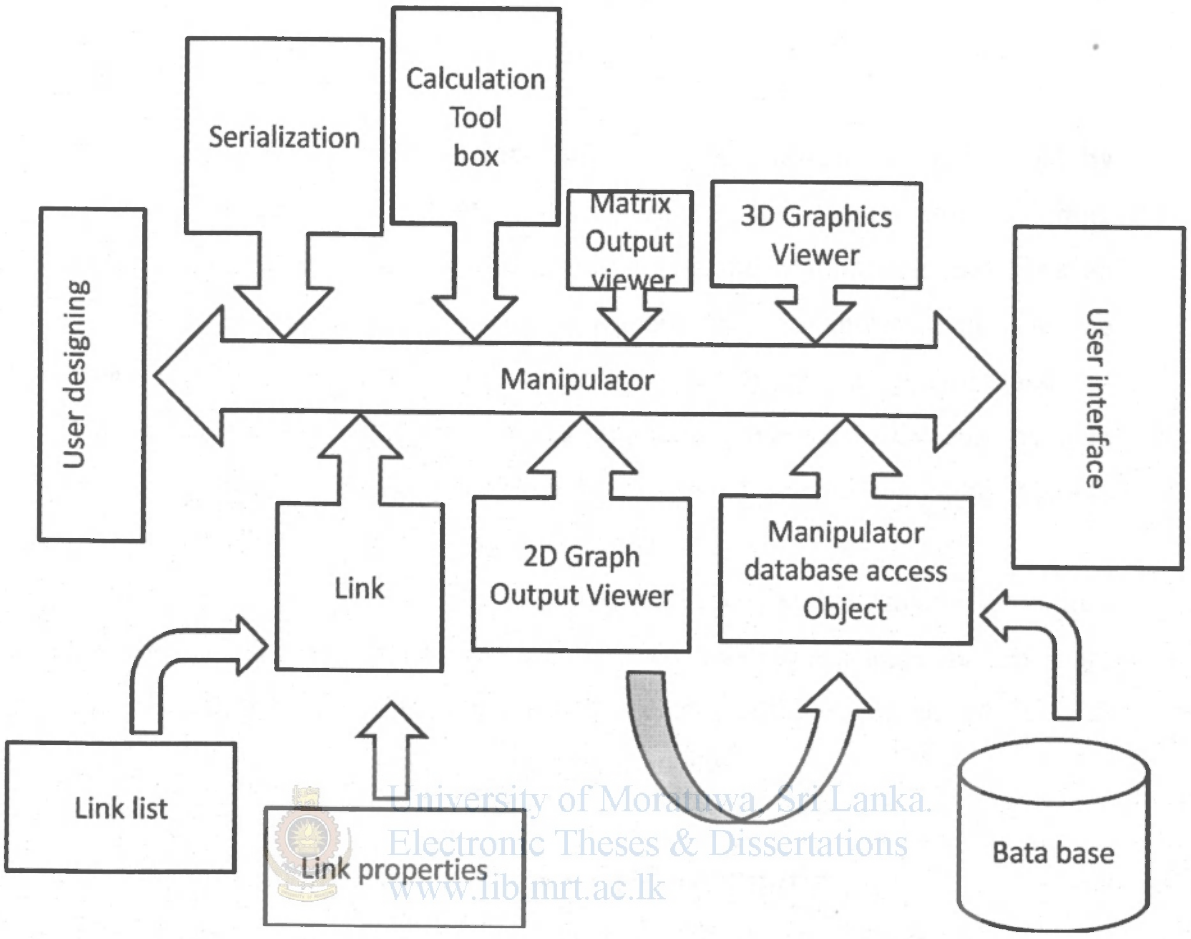


Fig. 5.2 -Software architecture

5.4 Functional relationships between the classes for simulator design

In an object oriented software design, the functional interaction is most important to eliminate the data duplication and modification of the functional operation as a requirement. Therefore the functional operation of this project is sharing in four basic classes. They are Manipulator, Newmat, NT Graph, NT Graph 3D. figure 5.3 The Manipulator class is handled for all kinds of robotic kinematic operations including kinematic calculation, updates, data storing and re-calling the application. Not only the kinematic but also the dynamic, trajectory planning and controlling operation are included too.

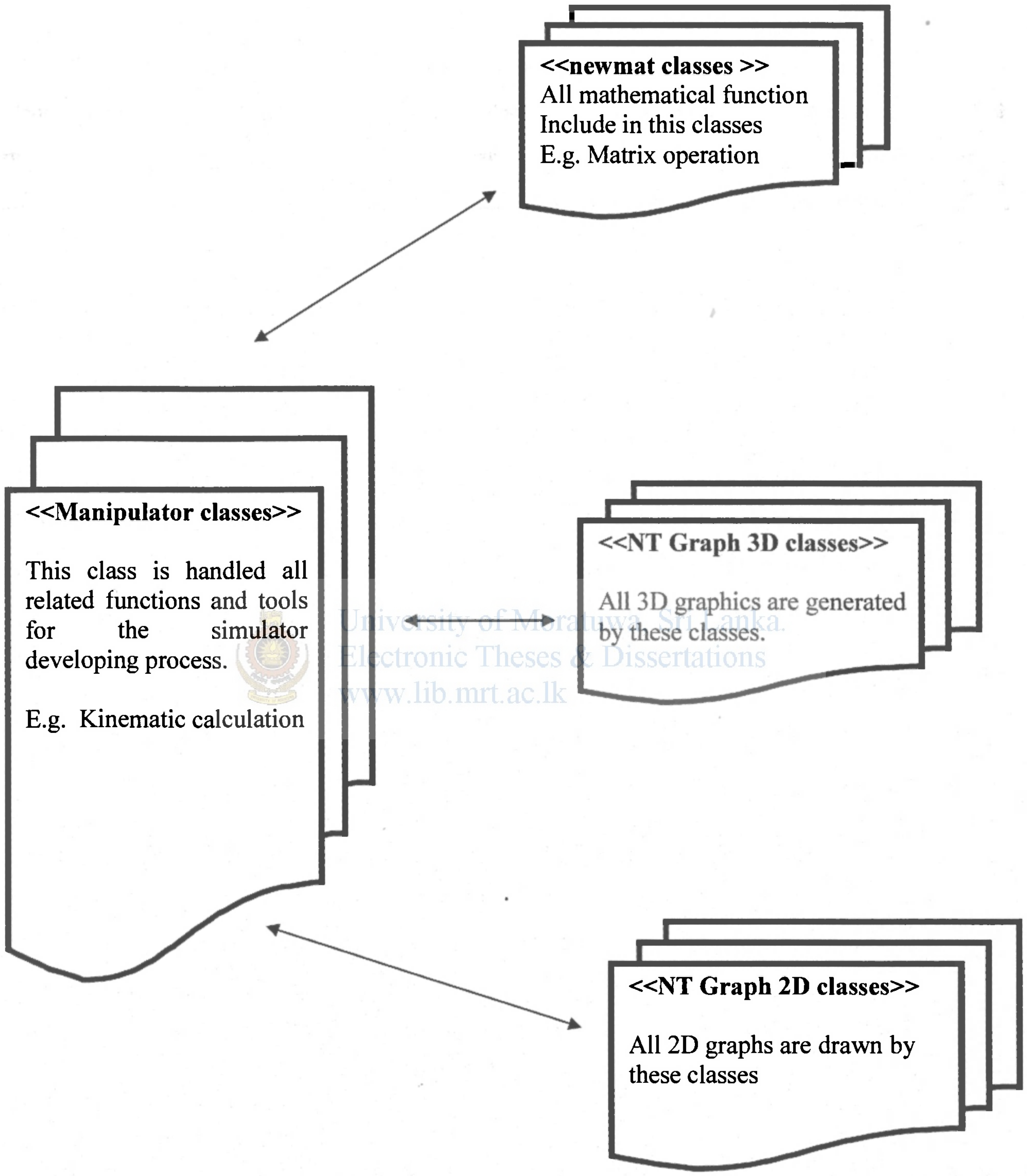
All kinds of mathematical operations including the matrix operation are handled by Newmat. If any mathematical operation is required in the Manipulator designing application the Newmat is called for this operation by the Manipulator class. As an example, to calculate the end effector transformation in the simulating process the Newmat class is called to the matrix multification operation by the Manipulator class.

The NT Graph function is called to draw the 2D graph to represent the 2 dimensional data by the NT Graph class. The joint angle, joint velocity, joint acceleration and the dynamic data in the time domain can be visualized on the graph.

The NT Graph 3D class is used to visualize the user defined manipulator link arrangement in the 3D environment. All the required tools to visualize like rotating, Zooming and colour changing are included in this class. Different colours are used to represent the links as well as to draw the trajectory segment.

5.4.1Functional relationship for the forward kinematics

The conceptual design for the kinematic development of this project is highly focused in this thesis and the sequence of C class which is related to the forward kinematic is shown in the above (Fig6.8) diagram. The entered link D-H parameters in the CNewlinkDlg class are stored in the Clink class as variables. The stored variables in the Clink class are used to calculate the link position and orientation in the CManipulator. The CManipulator is the base class of this project and all key features of the kinematic modeling are included in this class. They are *plot link, add new link, calculate_kine, Edit link properties* etc. The *plot link* function is used to transfer the data to the Cverttex class and also the *add new link* function is used for storing the new link data to find the position and orientation of the entire link. The calculate_kine is one of key function in the CManipulator class. This function handles all kinds of the forward kinematic algorithm and the important result of this operation is the position vector of the link. In the Cvertex class, the calculated 3D link position value is converted to the 2D value to display on the



<<newmat classes >>
All mathematical function Include in this classes
E.g. Matrix operation
<<Manipulator classes>>
This class is handled all related functions and tools for the simulator developing process.
E.g. Kinematic calculation
<<NT Graph 3D classes>>
All 3D graphics are generated by these classes.
<<NT Graph 2D classes>>
All 2D graphs are drawn by these classes

Fig.5.3. Functional relationship between the basic classes

computer screen. The calculated vertexes will draw on Cartesian in the 3DGraph class. The link joint variable can change in the link properties dialog. These changes are stored and update the Clink class. The updated values are repeated in the above sequence of forward kinematic and draw the current position on the user interface. This method is used to draw any number of links for any degree of freedom. The logic of the sequential designing is easy and flexible.

5.4.2 The matrix viewer

Matrix viewer is a different tool that can be view the matrix element in the kinematic operation and this is the subclass functional feature in the Manipulator class. Not only this operation but also through out this project this tool is used to test the data in the programming stage. This tool directly supports to check the validity of the used algorithm in kinematic modeling. And also it is used to check the error correction with the MATLAB robotic tool box results while in the programming.

5.4.3 Functional relationship for the inverse kinematics

The sequence of the reverse kinematic calculation is described as follows. Fig(6.9). The user can define his own trajectory and the end effector position and orientation can be found on the Cartesian space in the TP definition class. The stored end effector transformation matrix is used to calculate the joint space in the user define manipulator. This operation is done by CManipulator. The calculated joining variables will update the forward kinematic engine and it's draw on the Cartesian space.

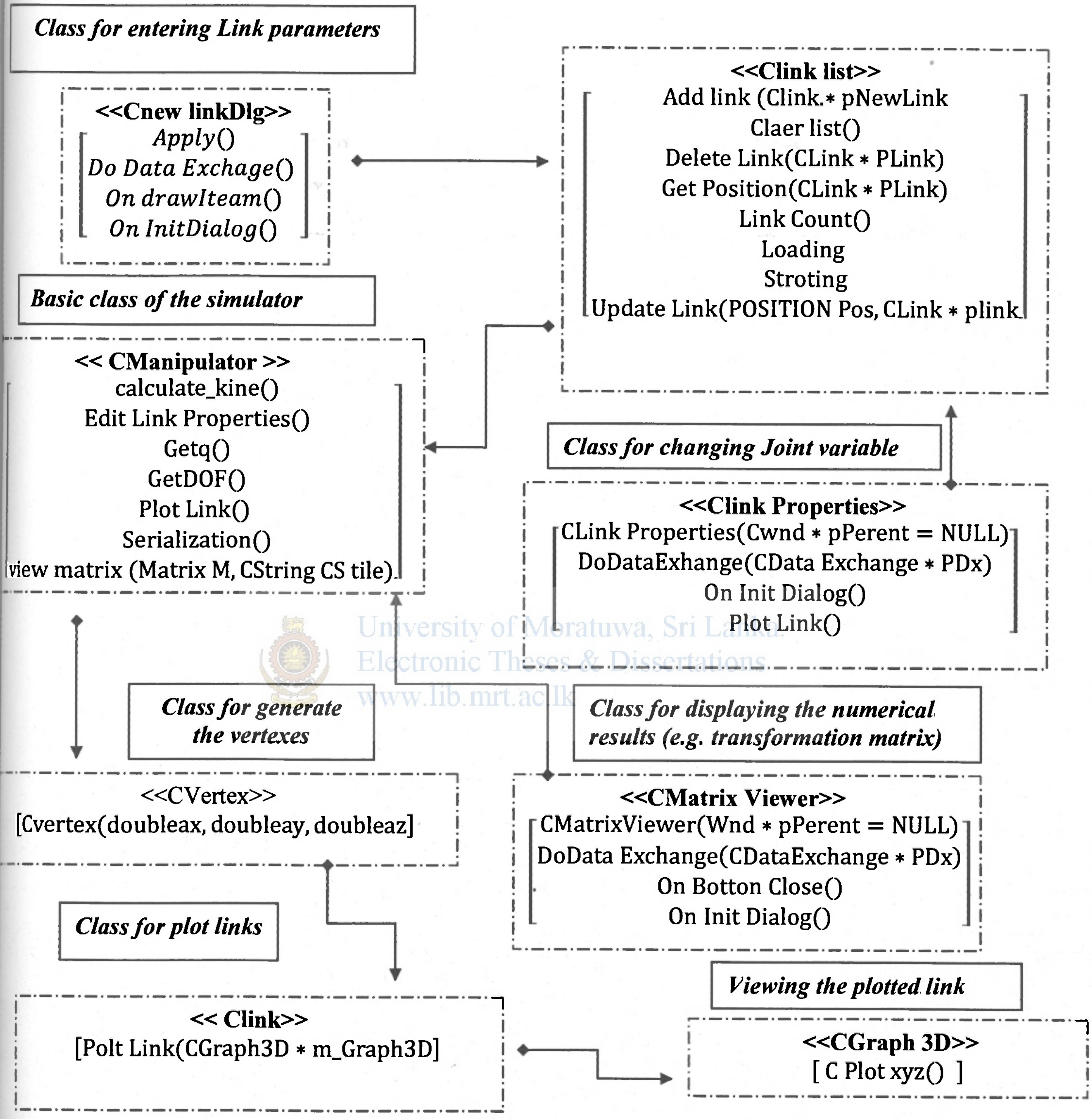

Class for entering Link parameters
<<Cnew linkDlg>>
Apply()
Do Data Exchage()
On drawIteam()
On InitDialog()
Basic class of the simulator
<< CManipulator >>
calculate_kine()
Edit Link Properties()
Getq()
GetDOF()
Plot Link()
Serialization()
view matrix (Matrix M, CString CS tile)
<<Clink list>>
Add link (Clink.* pNewLink
Claer list()
Delete Link(CLink * PLink)
Get Position(CLink * PLink)
Link Count()
Loading
Stroting
Update Link(POSITION Pos, CLink * plink
Class for changing Joint variable
<<Clink Properties>>
CLink Properties(Cwnd * pPerent = NULL)
DoDataExhange(CData Exchange * PDx)
On Init Dialog()
Plot Link()
Class for generate the vertexes
<<CVertex>>
[Cvertex(doubleax, doubleay, doubleaz]
Class for displaying the numerical results (e.g. transformation matrix)
<<CMatrix Viewer>>
CMatrixViewer(Wnd * pPerent = NULL)
DoData Exchange(CDataExchange * PDx)
On Botton Close()
On Init Dialog()
Class for plot links
<< Clink>>
[Polt Link(CGraph3D * m_Graph3D]
Viewing the plotted link
<<CGraph 3D>>
[C Plot xyz()]

Fig .5.4 Functional sequence of C class for forward kinematic

This thesis deeply discusses the kinematic behavior but the simulator provides the total solution (dynamic modeling, trajectory planning and manipulator control techniques for the manipulator users. But they are separate research components of this project. Therefore this thesis does not discuss another class and designing concept.

At the beginning, the members of this project have desired to develop separate classes (CKinermatic, CDyanmic, Ctrajectory etc.) which belong to their won research components. And further we desired to gather these classes to find the final solution of this project. Then some parts of the functions will be repeated and the program will get stuck while in the operation. Then the running efficiency of this project will reduce and increase the simulation error. Therefore the project members have desired to develop separate coding functions which are related to their won research components in the common classes to increase the running efficiency.

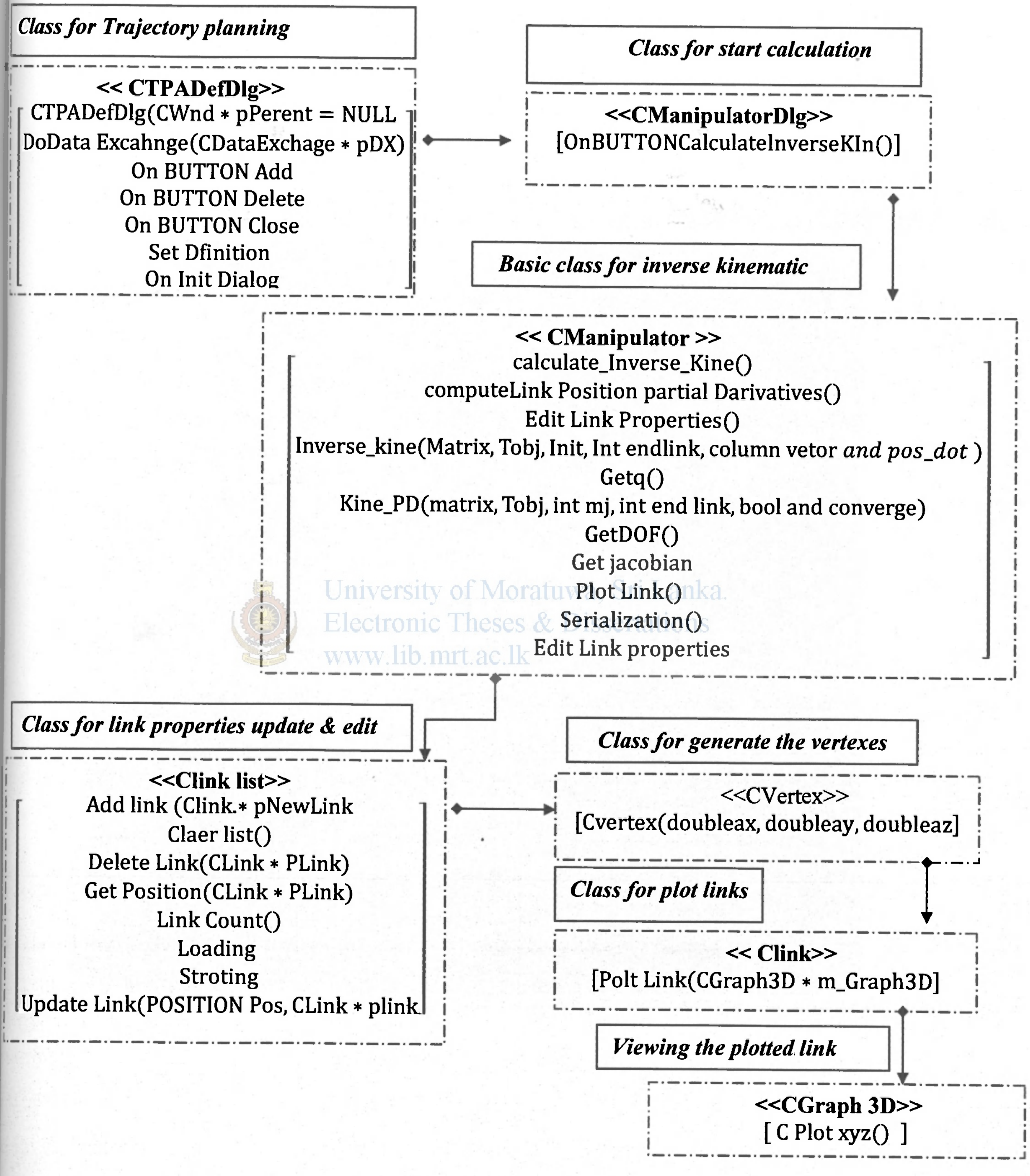

Class for Trajectory planning
<< CTPADefDlg>>
CTPADefDlg(CWnd * pPerent = NULL
DoData Excahnge(CDataExchage * pDX)
On BUTTON Add
On BUTTON Delete
On BUTTON Close
Set Dfinition
On Init Dialog
Class for start calculation
<<CManipulatorDlg>>
[OnBUTTONCalculateInverseKIn()]
Basic class for inverse kinematic
<< CManipulator >>
calculate_Inverse_Kine()
computeLink Position partial Darivatives()
Edit Link Properties()
Inverse_kine(Matrix, Tobj, Init, Int endlink, column vetor and pos_dot)
Getq()
Kine_PD(matrix, Tobj, int mj, int end link, bool and converge)
GetDOF()
Get jacobian
Plot Link()
Serialization()
Edit Link properties
Class for link properties update & edit
<<Clink list>>
Add link (Clink.* pNewLink
Claer list()
Delete Link(CLink * PLink)
Get Position(CLink * PLink)
Link Count()
Loading
Stroting
Update Link(POSITION Pos, CLink * plink
Class for generate the vertexes
<<CVertex>>
[Cvertex(doubleax, doubleay, doubleaz]
Class for plot links
<< Clink>>
[Polt Link(CGraph3D * m_Graph3D]
Viewing the plotted link
<<CGraph 3D>>
[C Plot xyz()]

Fig .5.5 funtional sequence of C classes in inverese kinematic

CHAPTER 6

Implementation of Programming Interface

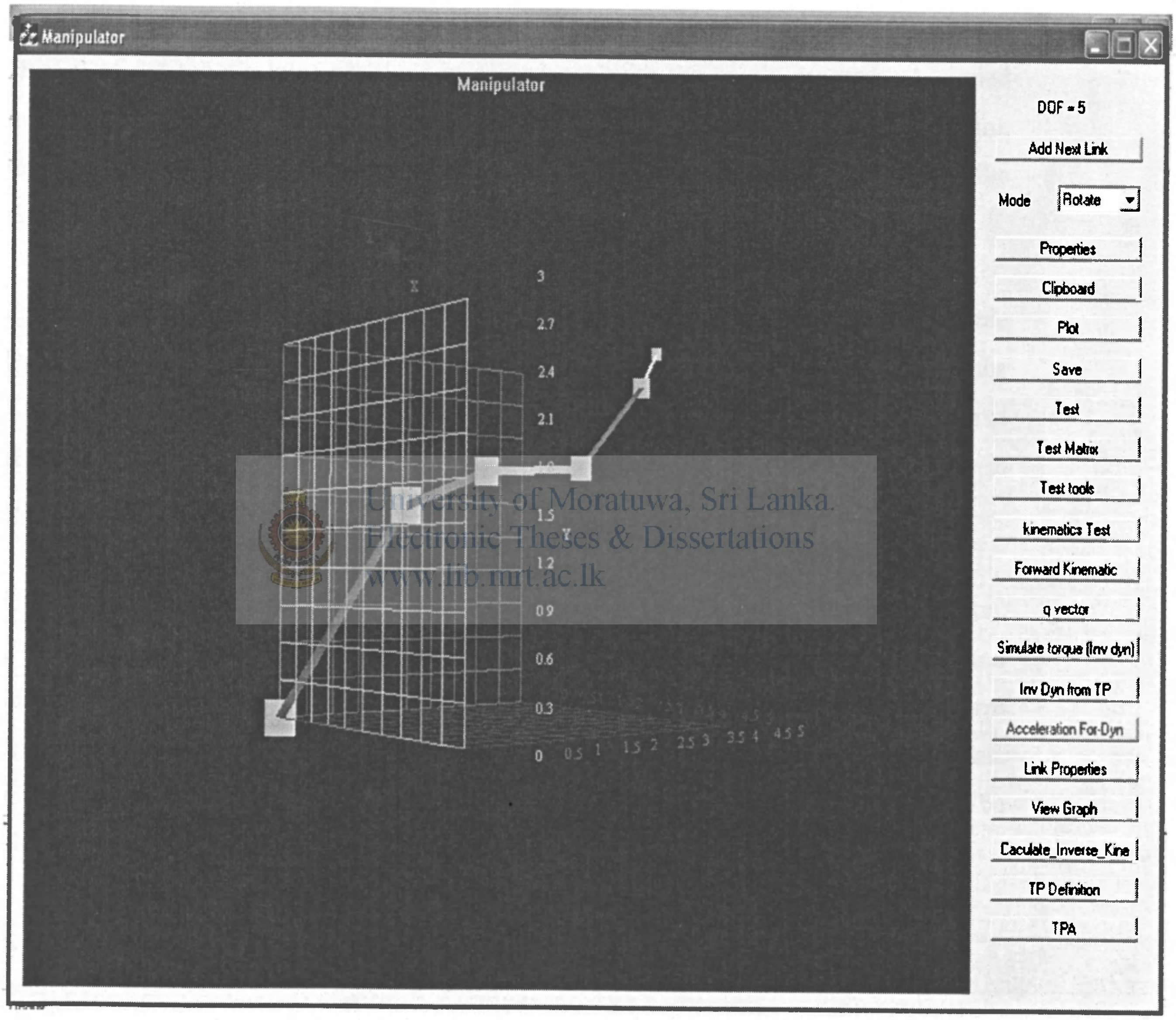


Fig .6.1-User Graphical interface (UGI)

6.1 Programming interface.

The completed application, shown in Figure 6.1 allows users to create and edit files containing commands to control the simulated robot, and run a simulation. Running a simulation updates both the 3-dimensional representation of the robot's motion, and the display of the robot's joint positions in accordance with the user's program. Assigned manipulator link frame on D-H parameter table (fig 6.3) can be entered on new link dialogs and it can be viewed as graphical on the 3D interface. Then the user can find the desired manipulator geometric view on the 3D environment.

Further this simulating tool provides facilities to user to the change the joining variable by using the link property dialog and the user can check the manipulator behavior in the Cartesian space. In the same instant the simulator updates both the 3-dimensional representation of the robot's motion and the display of the robot's joint positions in accordance with the user's designing.

User Graphical interface is used to display the user defined manipulator visualization. The right hand side buttons can be used to study the kinematic behaviors in forward and inverse bases, trajectory planning in the Cartesian and the joint space in a graphical form and dynamic modeling in a 2D graphical view. Further mode bottom can be used to rotate the 3D Viewer and zooming. The color of the X,Y,Z grid and the back ground can be changed by using the properties Button. (See fig 6.2)

6.2 New link entering dialog

The new link dialog can be used to enter the link parameters as D-H parameter mode. The group box of joint type can be used to select the joint type whether it is revaluate or prismatic. When the user selects the joint type by using the radio button the joint Variable group box will display the joint variable. Then the value of joint variable should be entered on that edit box. The link properties group box can be used to enter the D-H parameters like link offset, link length, link twist etc.

Not only the D-H parameter entering but also the new link dialog can change the color in the entire link and joint according the link thickness as required for the designing. Further more it is provided to enter joint mass, viscosity co-efficient of the link and the coulomb friction co-efficient of the link

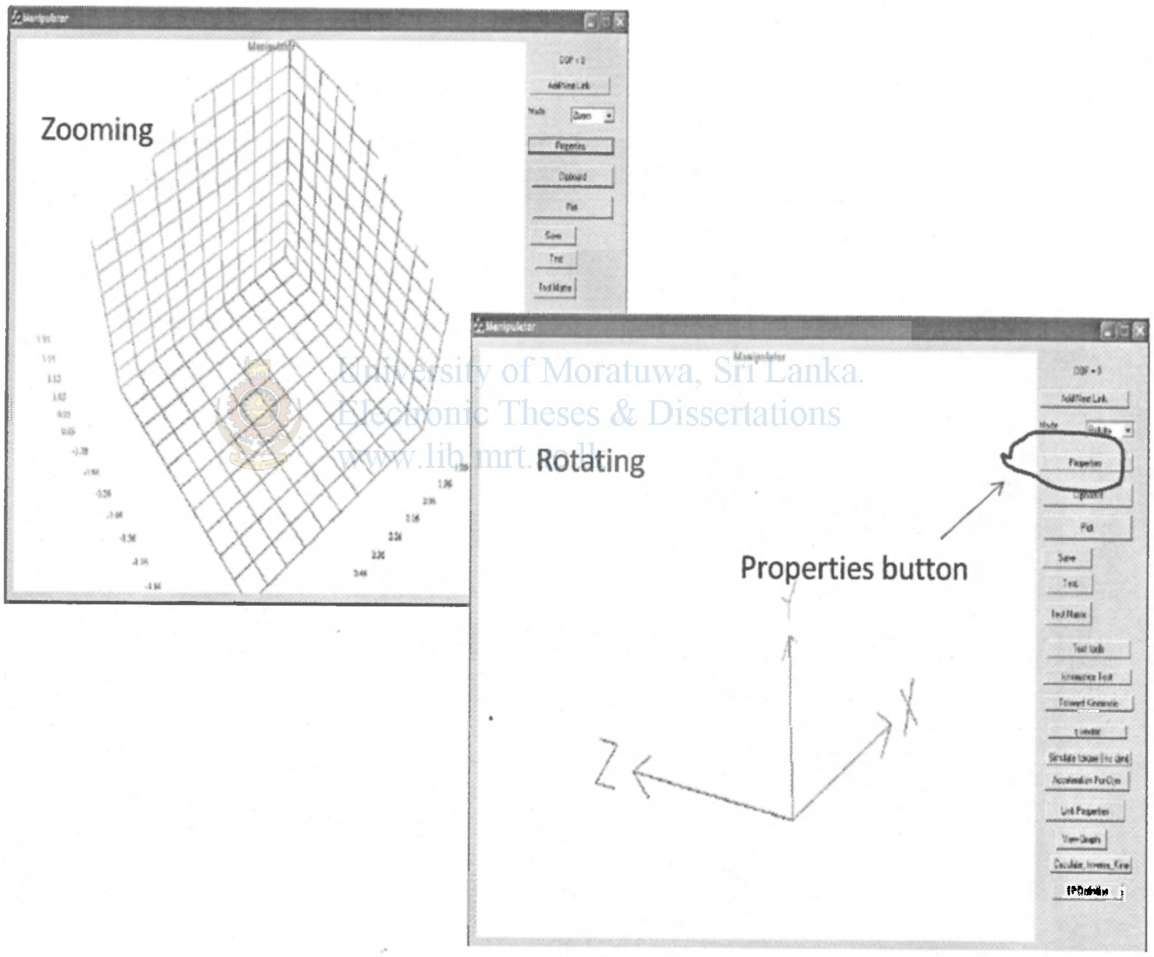
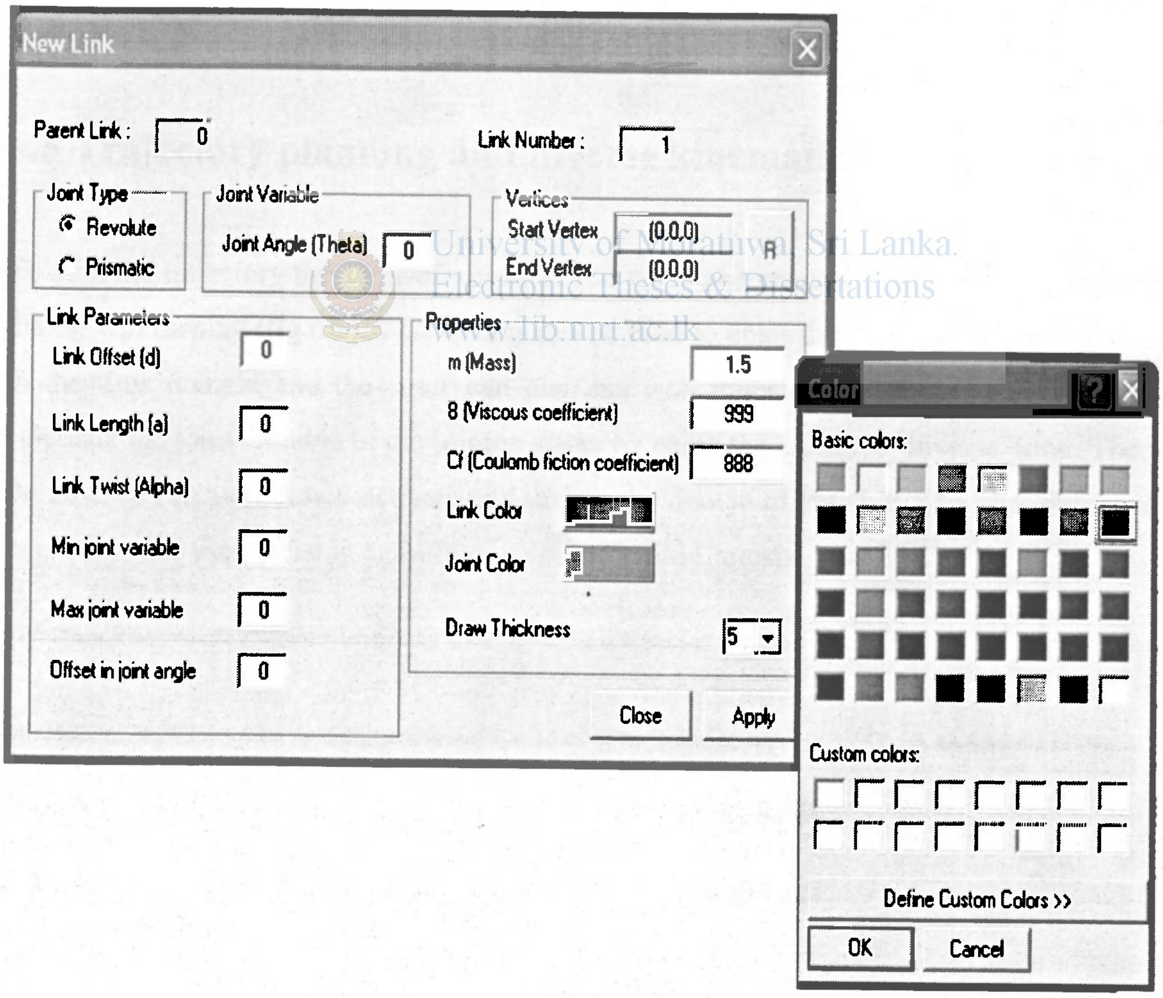


Fig. 6.2- Interface Properties

6.4. Link properties dialog

When the link properties button is pressed the above dialog will be displayed. The designed manipulator link joint variable need to change through this dialog and it will display the current orientation and the position matrixes and if the user can change the current joint variable and this new values will update the matrix values on this dialog box (fig 6.4). In the same instant, the manipulator new position and the orientation values will update on the graphical interface.


New Link

Parent Link : 0
Link Number : 1

Joint Type
[✓ Revolute]
Prismatic

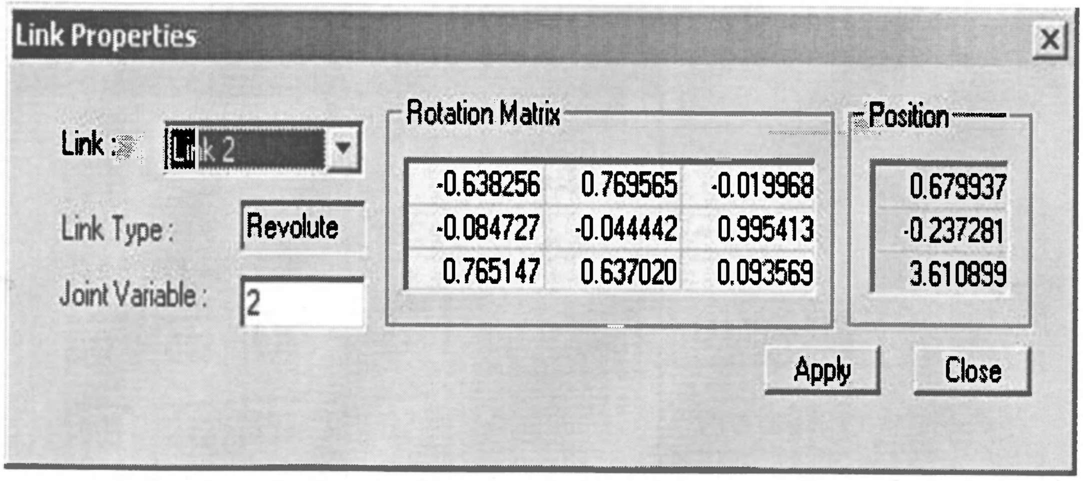
Joint Variable
Joint Angle (Theta) 0

Vertices
Start Vertex (0,0,0)
End Vertex (0,0,0)

Link Parameters
Link Offset (d) 0
Link Length (a) 0
Link Twist (Alpha) 0
Min joint variable 0
Max joint variable 0
Offset in joint angle 0

Properties
m (Mass) 1.5
8 (Viscous coefficient) 999
Cf (Coulomb fiction coefficient) 888
Link Color
Joint Color
Draw Thickness 5

Fig. 6.3- New Link Dialog



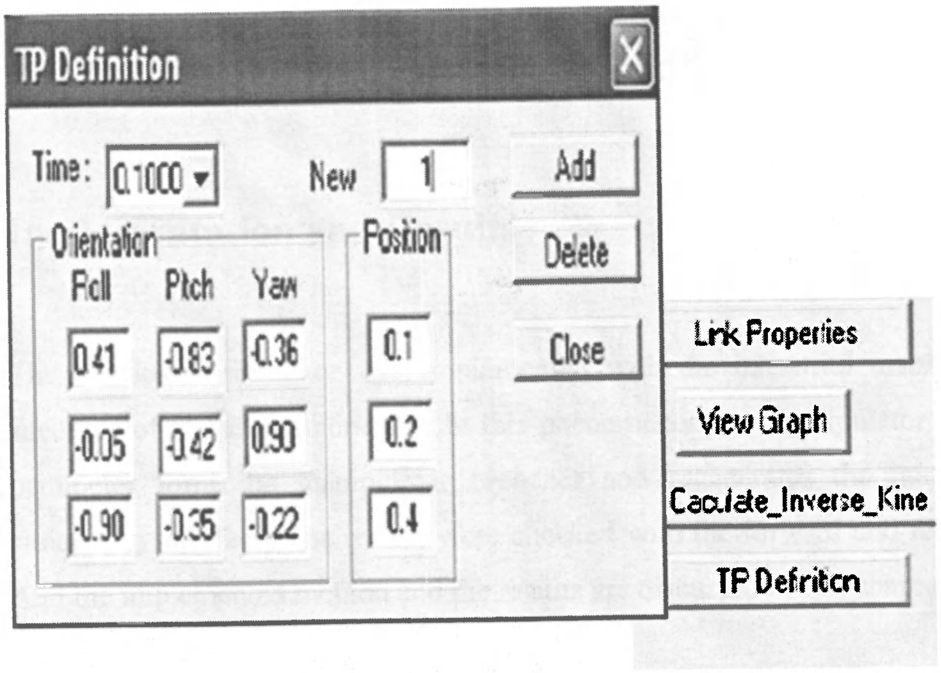
-0.638256	0.769565	-0.019968
-0.084727	-0.044442	0.995413
0.765147	0.637020	0.093569

0.679937
-0.237281
3.610899

Fig. 6.4- Link properties dialog

6.5 Trajectory planning and inverse kinematic calculation

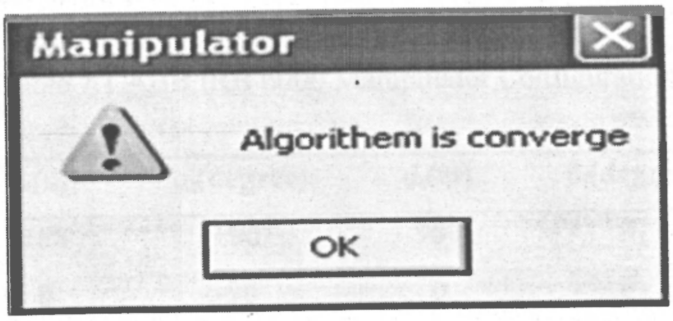
To plan the trajectory the TP_Definition button should be pressed and then the following dialog will display (fig 6.5). This dialog can be used to enter the position and orientation in the time domain and the user can plan his won trajectory. Then the tool box can calculate the joint variable in the joining space by using the calculate_Inverse_kine. Then the inverse calculation will activate and update the new joint variable. The converge tool is used to find the precision end-effectors point in the Cartesian space.



Roll	Pitch	Yaw	Position
0.41	-0.83	-0.36	0.1
-0.05	-0.42	0.90	0.2
-0.90	-0.35	-0.22	0.4

Fig. 6.5- Trajectory planning definition Dialog

And also the simulator will check the convergence of the calculation and display the message whether it is convergence or not (see fig.6.6). According to these results user graphical interface will update the new geometric view of the user designed manipulator.



Algorithem is converge

Fig .6.6- Algorithm convergence massage box

CHAPTER 7

Implementation and Results

The developed simulator was implemented with the industrial manipulator to check accuracy of the used algorithms. In this phenomenon, the manipulator took geometrical parameter form the manipulator broacher and regenerates the manipulator on the simulating interface. The results were checked with the forward and reverse algorithms. And the implemented method and the results are discussed in this chapter.

7.1 Testing the developed simulator on actual manipulator

Testing the software program on the actual environment is most important to understand the practical behavior of the actual simulator and accuracy of used algorithms. The ABB IRB 6000 robotic manipulator has been selected to test the developed algorithm of this project. It can be viewed on the developed simulator as shown in the figure 7.1. The ABB IRB 6000 is the world famous manipulator for many kinds of operation. It's configuration as follows [22].

Table 7.1 ABB IRB 6000 Manipulator Configuration

Link i	a_i(m)	α_i(degree)	d_i(m)	θ_i(degree)	q_i
1	188	$+90^0$	900	θ_1	θ_1
2	0	-90^0	0	θ_2	θ_2
3	1175	0	0	θ_3	θ_3
4	1300	0	0	θ_4	θ_4
5	0	-90^0	0	θ_5-90^0	θ_5
6	0	0	200	θ_6	θ_6

7.2 Testing process

In the forward kinematic process, the ABB IRB 6000 actual manipulator configuration was used to test the simulator accuracy. The method of comparison is, the selected joint variables and the arm configurations in the actual manipulator were regenerated on the developed simulator. After the actual manipulator end effector position and orientations checked with the simulated manipulator end effector position and orientation. For this process the BullEye is used to take the accurate reading with respect to base frame. The BullEye is a special tool which is used to measure the end effector position and orientation for the manipulator teaching process. The test results are shown in the table 7.2. In this testing process calibration tool orientation is disregarded. According to the above results the forward kinematic algorithms and the programming code are corrected and reliable.

Table 7.2-Froward kinematic results comparison

Joint variable						Simulated values			Actual value		
Θ_1	Θ_2	Θ_3	Θ_4	Θ_5	Θ_6	x	y	z	x	y	z
112°	-28°	-5°	105°	105°	-26°	1488.8	1145	2074.3	1488	1145	2075
0	-70°	-70°	40°	70°	37°	876	1145	2226	876.1	1145	2226.1
90°	36°	43°	0	55°	90°	660.85	491.47	849.92	661.5	492	850
135°	45°	-60°	47°	-60°	112°	204	487	208	204.3	487	208.1
-90°	0	43°	-59°	-155°	90°	349	200	900	350	200	900

In the inverse kinematic process, the manipulator end effector was moved to the different places in the Cartesian environment with respect to the base fame that the end effector position and orientation readings were taken form the manipulator controller for the known joint angles. After that this end effector positions and orientations were fed to the simulator and the joint variables readings were taken for the entire points. The actual readings and the simulated readings are shown in the table 7.3 and the end effector transformation matrixes for the entire points are shown in the table7.4. According to these results the actual readings are slightly deviated from the simulated readings in some point. The problem of this is that these points are located in the singularity. In this region simulated values are deviated from the actual readings. But no other algorithms are developed to solve the inverse kinematic as accurate results for the universal manipulator. In future if any body develops a more powerful algorithm it can be adapted to this software without any hindrance.

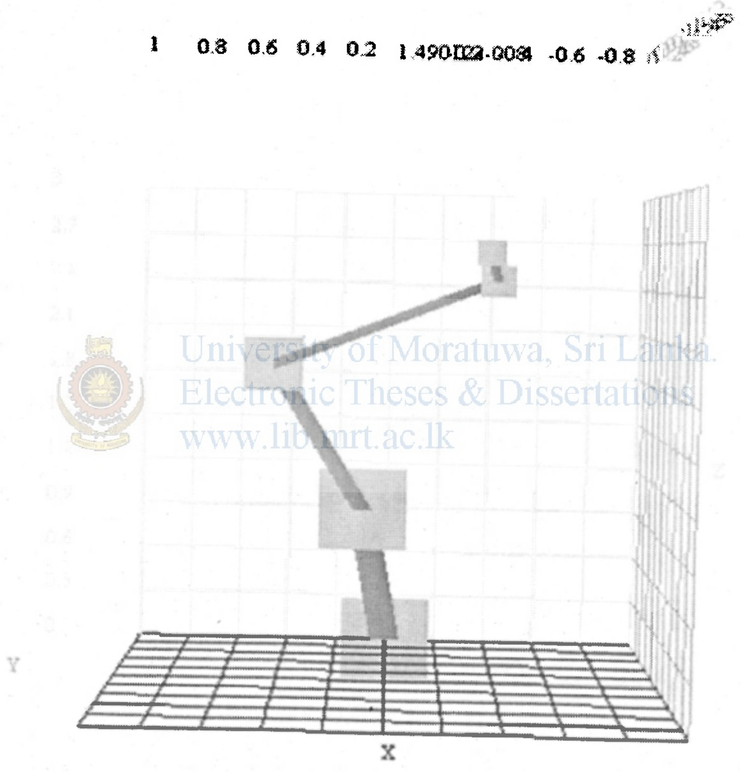


Fig. 7.1 ABB IRB 6000 Manipulator

Table 7.3 – Inverse kinematic results comparison

Reading	Actual valves						Simulated values					
	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6
1	73°	-123°	-78°	91°	103°	-63°	73°	-123°	-78°	91°	103°	-63°
2	0	-70°	-87°	37°	70°	37°	0	-70°	-86°	40°	70°	37°
3	65°	-57°	43°	0	145°	90°	65°	-57°	43°	12°	143°	90°
4	65°	-57°	43°	34°	121°	103°	70°	-57°	43°	37°	134°	104°

Table 7.4- End effector transformation matrix for the four different Cartesian points for inverse kinematic results comparison

Point 1				Point 2			
-0.6073	-0.6972	0.4031	1500.1	0.8776	-0.2531	0.4301	-215.4
0.2318	-0.6330	-0.7345	1145.3	0.4791	0.4731	-0.7381	-687.4
0.7652	-0.3501	0.5410	2315.7	0.0211	0.8413	0.5431	1356.3
0	0	0	1	0	0	0	1
Point 3				**Point 4**			
0.0102	-0.5462	0.8645	463.2	0.9867	-0.4563	-0.3421	-853.0
0.9976	0.0896	-0.0934	2651.8	-0.0123	-0.2375	0.5631	2775.8
0.7845	0.8453	-0.5432	-1334.0	-0.3452	0.3428	0.5432`1	875.2
0	0	0	1	0	0	0	1

7.3 Further development

The software architecture is designed to enhance the project without change in the basic functions. Therefore the expansion can be done without duplicating the data and the running efficiency can be maintained at the same level. That is the art of object oriented programming.

Although solving inverse kinematics constraint problems for the universal models is not a well developed research topic in computer graphics so far, the author believes it will attract the attention of more and more researchers in the new century. General high-performance solutions to this problem will have wide applicability in animation creation and simulation.

Solving the inverse kinematic is the real challenge of the project because no body has found a more accurate method for the universal manipulator. But the selected method of this simulator inverse kinematic is most suitable for the universal manipulator. If the inverse kinematic solution is the combination of different application oriented kinematic solving techniques then the simulator will give a more correct inverse kinematic answer. But there is a lot of work to do.

Next step of this project is to introduce, the developed simulator as a commercial product. But this simulator should be tested in a different practical environments. Therefore we have to plan to take a feed back from the simulator user through the internet. Hopefully, in the future this will be a good simulating tool for the manipulator users.

Conclusions

This thesis is designed to understand the kinematic behavior in the dynamic simulator for the universal manipulator. Different kinds of alternative kinematic techniques are used to develop the algorithms to find the joint space parameters and end-effector position and orientation in a graphical way. Among the various techniques, the D-H parameter analytical method is selected to view the developed manipulator on the graphical interface. This method is more efficient and accurate to solve the forward kinematics.

In the inverse kinematic, the Jacobian base Newton-Raphson techniques and the Taylor series expansion are used to find the joint space parameters from the end effector parameters. This combination is well adaptive to solve inverse kinematic for the universal manipulator. The final results are converged for 1000 times to error minimizing of this used method. If it increases higher than the 1000 there is no improvement in the results and it will increase the time.

The developed simulator was implemented on the industrial manipulator (ABB IRB 6000) to test the algorithms that are used for the kinematic modeling. According to the results, there is no deviation from actual values and the simulated values for the simulation. Therefore the used kinematic theories are correct for the simulating process.

Finally, the goal of this project has been achieved and this simulator may be a good tool for the redundant base and non- redundant base manipulator users and the designers.

References

[1] Nayar, H.D. Serial-link manipulator design software for modeling, visualization and performance analysis, Control, Automation, Robotics and Vision, 2002. ICARCV 2002. 7th International Conference on Volume 3, Issue , 2-5 Dec. 2002 Page(s): 1359 - 1364 vol.3

[2] R P Paul ,B Shimano, Kinematics control equations for simple manipulators. Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on Volume , Issue , 24-29 Apr 1988 Page(s):297 - 302 vol.1

[3] Eric M.Schwartz NON-COMMENSURATE MANIPULATOR JACOBIAN , Machine Intelligence Laboratory, University of Florida, IASTED International Conference ROBOTICS AND APPLICATIONS June 25-27, 2003, Salzburg, Austria

[4] K Mittal & I J Nagrath , Robotics and control text book ,1st edition ,Tata McGraw Hill publication, India,2004, chapter(s)-1,3,4,5

[5] J J. Craig, Introduction of robotic mechanics and control text book, second edition, PEARSON Education publication ,2005, Chapter(s)-3,4,5

[6] Peter I Corke , (http://www.cat.csiro.au/cmst/staff/pic/robot), Robotic tool box for MATLAB CSIRO Manufacturing Science and Technology Pinjarra Hills, AUSTRALIA.2001 Page(s) 2-10

[7] Deepak Tolani, Ambarish Goswami, and Norman I .Badler Real-Time Inverse Kinematics Techniques for Anthropomorphic Limbs Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on Volume , Issue , 24-29 Apr 1988 Page(s):669 - 674 vol.2

[8] Samuel R. Buss ,Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods. Department of Mathematics University of California, San Diego La Jolla, CA 92093-0112 sbuss@math.ucsd.edu April 17, 2004

[9] Hick perent ,Power point presentation for inverse kinematics

[10] Takahashi, T.; Kawamura, A.High speed numerical calculation method for on-line inverse kinematic of robot manipulators

[11] Robot Simulator on the Net –Essex University resource (eurobot@essex.ac.uk)

[12] Samuel R Buss, Jin-su kin, Selectively Damped Least Squares for Inverse Kinematics. Technical Report CS-2000-19 , Waterloo, Ontario, Canada, 2000

[13] Javier Roldan , Carl Crance, David Dooner ,Reverse kinematic analysis of the spatial six axis robotic manipulator with consecutive joint axes parallel.

[14] Why C++ is not just an Object Oriented Programming Language, AT&T Bell Laboratories.Murray Hill, New Jersey 07974

[15] R B Davies, Documentation for newmat11, a matrix library in C++Copyright (C) 2005:,*22 April, 2006.*

[16] Steve Rotenberg UCSD, Inverse Kinematic Analysis –power point presentation by CSE169: Computer Animation. Instructor: Winter 2005

[17] Ezio Malis, Lionel Morin ,Sylvie Boude. Two New Algorithms for Forward and Inverse Kinematics under Degenerate Conditions

[18] Newton-Raphson base inverse kinematic Solution. D The Shodor Education Foundation, Inc. in cooperation with the Department of Chemistry, The University of North Carolina at Chapel Hill

[19] Kang Teresa Ge Solving Inverse Kinematics Constraint Problems for Highly Articulated Models. the University of Waterloo

[20] Brad Howard ,Interface Design for Offline Robot Programming with the use of Virtual Simulation, Iowa State University 2112 Lincoln Way Ames IA, 50014 (319) 296-2112 bhoward@iastate.edu

[21] ABB IRB industrial application manual by ABB Industries

(www.ABB.com/Robotic)

[22] John E. Lloyd Vincent Hayward , A Discrete Algorithm for Fixed-path Trajectory Generation at Kinematic Singularities. Computer Science Dept., University of British Columbia Center for Intelligent Machines, McGill University Vancouver, B.C., Canada Montr´eal, P.Q., Canada
lloyd@cs.ubc.ca hayward@cim.mcgill.ca

[23] Removing the Singularities of Serial Manipulators by Transforming The Workspace John E. Lloyd Computer Science Dept., University of British Columbia Vancouver, B.C., Canada lloyd@cs.ubc.ca

[24] *Jinhyun Kim* ,Kinematic Singularity Avoidance for Autonomous Manipulation in Underwater Robotics & Bio-Mechatronics Lab., Pohang University of Science & Technology (POSTECH), Pohang, KOREA, 2003

[25] B. Sivaraman, T. Burks, and J. Schueller. "Using Modern Robot Synthesis and Analysis Tools for the Design of Agricultural Manipulators". Agricultural

Engineering International: the CIGR Ejournal. Invited Overview Paper No. 2. Vol. VIII. January, 2006.

[26] Nikolai Teofilov, 3D Graph ActiveX Control. nteofilov@yahoo.de

[27] B. Sivaraman, T. F. Burks1, J. K. Schueller ,Using Modern Robot Synthesis and Analysis Tools for the Design of Agricultural Manipulators University of Florida, Dept. of Agr. and Bio. Engineering

[28] Lorenzo Fluckiger*, Laurent Piguet**, Charles Baur*. Generic robotic kinematic generator for virtual environment interfaces Published in SPIE Telemanipulator and Telepresence Technologies III, Vol. 2901, pp. 186-195, Boston, Nov. 1996. NASA Ames Research Center - Intelligent Mechanisms Group Moffet Field, CA 94301 flueckiger@dmt.epfl.ch

[29] Peng Song, MODELING, ANALYSIS AND SIMULATION OF MULTIBODY SYSTEMS WITH CONTACT AND FRICTION, Dissertation in Mechanical Engineering and Applied Mechanics

[30] Herman Bruyninckx_*, Serial robots, 19 Aug 2005. (http://www.roble.info/)

Appendix A –Function of forward kinematics

```
void CManipulator::Calculate_kine()
//Calculate Direct kinematics at each Link
{
        int i = 1;
        ColumnVector PreviousEnd(3); PreviousEnd=0.0;
        Real arPreviousEnd[3];
        ColumnVector tmp_Pos(3); tmp_Pos=0.0;
        IdentityMatrix I(3);
        m_Pos=tmp_Pos;
        m_Rot=I;
        //ViewMatrix(m_Pos,"m_Pos");
        //ViewMatrix(m_Rot,"m_Rot");


        POSITION Pos = m_LinkList->GetHeadPosition();
                while( Pos != NULL )
                {
                        CLink* pLink = m_LinkList->GetNext( Pos ) ;
                        m_Pos = m_Pos + m_Rot*pLink->m_p;
                        m_Rot = m_Rot*pLink->m_R;

                        pLink->m_Start_Vertex.x=PreviousEnd(1);
                        pLink->m_Start_Vertex.y=PreviousEnd(2);
                        pLink->m_Start_Vertex.z=PreviousEnd(3);

                        pLink->m_End_Vertex.x=m_Pos(1);
                        pLink->m_End_Vertex.y=m_Pos(2);
                        pLink->m_End_Vertex.z=m_Pos(3);

                        pLink->m_pb=m_Pos;
                        pLink->m_Rb=m_Rot;

                        PreviousEnd=m_Pos; //Hold previous end posision for start of Next Link
                        //ViewMatrix(m_Pos,"m_Pos");
                        //ViewMatrix(m_Rot,"m_Rot");

                        i++;
                }
}

ReturnMatrix CManipulator::Torque_ZeroVelocity(ColumnVector qpp)
//Joint torque. when joint velocity is 0, based on Recursive Newton-Euler formulation.
{
        int i=1;
        ColumnVector ltorque(m_nDOF);
        Matrix Rt, temp;
        if(qpp.Nrows() != m_nDOF) AfxMessageBox("qpp is Invalied");
```

```
ColumnVector m_z0(3);
        m_z0(1) = 0.0; m_z0(2) = 0.0; m_z0(3) = 1.0;
        ColumnVector Previous_vp(3); Previous_vp=9.81;
        ColumnVector Previous_wp(3); Previous_wp=0.0;
        POSITION Pos = m_LinkList->GetHeadPosition();
                while( Pos != NULL )
                {
                        CLink* pLink = m_LinkList->GetNext( Pos ) ;
                        Rt = pLink->m_R.t();

                        if(pLink->m_njoint_type== 0)
                        {
                                pLink->m_wp = Rt*(Previous_wp + m_z0*qpp(i));
                                pLink->m_vp     =     CrossProduct(pLink->m_wp,pLink->m_p)     +
Rt*(Previous_vp);
                        }
                        else
                        {
                                pLink->m_wp = Rt*Previous_wp;
                                pLink->m_vp     =     Rt*(Previous_vp     +     m_z0*qpp(i))+
CrossProduct(pLink->m_wp,pLink->m_p);
                        }
                        pLink->m_acc = CrossProduct(pLink->m_wp,pLink->m_r) + pLink->m_vp;
                        Previous_vp=pLink->m_vp;
                        Previous_wp=pLink->m_wp;
                        i++;
                }
                Matrix PreviousLink_Rot; PreviousLink_Rot=0.0;
                i=m_nDOF;

                ColumnVector Next_f_nv;
                ColumnVector Next_n_nv;
                Pos = m_LinkList->GetTailPosition();
                while( Pos != NULL )
                {
                        CLink* pLink = m_LinkList->GetPrev( Pos ) ;
                        pLink->m_F = pLink->m_acc * pLink->m_m;
                        pLink->m_N = pLink->m_I*pLink->m_wp;
                        if(i == m_nDOF)
                        {
                                pLink->m_f_nv = pLink->m_F;
                                pLink->m_n_nv = CrossProduct(pLink->m_p,pLink->m_f_nv)
     + CrossProduct(pLink->m_r,pLink->m_F) + pLink->m_N;
                                PreviousLink_Rot=pLink->m_R; //Set Las Link m_R as Previous one
                                Next_f_nv=pLink->m_f_nv;//Set from last Link
                                Next_n_nv=pLink->m_n_nv;//Set from last Link
                        }
                        else
                        {
                                pLink->m_f_nv = PreviousLink_Rot*Next_f_nv + pLink->m_F;
                                pLink->m_n_nv     =     PreviousLink_Rot*Next_n_nv     +
CrossProduct(pLink->m_p,pLink->m_f_nv)
     + CrossProduct(pLink->m_r,pLink->m_F) + pLink->m_N;
                        }
PreviousLink_Rot=pLink->m_R;
                              if(pLink->m_njoint_type == 0)
                               temp = ((m_z0.t()*pLink->m_R)*pLink->m_n_nv);
                              else

temp = ((m_z0.t()*pLink->m_R)*pLink->m_f_nv);
                              ltorque(i) = temp(1,1);
                              Next_f_nv=pLink->m_f_nv;
                              Next_n_nv=pLink->m_n_nv;
                              i--;
                    }

  ltorque.Release(); return ltorque;

}

ReturnMatrix CManipulator::Inertia(ColumnVector q)
{
//AfxMessageBox("Inertia");
  Matrix M(m_nDOF,m_nDOF);
  ColumnVector torque(m_nDOF);
  Setq(q);
//IdentityMatrix I(m_nDOF);
//torque=I;

          for(int i = 1; i <= m_nDOF; i++)
          {
                    for(int j = 1; j <= m_nDOF; j++)
                    {
                              torque(j) = (i == j ? 1.0 : 0.0);
                    }
                    torque = Torque_ZeroVelocity(torque);
                    M.Column(i) = torque;
          }
          M.Release(); return M;
```

Appendix B –Function of Jacobian matrix

```
ReturnMatrix CManipulator::GetJacobian()
{
        int i, j=1;
        Matrix jac(6,m_nDOF);
        Matrix pr, temp(3,1);

        POSITION Pos = m_LinkList->GetHeadPosition();
        while( Pos != NULL )
        {
        CLink* pLink = m_LinkList->GetNext( Pos ) ;
                if(pLink->m_njoint_type == 0)  //if Revolute
                {
                        temp(1,1) = pLink->m_R(1,3);
                        temp(2,1) = pLink->m_R(2,3);
                        temp(3,1) = pLink->m_R(3,3);
                        //pr = p[dof]-p[i-1];
                        pr=m_Pos-pLink->m_pb; ///Check correct one is m_pb or m_p
                        temp = CrossProduct(temp,pr);
                        jac(1,j) = temp(1,1);
                        jac(2,j) = temp(2,1);
                        jac(3,j) = temp(3,1);
                        jac(4,j) = pLink->m_R(1,3);
                        jac(5,j) = pLink->m_R(2,3);
                        jac(6,j) = pLink->m_R(3,3);
                }
                else //Prismatic
                {
                        jac(1,j) = pLink->m_R(1,3);
                        jac(2,j) = pLink->m_R(2,3);
                        jac(3,j) = pLink->m_R(3,3);
                        jac(4,j) = jac(5,j) = jac(6,j) = 0.0;
                }
        j++;

        }

        Matrix zeros(3,3);
        zeros = (Real) 0.0;
        Matrix RT = m_Rot.t();
        Matrix Rot;
        Rot = ((RT & zeros) | (zeros & RT));
        jac = Rot*jac;

        jac.Release(); return jac;
}
```

Appendix C –Function of inverse kinematics

```
void CManipulator::Caculate_Inverse_Kine()
{
        Matrix TransformationMatrix(4,4); TransformationMatrix=0.0;
        _RecordsetPtr rsTrajectoryDef;
        CMDBA MDBA;
        MDBA.Connect();
        CString csSQL;
        csSQL="SELECT * FROM tbltrajectoryDef";
        rsTrajectoryDef=MDBA.GetRecordSet(csSQL);
        if(rsTrajectoryDef->GetRecordCount()>0)
        {
                CString csFVal = _T("");
                //int nFval=0;
    double dblFval=0;
                _variant_t vField(_T(""));
                _variant_t vResult;

                while(rsTrajectoryDef->A_EOF != VARIANT_TRUE)
                {

                        TransformationMatrix(1,1)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Roll_1")->Value);
                        TransformationMatrix(2,1)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Roll_2")->Value);
                        TransformationMatrix(3,1)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Roll_3")->Value);

                        TransformationMatrix(1,2)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Pitch_1")->Value);
                        TransformationMatrix(2,2)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Pitch_2")->Value);
                        TransformationMatrix(3,2)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Pitch_3")->Value);

                        TransformationMatrix(1,3)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Yaw_1")->Value);
                        TransformationMatrix(2,3)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Yaw_2")->Value);
                        TransformationMatrix(3,3)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("Rot_Yaw_3")->Value);

                        TransformationMatrix(1,4)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("CPos_x")->Value);
                        TransformationMatrix(2,4)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("CPos_y")->Value);
                        TransformationMatrix(3,4)=Getdouble_Value(rsTrajectoryDef->GetFields()-
>GetItem("CPos_z")->Value);
TransformationMatrix(4,1)=0;TransformationMatrix(4,2)=0;TransformationMatrix(4,3)=0;Transformation
Matrix(4,4)=1;

                              CString csMatrixName;
                              csMatrixName.Format("TransformationMatrixat%f
sec",Getdouble_Value(rsTrajectoryDef->GetFields()->GetItem("TimeI")->Value));
//ViewMatrix(TransformationMatrix,csMatrixName);
//////////////
                              ColumnVector tmp_q(m_nDOF);
                              bool converge;
                              tmp_q=Inverse_Kine(TransformationMatrix, 0,m_nDOF, converge);
//ViewMatrix(tmp_q,"tmp_q from Inverse_Kine");
                              if(converge)
                              {

                              }
                              else
                              {
                                        AfxMessageBox("Algorithem is not converge");
                              }
//////////////
                              PlotLinks();
                              rsTrajectoryDef->MoveNext();
                    }
          }
          else
                    AfxMessageBox("No Records in the Trajectory Definition");
}

ReturnMatrix CManipulator::Inverse_Kine(Matrix &Tobj, int mj, int endlink, bool &converge)
//Numerical inverse kinematics.

 //Tobj: Transformation matrix expressing the desired end effector pose.
 //mj: Select algorithm type, 0: based on Jacobian, 1: based on derivative of T.
 //converge: Indicate if the algorithm converge.
 //endlink: the link to pretend is the end effector
{
          IdentityMatrix I(4);
          ColumnVector qPrev, qout, dq, q_tmp;
          Matrix B, M;
          UpperTriangularMatrix U;

          qPrev = Getq();
          qout = qPrev;
          q_tmp = qout;

          converge = false;
          if(mj == 0) {  // Jacobian based

          Matrix Ipd(4,4), A, B(6,1),tmp(4,4);

          for(int j = 1; j <= NITMAX; j++)
          {



tmp=I;
                    Calculate_kine();
                    tmp.SubMatrix(1,3,1,3) = m_Rot;
                    tmp.SubMatrix(1,3,4,4) = m_Pos;
                    Ipd =tmp.i()*Tobj;
                    tmp.Release();
                    B(1,1) = Ipd(1,4);
                    B(2,1) = Ipd(2,4);
                    B(3,1) = Ipd(3,4);
                    B(4,1) = Ipd(3,2);
                    B(5,1) = Ipd(1,3);
                    B(6,1) = Ipd(2,1);
                    A=GetJacobian();
                    QRZ(A,U);
                    QRZ(A,B,M);
                    dq = U.i()*M;
                    while(dq.MaximumAbsoluteValue() > 1)
                              dq /= 10;

                    for(int k = 1; k<= dq.nrows(); k++)
                              qout(k)+=dq(k);
                    Setq(qout);

                    if (dq.MaximumAbsoluteValue() < ITOL)
                    {
                              //AfxMessageBox("Algorithem is converge");
                              converge = true;
                              break;
                    }
          }

          } else   // using partial derivative of T
          {
                    Matrix tmp(4,4);
                    Matrix A(12,m_nDOF);
                    ComputeLinkPositionPartialDerivative();
                    for(int j = 1; j <= NITMAX; j++)
                    {
                              tmp=I;
                              Calculate_kine();
                              tmp.SubMatrix(1,3,1,3) = m_Rot;
                              tmp.SubMatrix(1,3,4,4) = m_Pos;
                              B = (Tobj-tmp).SubMatrix(1,3,1,4).AsColumn();
                              int k=1;
                              POSITION Pos = m_LinkList->GetHeadPosition();
                              while( Pos != NULL )
{
                                        CLink* pLink = m_LinkList->GetNext( Pos );
                                        A.SubMatrix(1,12,k,k)                    =                    pLink-
>m_PositionPartialDerivative.SubMatrix(1,3,1,4).AsColumn();
                                        k++;
                              }

                              QRZ(A,U);
                              QRZ(A,B,M);
                              dq = U.i()*M;

                              while(dq.MaximumAbsoluteValue() > 1)
                              dq /= 10;

                              for(k = 1; k<=m_nDOF; k++)
                              qout(k)+=dq(k);
                              Setq(qout);
                              if (dq.MaximumAbsoluteValue() < ITOL)
                              {
                                        converge = true;
                                        break;
                              }
                    }
          }

          if(converge)
          {
                    int i = 1;
                    POSITION Pos = m_LinkList->GetHeadPosition();
                    while( Pos != NULL )
                    {
                              CLink* pLink = m_LinkList->GetNext( Pos ) ;
                              if(pLink->m_njoint_type == 0)  //if Revolute
                              {
                                        qout(i) = fmod(qout(i), 2*m_PI);
                              }
                              i++;
                    }
                    Setq(qPrev);
                    qout.Release();
                    return qout;
          }
          else
          {
                    Setq(qPrev);
                    q_tmp.Release();
                    return q_tmp;
          }
}

ReturnMatrix CManipulator::GetJacobian()
{
```

```
int i, j=1;
        Matrix jac(6,m_nDOF);
        Matrix pr, temp(3,1);

        POSITION Pos = m_LinkList->GetHeadPosition();
        while( Pos != NULL )
        {
        CLink* pLink = m_LinkList->GetNext( Pos ) ;
                if(pLink->m_njoint_type == 0)  //if Revolute
                {
                        temp(1,1) = pLink->m_R(1,3);
                        temp(2,1) = pLink->m_R(2,3);
                        temp(3,1) = pLink->m_R(3,3);
                        //pr = p[dof]-p[i-1];
                        pr=m_Pos-pLink->m_pb; ///Check correct one is m_pb or m_p
                        temp = CrossProduct(temp,pr);
                        jac(1,j) = temp(1,1);
                        jac(2,j) = temp(2,1);
                        jac(3,j) = temp(3,1);
                        jac(4,j) = pLink->m_R(1,3);
                        jac(5,j) = pLink->m_R(2,3);
                        jac(6,j) = pLink->m_R(3,3);
                }
                else //Prismatic
                {
                        jac(1,j) = pLink->m_R(1,3);
                        jac(2,j) = pLink->m_R(2,3);
                        jac(3,j) = pLink->m_R(3,3);
                        jac(4,j) = jac(5,j) = jac(6,j) = 0.0;
                }
        j++;

        }

        Matrix zeros(3,3);
        zeros = (Real) 0.0;
        Matrix RT = m_Rot.t();
        Matrix Rot;
        Rot = ((RT & zeros) | (zeros & RT));
        jac = Rot*jac;

        jac.Release(); return jac;
}
```