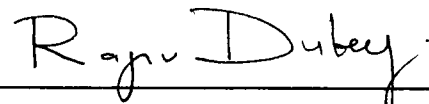


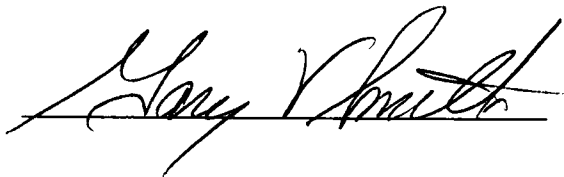
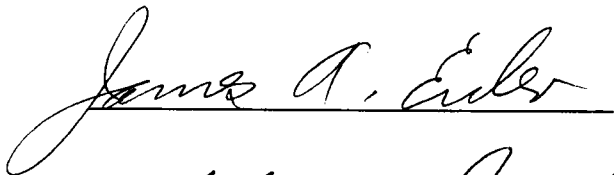
To the Graduate Council:

I am submitting herewith a thesis written by Robert Bidez Magness entitled "Robotic Control of the Seven-Degree-of-Freedom NASA Laboratory Telerobotic Manipulator." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Mechanical Engineering.



Rajiv V. Dubey, Major Professor

We have read this thesis and
recommend its acceptance:



Accepted for the Council:



Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master of Science degree at the University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotation from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of the Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of material in this thesis for financial gain shall not be allowed without my written permission

Signature Robert B. Magness

Date July 13, 1989

ROBOTIC CONTROL OF THE SEVEN-DEGREE-OF-FREEDOM
NASA LABORATORY TELEROBOTIC MANIPULATOR

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Robert Bidez Magness

August 1989

To my dear wife Catherine, and my wonderful children William, Heather, and Jackie, all of whom were the best reasons to continue this work, and all of whom inspired me with their patience and acceptance of a father not at home because "Daddy working".

ACKNOWLEDGMENTS

I would like to thank the members of my graduate committee, Professor Rajiv V. Dubey, Professor James A. Euler, and Professor Gary V. Smith, for their support during the course of this thesis work.

I would especially like to thank Professors Rajiv V. Dubey and James A. Euler for their suggestions and guidance, and their ready willingness to help me increase my understanding of manipulator control.

I would like to thank William R. Hamel, Wayne W. Manges, and Scott M. Babcock of the I&C Division of Oak Ridge National Laboratory for making computing, printing, and copying equipment available for my use at any time of any day or night. Their support greatly facilitated the completion of this work.

My parent's help during my graduate work at the University of Tennessee and during my undergraduate work at MIT whenever I needed it will always be remembered with love. I only hope that in the future when my children call upon me for my help, I will be able to give to them the same kindness, understanding, and support that my parents have given to me.

My undying admiration goes to my wife for her willingness and ability to handle her duties and mine in the home and with the children so that this work could be done.

Finally, I would like to thank NASA for the financial support needed to conduct the research and complete the work.

ABSTRACT

A computationally efficient real-time robotic control scheme for the seven-degree-of-freedom NASA Laboratory Telerobotic Manipulator (LTM) which utilizes the redundancy of the LTM to avoid joint limits, internal singularities and obstacles is developed. Real-time implementation is essential if the end-effector trajectory is continuously modified based on sensory feedback. Real-time resolution of the redundancy is accomplished in the framework of resolved motion rate control through an efficient implementation of the gradient projection method which does not require the generalized inverse of the Jacobian.

Trajectory planning capability in both Cartesian and joint space is developed. Starting, stopping, and transitions between two path segments are done in a smooth fashion by ensuring continuity in position, velocity, and acceleration. The complete end-effector trajectory does not have to be known, and only the current position and the next desired position are required, which allows immediate modification of the trajectory based on sensory feedback.

A user friendly interface was developed to control the LTM. The interface permits performance criteria and trajectory related information to be added or modified easily. A graphical representation capability showing three orthogonal views of the LTM was developed which permits investigation of the feasibility of control schemes before implementation. A complete set of instructions for the interface is provided.

The robotic control scheme has been shown to be both feasible and effective through real-time graphical simulations demonstrating obstacle avoidance, joint limit avoidance, and singularity avoidance. Integration of the control scheme into the overall system software for the LTM is currently in progress.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
1.1. Non-redundant Manipulators	
1.2. Redundant Manipulators	
1.3. Redundancy Control Problem	
1.4. User Interface	
1.5. Thesis Objectives	
1.6. Thesis Organization	
2. BACKGROUND.....	9
2.1. Resolved Motion Rate Control	
2.2. Singular Configurations	
2.3. Redundant Manipulators and Optimization	
2.3.1. The Null Space	
2.3.2. Seven-Degree-of-Freedom Manipulator	
2.4. Inverse Kinematics	
2.4.1. Generalized Inverse (Moore-Penrose pseudoinverse)	
2.4.2. The Gradient Projection Method	
3. AN EFFICIENT IMPLEMENTATION OF THE GRADIENT PROJECTION METHOD.....	28
3.1. Avoiding the Moore-Penrose Pseudoinverse Calculations	
3.2. Particular and Homogeneous Solutions	
3.3. Dubey-Euler-Babcock Method for Determining Least-Norm Solutions	
3.4. Relationship Between $A^{\oplus}y$ and $A^{+}y$	
3.5. Dubey-Euler-Babcock Approach for the Gradient Projection Method	
3.6. DEBA, Spherical Wrist, and Real Time Motion	
3.7. Null Space Projection Function	
3.7. Control Scheme Implementation for the LTM	
4. KINEMATICS OF THE LTM.....	43
4.1. Forward Kinematics of the LTM	

4.2. Inverse Kinematics of the LTM

4.2.1. The Midframe Jacobian

4.2.2. Least Euclidean Norm Solution

4.2.2.1. Selection of α

4.2.2.2. When α is the First Column

4.2.2.3. When α is the Third Column

4.2.2.4. When α is the Fourth Column

4.2.2.5. Wrist Singularity Case

4.2.3. Homogeneous Solution

4.2.3.1. Projection of Optimization Criterion Gradient onto the Null Space

4.2.3.2. Choosing the Scalar Constant k

5. COMPUTER PROGRAM.....59

5.1. Program Overview

5.2. Concise Description of Program Functions

5.2.1. Cartesian Space Functions

5.2.2. Joint Space Functions

5.3. Detailed Description of Program Functions

5.3.1. Cartesian Space Functions

5.3.1.1. GetDesiredPosition()

5.3.1.2. Accelerate()

5.3.1.3. AccSetup()

5.3.1.4. DotProduct ()

5.3.1.5. ForwardPosKinematics()

5.3.1.6. GetEulerAngles()

5.3.1.7. GetGradientH()

5.3.1.8. GetInitialAngles()

5.3.1.9. GetSize()

5.3.1.10. HandError()

5.3.1.11. InverseKinematics()

5.3.1.12. JointLimit()

5.3.1.13. KmaxKmin ()

5.3.1.14. LeastNorm()

5.3.1.15. LeastNormVelCheck()

5.3.1.16. MaintainConstantVelocity()

- 5.3.1.17. NextNode()
- 5.3.1.18. NullProjector()
- 5.3.1.19. NullSpaceOperator()
- 5.3.1.20. ProjectionElements()
- 5.3.1.21. ProjectVector()
- 5.3.1.22. SendCommand ()
- 5.3.1.23. SubtractVectors1()
- 5.3.1.24. TrigValues()
- 5.3.1.25. UpdateJointAngles()
- 5.3.1.26. VelocityLimit()
- 5.3.1.27. WristError()
- 5.3.2. Joint Space Functions
 - 5.3.2.1. GetJointConfig()
 - 5.3.2.2. ConstantJointVel()
 - 5.3.2.3. GetTimeSegment()
 - 5.3.2.4. JointAcc()
 - 5.3.2.5. JointAccSetup()
 - 5.3.2.6. JointVelCheck()
 - 5.3.2.7. NextJointNode()
 - 5.3.2.8. SubtractVectors2()

6. SIMULATIONS.....118

- 6.1. Introduction
- 6.2. Holding the End-Effector Stationary While
Improving a Performance Criterion
- 6.3. Avoiding an Obstacle
- 6.4. Avoiding a Joint Limit
- 6.5. Avoiding High Joint Velocities
- 6.6. Smooth Changes in Trajectory

7. USER INSTRUCTIONS.....130

- 7.1. Introduction
- 7.2. Robotics Menu
- 7.3. Cartesian Space Dialog

7.3.1. Alter a Path	
7.3.2. Choose a Path	
7.3.3. Create a Path	
7.3.4. Graphics Options	
7.3.5. Real Motion / Simulation	
7.3.6. Save Run	
7.3.7. Set Up Run Features	
7.4. Joint Space Dialog	
7.4.1. Alter a Path	
7.4.2. Choose Path	
7.4.3. Create a Path	
7.4.4. Graphics Options	
7.4.5. Real Motion / Simulation	
7.4.6. Save Run	
7.4.7. Set Up Run Features	
7.5. Help Menu	
7.5.1. Help with Demonstration	
7.5.2. Reference for Joint Angles	
7.5.3. Cartesian Space Help Dialog	
7.5.4. Joint Space Help Dialog	
7.6. Supplementary Information	
7.6.1. Obtaining Hard Copies	
7.6.2. Multiple Dialogs/Multiple Windows	
7.6.3. Hypothetical Run	
 8. CONCLUSIONS AND RECOMMENDATIONS.....	155
8.1. Conclusions	
8.2. Recommendations	
 LIST OF REFERENCES.....	157
 APPENDIXES.....	160
A. Inverse Kinematics for a Three-Degree-Of-Freedom Manipulator	
B. The A Matrices of the LTM	
C. The T Matrices of the LTM	

- D. The (wrist) Jacobian
- E. The Midframe (wrist) Jacobian
- F. User Interface Module (MacInterface.c)
- G. Macintosh Resource Files (MacInterface.r)
- H. Header Files
- I. Graphics Module (Graphics.c)
- J. Joint Space Module (LTM_JointSpace.c)
- K. Cartesian Space Module (LTM_CartSpace.c)
- L. Gradient Module (GetGradientH.c)

VITA	

	304

LIST OF FIGURES

Figure	Page
1-1. NASA Laboratory Telerobotic Manipulator (LTM).....	2
1-2. Four types of coordinate systems used in positioning an end-effector.....	4
1-3. Cincinnati Milacron T ³ and PUMA 560 series robot arms.....	5
1-4. Seven-degree-of-freedom CESAR Arm.....	6
2-1. A 3-degree-of-freedom manipulator.....	10
2-2. Singular configuration of a 3-degree-of-freedom manipulator (I).....	12
2-3. Singular configuration of a 3-degree-of-freedom manipulator (II).....	13
2-4. Singular configuration of a 3-degree-of-freedom manipulator (III).....	14
2-5. Planar manipulator utilizing the null space of manipulator Jacobian.....	17
2-6. Lines CD and EF drawn on Cartesian coordinate system.....	21
2-7. Matrix representation of lines CD and EF.....	22
2-8. Gradient projection method for redundant manipulator with one extra degree of freedom.....	27
3-1. Particular solution projected onto null space and row space of matrix A.....	35
3-2. Real-time implementation block diagram.....	42
4-1 Reference configuration for kinematics of the LTM.....	44
4-2 Flowchart showing how α is chosen.....	51
5-1 Global variable buffer specifies a standard set of variables.....	60
5-2 Function tree of the robotic control software for the LTM.....	61
5-3. GetDesiredPosition() function tree for motion in Cartesian space.....	62
5-4. GetJointConfig() function tree for motion in joint space.....	63
5-5. Starting and stopping motion.....	68
5-6. Straight line motion.....	68
5-7. Trajectory straight line segment transitions.....	70
5-8. Flowchart for GetDesiredPosition().....	71

5-9. Starting and stopping.....	74
5-10. Flow of variables in GetDesiredPosition() after switch statement	76
5-11. Euler angles defined and interpreted in base coordinates.....	85
5-12. Cross product representation of infinitesimal rotation vector.....	89
6-1. Reference configuration of LTM for simulations.....	119
6-2. Anthropomorphic stance defining top, front, and side views for simulations....	119
6-3. Improving a performance criterion while holding end-effector stationary.....	120
6-4. End-effector motion along a straight line with a fixed orientation (Least-norm solution -- arm hits object).....	122
6-5. End-effector motion along a straight line with a fixed orientation (Redundancy utilized to avoid object).....	123
6-6. End-effector motion along a straight line with a fixed orientation (Least-norm solution -- reaches joint 5 limit).....	124
6-7. End-effector motion along a straight line with a fixed orientation (Redundancy utilized to avoid joint limits).....	125
6-8. End-effector motion along Cartesian path with a fixed orientation (Least-norm solution -- joint velocity too high).....	127
6-9. End-effector motion along Cartesian path with a fixed orientation (Redundancy utilized to avoid singularity).....	128
6-10. End-effector motion with smooth changes in trajectory.....	129
7-1. The <i>Robotics</i> menu.....	131
7-2. <i>Cartesian Space</i> dialog.....	132
7-3. <i>Alter A Path</i> dialog.....	133
7-4. <i>Choose Path</i> dialog.....	133
7-5. <i>Create A Path</i> dialog (Cartesian space).....	134
7-6. <i>Graphics Options</i> dialog.....	136
7-7. <i>Show Shoulder</i> option.....	137
7-8. All three <i>Show</i> options.....	137
7-9. <i>Real Motion / Simulation</i> controls.....	138

7-10. <i>Save Run</i> dialog.....	138
7-11. <i>Set Up Run Features</i> dialog (Cartesian space).....	139
7-12. <i>Joint Space</i> dialog.....	143
7-13. <i>Create A Path</i> dialog for joint space.....	144
7-14. <i>Set Up Run Features</i> dialog (joint space).....	145
7-15. The <i>Help</i> menu.....	146
7-16. Reference configuration of the LTM for specifying joint angles.....	147
7-17. <i>Help</i> dialog for Cartesian space.....	148
7-18. <i>Help</i> dialog for joint space.....	149
7-19. Multiple dialogs and windows.....	152
7-20. The project icon.....	152
A-1. A 3-degree-of-freedom manipulator.....	162

CHAPTER 1

INTRODUCTION

NASA has embarked on an extensive national project to establish a permanent human-occupied space station. To accomplish this project, significantly increased levels of dexterous human-like handling tasks will be required in orbit. This will include space station construction as well as planned and unplanned maintenance operations on the space station. In addition, a significant amount of satellite repair and maintenance is expected in the future. To meet the needs for sharply increased levels of dexterous handling while decreasing the levels of required human extravehicular activity, NASA has established a goal for significant use of telerobotic hardware in future space activities.

The NASA Langley Research Center has sponsored the Laboratory Telerobotic Manipulator (LTM) Prototype Project at the Oak Ridge National Laboratory to develop prototypical manipulators for use in NASA laboratories to develop and demonstrate telerobotic and robotic capabilities in an earth-based environment [1]. As a result, the LTM must be designed to be a high-quality force-reflecting teleoperator with capabilities for robotic operation. High performance under human control, low required backdriving torque, high velocity and acceleration capability, and good capacity-to-weight ratio are emphasized in the design. To provide the basis for a transition to autonomous robotic operation, features for high-quality robotic operation are also provided. They include good end-effector positioning accuracy and high mechanical and control stiffness. The LTM is also designed for modular maintainability to facilitate repair and reconfiguration.

The LTM arm, shown in Fig. 1-1, has seven degrees of freedom that provide kinematic redundancy. The arm is configured from three common pitch/yaw joints which combine to provide shoulder, elbow, and wrist joints. The interface boundaries of these joints provide inherent modularity. A wrist roll mechanism, mounted on the output of the wrist joint, provides the seventh degree of freedom. Seven degrees of freedom allow the LTM to reorient itself without changing the end-effector position and orientation. This research was aimed at developing a robotic control scheme for the LTM which provides the capability to utilize the redundancy of the LTM to improve its performance.

1.1. Non-redundant Manipulators

Non-redundant manipulators have the minimum number of joints or degrees of freedom to perform a given task. For general purpose non-redundant manipulators, six

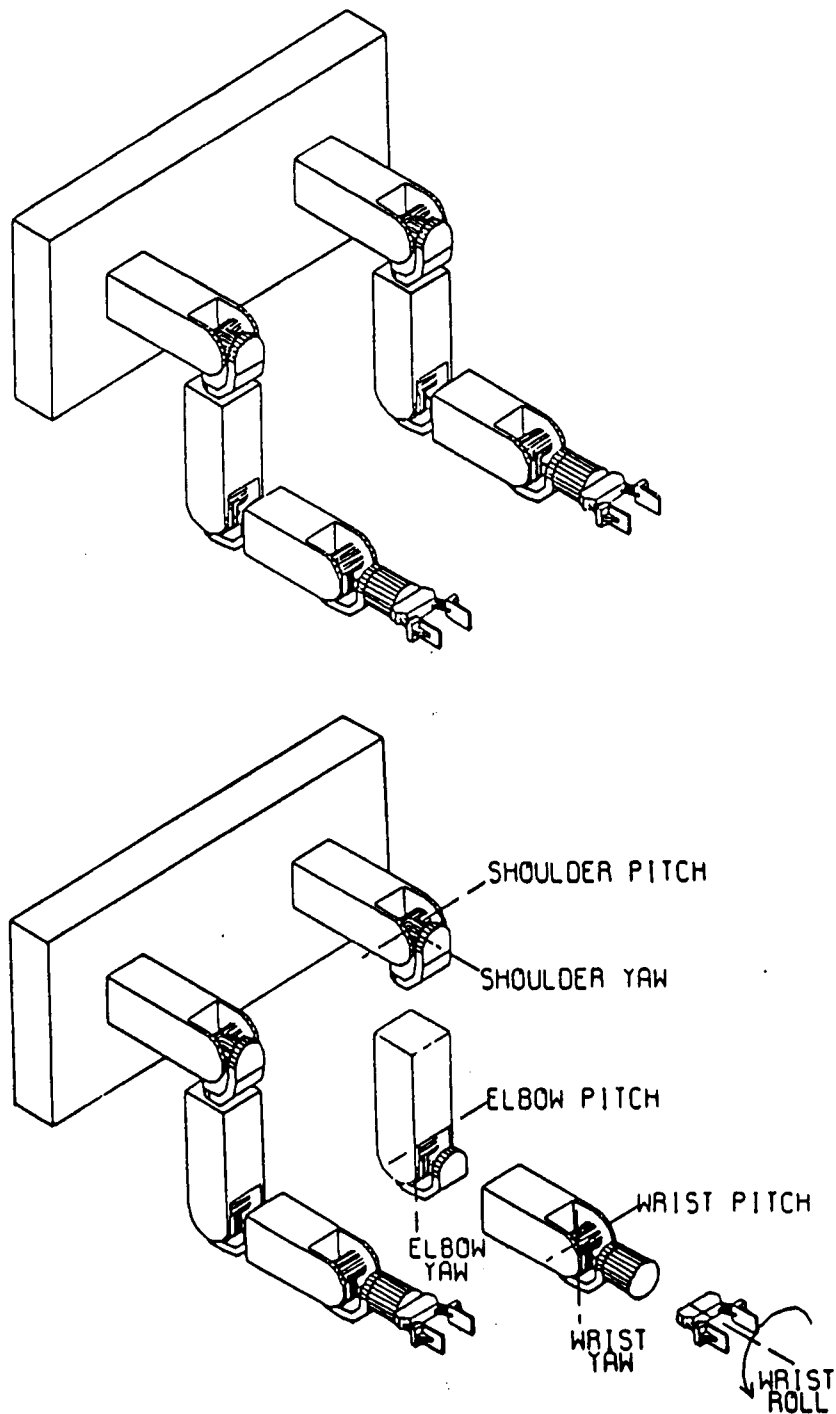


Figure 1-1: NASA Laboratory Telerobotic Manipulator (LTM) (a) anthropomorphic stance (b) joint axis and modularity (Courtesy I&C Division ORNL)

degrees of freedom are required, three for position and three for orientation. Generally the arm subassembly positions the end-effector and the wrist subassembly orients it. Figure 1-2 shows the four types of coordinate systems used in positioning an end-effector: Cartesian coordinates (Sigma robot from Olivetti and IBM's RS-1), cylindrical coordinates (Versatran 600 from Prab), spherical coordinates (Unimate 2000B from Unimation Inc.), and revolute coordinates (PUMA from Unimation Inc., and Cincinnati Milacron T³ robot). Spherical wrists, where the axes of rotation of the last three coordinate systems intersect at a point, are common. The Cincinnati Milacron T³ robot and PUMA 560 series (Fig. 1-3) show what are often called the pitch, yaw, roll joints of spherical wrists.

1.2. Redundant Manipulators

Redundant manipulators differ from non-redundant manipulators in the number of degrees of freedom used for a particular task. A manipulator can be termed redundant for a particular task if the number of degrees of freedom needed to perform the task is less than the number of degrees of freedom of the manipulator--extra degrees of freedom can be used to improve the performance of the manipulator. For an end-effector to move with arbitrary linear and angular velocities while at the same time optimizing the configuration of the arm, manipulators must have seven or more degrees of freedom.

Redundant manipulators are currently objects of research. Two seven-degree-of-freedom redundant manipulators presently being used for research in the United States are the Laboratory Telerobotic Manipulator (Figure 1-1) and the CESAR manipulator (Figure 1-4). Both manipulators were developed and built at Oak Ridge National Laboratory (ORNL). Research into the control of redundant manipulators is underway at ORNL using the CESAR manipulator, while the LTM is being used at NASA Langley to investigate robot manipulators in relation to construction of the U.S. space station. A commercial seven-degree-of-freedom manipulator is built by the Robotics Research Corporation.

1.3. Redundancy Control Problem

For a six-degree-of-freedom manipulator there is a unique set of joint velocities for a specified linear and angular velocity of the end-effector. For a seven-or-more-degree-of-freedom manipulator, however, there is an infinite number of joint velocity solutions to cause the same end-effector motion. Extra degrees of freedom can be used not only to avoid obstacles, joint limits, and joint velocity limits; but also to improve the performance

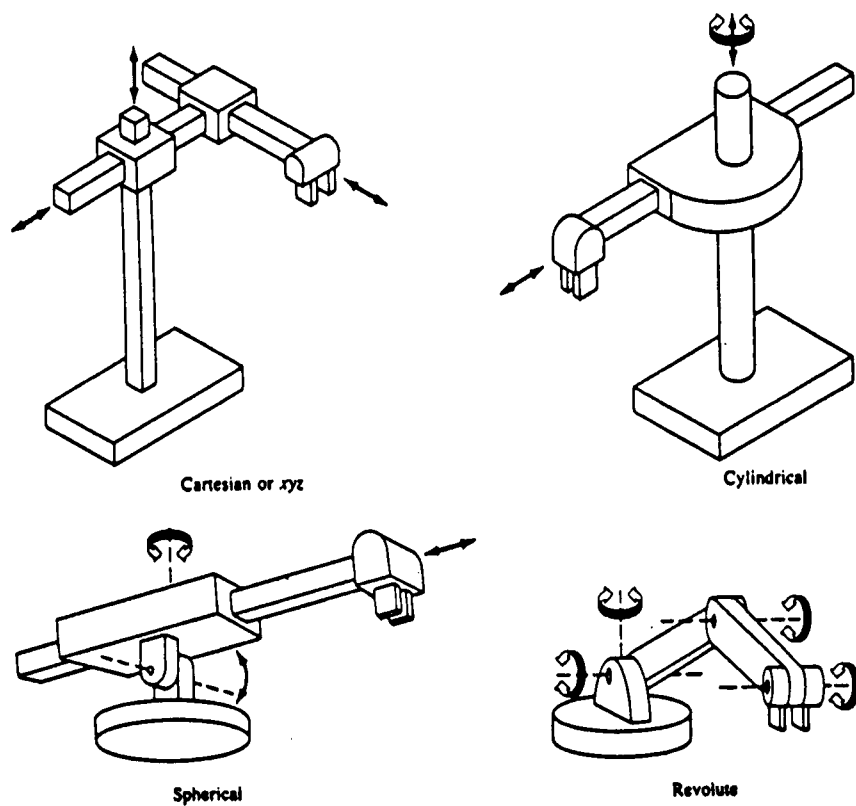


Figure 1-2: Four types of coordinate systems used in positioning an end-effector.
 (K.S.Fu,R.C.Gonzalez,C.S.G.Lee, "Robotics: Control,Sensing, Vision,
 and Intelligence", McGraw-Hill Book Company, 1987, p. 3)

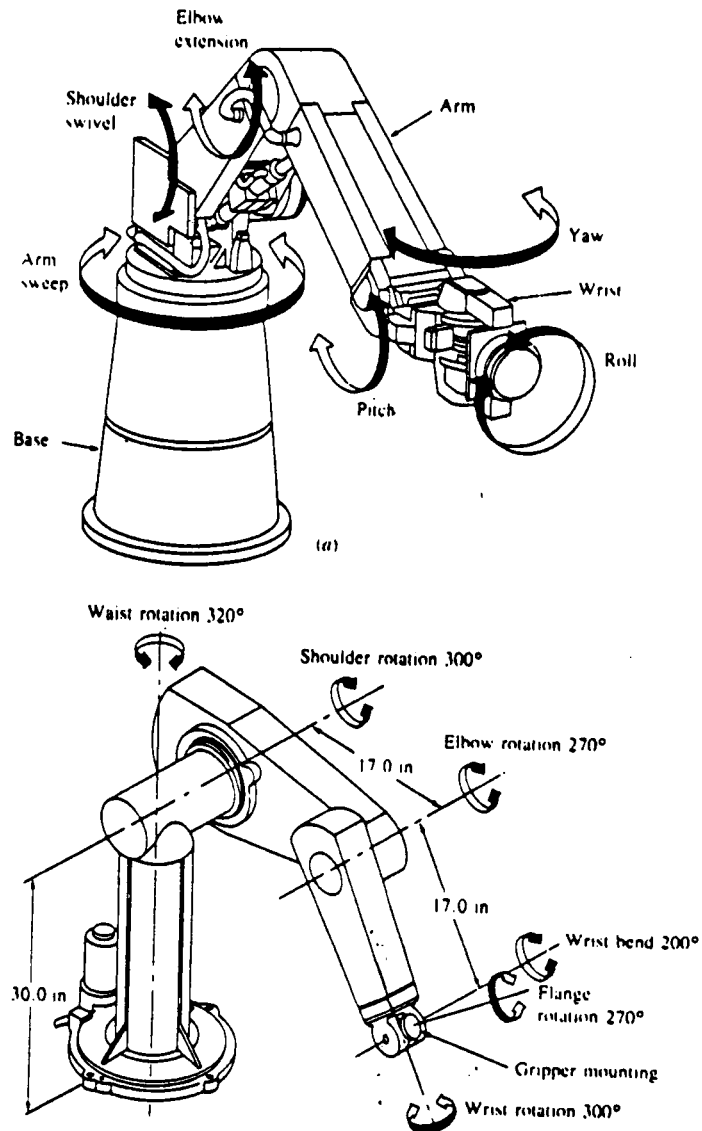


Figure 1-3 : Cincinnati Milacron T³ (a) and PUMA series (b) robot arms.
 (K.S. Fu, R.C. Gonzalez, C.S.G. Lee, "Robotics: Control, Sensing, Vision, and Intelligence", McGraw-Hill Book Company, 1987, p.2)

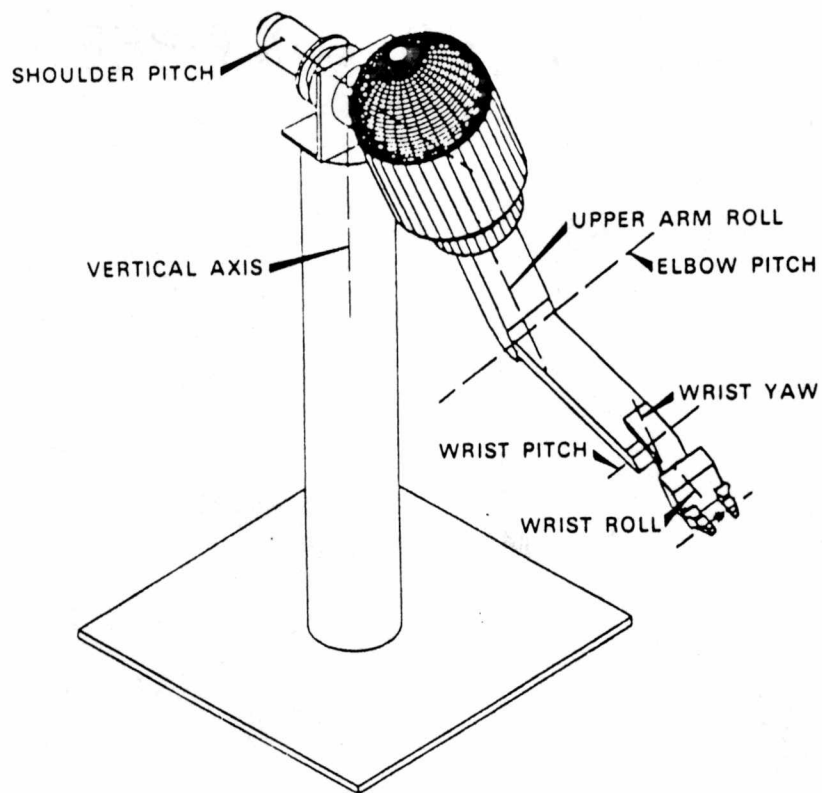


Figure 1-4: Seven-degree-of-freedom CESAR Arm (Courtesy I&C Division ORNL)

in various ways such as maximizing the mechanical advantage. Much research is currently underway on the problem of utilizing extra degrees of freedom of redundant manipulators.

Control methods generally determine joint velocities using either global or local optimization schemes. The complexity of the global control schemes requires the calculations to be done off-line. Such preplanned motion, however, is in direct conflict with the goal of avoiding moving obstacles in the path of the manipulator, that is, being able to operate in an unstructured environment. Therefore, it is desirable to have schemes that can be calculated on-line because, by using vision or proximity sensors or a combination of both, the manipulator may then respond to changes in its environment. It is possible for local optimization schemes, which are generally based on resolved motion rate control [5], to be used on-line for real-time optimization.

1.4. User Interface

There are two user interface problems which would be beneficial to address: robot control software which is hard to use, and visualization of what a manipulator's links may do when the end-effector is moving along a specified trajectory. Visualization of a manipulator's trajectory configurations from merely looking at a listing of joint angles is time consuming and difficult, and such knowledge is sometimes crucial to avoid subjecting an expensive manipulator to some unknown possibly damaging motion. It is also extremely useful for testing the feasibility and effectiveness of different control schemes before implementation. Graphic simulation of manipulator motion to test the control scheme for different trajectories is both easy to visualize and quickly done. A robotics program which provides a user friendly interface that makes parameter changes easy to perform and that provides a graphics capability is therefore desirable.

1.5. Thesis Objectives

The objective of this thesis was to develop a real-time robotic control scheme for the utilization of redundancy of the NASA LTM. It was also planned to develop a graphics simulation package to test the redundancy control scheme with several performance criteria. The following issues were addressed to accomplish the thesis objectives:

- Kinematics of the LTM
- Optimization schemes for kinematic redundancy resolution of manipulators
- Performance criteria for the LTM
- Cartesian space trajectory planning
- Joint space trajectory planning

- Real-time control
- Computational efficiency of the redundancy resolution schemes for real-time implementation
- User friendly computer interface for real-time control and simulation of LTM
- Portability of the software to integrate with the higher level software
- Modularity of the software for easy expansion.

1.6. Thesis Organization

Chapter 2 presents a background on kinematic control schemes for manipulators, singular configurations, redundant manipulators, and redundancy resolution schemes. Chapter 3 presents an efficient scheme for implementation of the Gradient Projection Method. Forward kinematics and inverse kinematics of the LTM for optimized redundancy resolution are discussed in Chapter 4. Chapter 5 presents the real-time control computer program overview along with the details of each function or subroutine in the program. Chapter 6 contains computer simulations demonstrating the effectiveness of the control scheme. User instructions for control software are given in Chapter 7. Conclusions and recommendations are presented in Chapter 8.

CHAPTER 2

BACKGROUND

Manipulators with redundant degrees of freedom are capable of avoiding configurations which would result in problems such as colliding with obstacles in the environment, reaching joint limits, and exceeding joint velocity limits. However, such a capability requires proper utilization of redundancy. This chapter presents background material on the control scheme for redundancy utilization.

2.1. Resolved Motion Rate Control

Linear and angular velocities of a manipulators's end-effector can be related to joint velocities by the equation

$$\dot{\mathbf{x}} = \mathbf{J} \dot{\mathbf{q}} \quad (2.1)$$

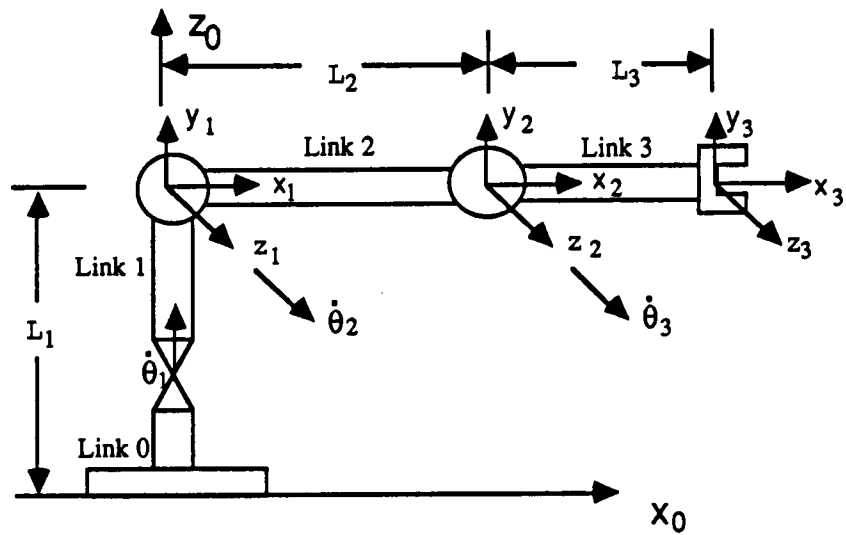
where $\mathbf{J}$ is an $m \times n$ matrix which transforms the $n \times 1$ joint velocity vector $\dot{\mathbf{q}}$ into an $m \times 1$ end-effector velocity vector $\dot{\mathbf{x}}$. The first column of $\mathbf{J}$ is the contribution of the first element of $\dot{\mathbf{q}}$ (for a unit velocity) to the velocity of the end-effector. Likewise, column two gives the contribution of the second element of $\dot{\mathbf{q}}$, and so on. The matrix $\mathbf{J}$ is known as the manipulator Jacobian. The standard solution for (2.1) is

$$\dot{\mathbf{q}} = \mathbf{J}^{-1} \dot{\mathbf{x}} \quad (2.2)$$

where $\mathbf{J}^{-1}$ is the inverse of the Jacobian. The method of (2.2) is called resolved motion rate control and is attributed to Whitney [2]. The elements of $\mathbf{J}$, however, are dependent on joint angles and therefore $\mathbf{J}$ varies with the configuration of the manipulator. At certain configurations $\mathbf{J}$ becomes singular and (2.2) is no longer valid. When a manipulator is in a configuration such that it has lost one or more degrees of freedom the Jacobian will be singular, and the manipulator is said to be in a singular configuration.

2.2. Singular Configurations

Singular configurations can be easily illustrated using the three-degree-of-freedom manipulator composed of three revolute joints shown in Fig. 2-1. When coordinate frames



Denavit-Hartenberg Parameters

Link _i	α_i	a_i	d_i	θ_i
1	90	0	L_1	θ_1
2	0	L_2	0	θ_2
3	0	L_3	0	θ_3

Figure 2-1 : A 3-degree-of-freedom manipulator

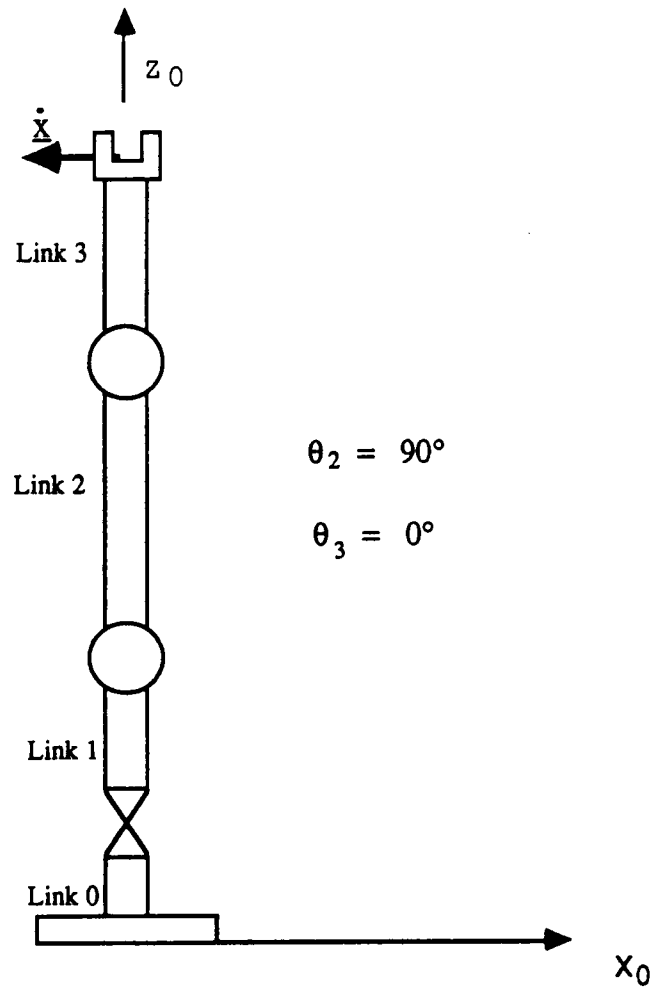
are affixed to the links in accordance with the Danevit-Hartenberg notation [3], we obtain parameters as shown in the same figure. Appendix A shows how both the forward (or direct) kinematic equations and the manipulator Jacobian are obtained from these parameters. If for this manipulator we let $\dot{\underline{x}}$ represent the 3×1 end-effector velocity vector, $\dot{\underline{\theta}}$ represent the 3×1 joint velocity vector, and J represent the Jacobian, then the relationship between velocities can be expressed as $\dot{\underline{x}} = J \dot{\underline{\theta}}$.

When the manipulator is in the configuration shown in Fig. 2-2, with $\theta_2 = 90^\circ$ and $\theta_3 = 0^\circ$, the end-effector cannot move in the Z direction along link 3. Furthermore, if we look at the manipulator Jacobian for this configuration, also shown in Fig. 2-2, we see that the row corresponding to a translational velocity in the Z direction is full of zeros. If we use the augmented matrix $[J, \dot{\underline{x}}]$ to investigate the existence of solutions for this particular system, we can immediately see that the system of equations is inconsistent when $\dot{\underline{x}} = [0, v_z, 0]^T$, where v_z is the component of the translational velocity in the Z direction. Indeed, when $\dot{\underline{x}}$ is *any* vector containing v_z then no solution exists for $\dot{\underline{x}} = J \dot{\underline{\theta}}$ and, therefore, the manipulator cannot move in directions associated with such vectors; it can, however, move in a direction perpendicular to link 2.

When the manipulator is in the configuration shown in Fig. 2-3, with $\theta_3 = 0^\circ$, the end-effector again cannot move in a direction along the length of link 3, but it does not have to move only in a direction perpendicular to link 3. Inspection of the Jacobian for the configuration shown in Fig. 2-3 will show that angular velocity $\dot{\theta}_1$ contributes to both v_x and v_y . At any value of θ_1 the angular velocity $\dot{\theta}_1$ creates a translational velocity $\dot{\underline{x}}_2$ that is perpendicular to both link 3 and $\underline{\dot{x}}_1$. The resultant translational velocity vector includes v_x , v_y , and v_z . The impossible end-effector velocity vector $\dot{\underline{x}}_{np}$ (where the subscripts np stands for not possible) may also contain the components v_x , v_y , and v_z for any particular value of θ_1 . The velocity components of $\dot{\underline{x}}_{np}$, however, must exist such that the augmented matrix $[J, \dot{\underline{x}}]$ is inconsistent.

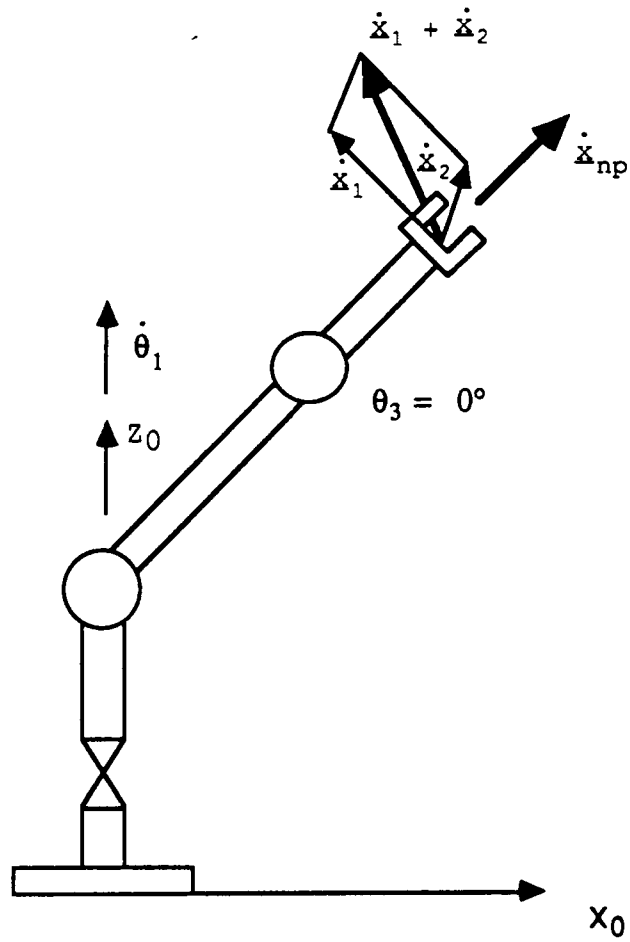
For a specific example Fig. 2-4 shows the same manipulator, but in a different configuration with $\theta_2 = 90^\circ$ and $\theta_3 = 0^\circ$. The augmented matrix $[J, \dot{\underline{x}}]$ and row canonical form of the augmented matrix for the shown configuration are also given. In order for an infinite number of solutions to exist the third row of the row canonical form of the augmented matrix must be full of zeros; therefore, for solutions to exist the following condition must be satisfied: $v_z = -(\cos\theta_2 / \sin\theta_2)v_y$. Any specified velocity vector in which this ratio did not exist would cause the augmented matrix to be inconsistent.

When the Jacobian is not of full rank, as in the past three examples, then it is singular and the manipulator is said to be in a singular configuration. If in general we define the direction of a vector to mean that property of the vector in which the components



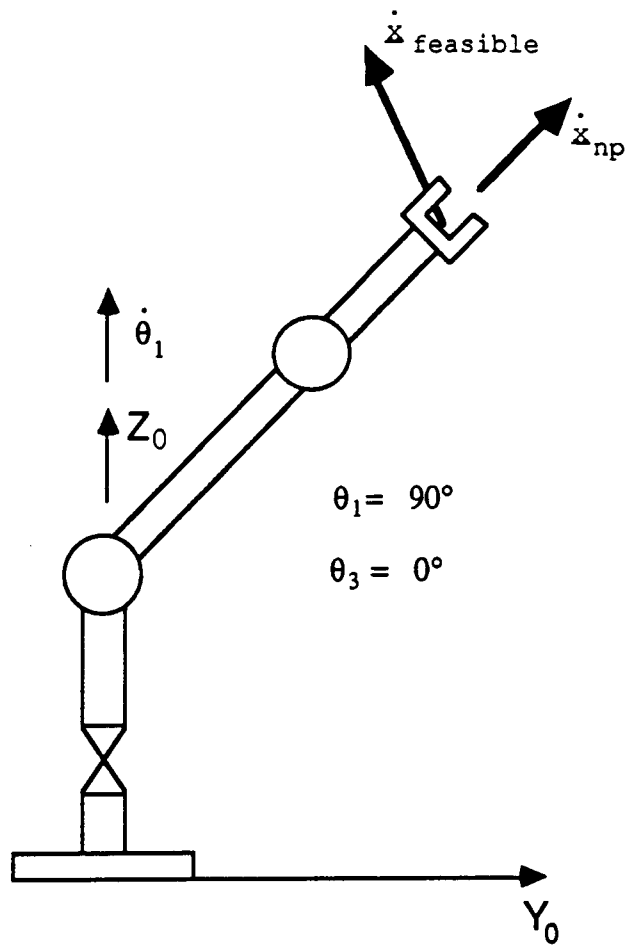
$$J = \begin{bmatrix} 0 & -(L_2 + L_3)\cos\theta_1 & -L_3\cos\theta_1 \\ 0 & -(L_2 + L_3)\sin\theta_1 & -L_3\sin\theta_1 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 2-2 : Singular configuration of a 3-degree-of-freedom manipulator (I)



$$J = \begin{bmatrix} -(L_2 + L_3)\sin\theta_1\cos\theta_2 & -(L_2 + L_3)\cos\theta_1\sin\theta_2 & -L_3\cos\theta_1\sin\theta_2 \\ (L_2 + L_3)\cos\theta_1\cos\theta_2 & -(L_2 + L_3)\sin\theta_1\sin\theta_2 & -L_3\sin\theta_1\sin\theta_2 \\ 0 & (L_2 + L_3)\cos\theta_2 & L_3\cos\theta_2 \end{bmatrix}$$

Figure 2-3 : Singular configuration of a 3-degree-of-freedom manipulator (II)



Augmented Matrix

$$\begin{bmatrix} J & \dot{\underline{x}} \end{bmatrix} = \begin{bmatrix} -(L_2 + L_3)\cos\theta_2 & 0 & 0 & V_x \\ 0 & -(L_2 + L_3)\sin\theta_2 & -L_3\sin\theta_2 & V_y \\ 0 & (L_2 + L_3)\cos\theta_2 & L_3\cos\theta_2 & V_z \end{bmatrix}$$

Augmented Matrix in Row Canonical Form

$$\begin{bmatrix} 1 & 0 & 0 & \frac{V_x}{-(L_2 + L_3)\cos\theta_2} \\ 0 & 1 & \frac{L_3}{L_2 + L_3} & \frac{V_y}{-(L_2 + L_3)\sin\theta_2} \\ 0 & 0 & 0 & V_z + \frac{V_y\cos\theta_2}{\sin\theta_2} \end{bmatrix}$$

Figure 2-4 : Singular configuration of a 3-degree-of-freedom manipulator (III)

of the vector exist in a certain specific ratio, as is intuitive with a vector existing in three dimensional space, then when a manipulator is in a singular configuration its end-effector is unable to move in at least one direction. In the first example the end-effector could not move in the principal Z direction. In the second and third examples impossible velocity vectors exist with components in all three principal directions. On the other hand feasible velocity vectors also exist with components in all three principal directions. That is, just because the manipulator has lost a degree of freedom does not mean it cannot move in a direction that requires a full set of velocity components. On the other hand, were the velocity vectors referred to the end-effector coordinate frame, rather than the inertial coordinate frame, then there is at least one *principal* direction, in relation to the coordinate frame of the end-effector, that the end-effector cannot move in. When speaking of a six-degree-of-freedom manipulator, angular velocities must be considered; however, the above statements are still valid.

But it is also desirable to avoid even the vicinity of a singular configuration. This is because when the configuration of the manipulator gets close to being singular the joint velocities required to keep the end-effector on its trajectory may exceed hardware limits on the joint velocities: the motors can no longer deliver the required speeds. Indeed, at a singular configuration boundless joint velocities would be necessary in order "to move" the manipulator's end-effector in a direction it cannot be moved. When $\dot{\mathbf{x}} = \mathbf{J} \dot{\mathbf{q}}$ is solved (see Appendix A) the following equations hold:

$$\dot{\theta}_1 = \frac{s_1 v_x - c_1 v_y}{\cos\theta_2 + \cos(\theta_2 + \theta_3)} \quad (2.3)$$

$$\dot{\theta}_2 = \frac{\cos(\theta_2 + \theta_3)(\cos\theta_1 v_x + \sin\theta_1 v_y) + \sin(\theta_2 + \theta_3)v_z}{\sin\theta_3} \quad (2.4)$$

$$\dot{\theta}_3 = \frac{v_z - [\cos\theta_2 + \cos(\theta_2 + \theta_3)]\dot{\theta}_2}{\cos(\theta_2 + \theta_3)} \quad (2.5)$$

These equations show that when θ_3 is close to 0° or $\pm 180^\circ$ the velocity $\dot{\theta}_2$ becomes large, and that whenever θ_2 is close to $\pm 90^\circ$ and θ_3 is close to 0° or $\pm 180^\circ$, the velocities $\dot{\theta}_2$ and $\dot{\theta}_3$ become large.

Like singular configurations, other undesirable conditions such as obstacles and joint limits must be handled optimally in some way. The next section discusses the role a redundant manipulator can play in optimization of a manipulator required to move its end-effector on a specified trajectory.

2.3. Redundant Manipulators and Optimization

Most manipulators currently in use have six or less degrees of freedom. A manipulator must have at least six degrees of freedom to move its end-effector with an arbitrary linear and angular velocity. Resolved motion rate control can be used to do this, but there are many configurations of the manipulator where the manipulator Jacobian becomes singular. At a singular configuration a manipulator is unable to move out of the subspace spanned by the independent column vectors of the Jacobian. If, however, there were at least one more degree of freedom, then that degree of freedom could be used to avoid a singularity while still moving the end-effector along a specified trajectory. For a six-degree-of-freedom manipulator where the Jacobian is a 6×6 nonsingular square matrix, there is a unique joint velocity solution for each set of specified end-effector velocities. However, for a seven-degree-of-freedom manipulator where the Jacobian is a 6×7 rectangular matrix of rank 6, there are an infinite number of solutions. A common way to describe this by an example is to tell one to put a hand on one edge of a table and notice that the elbow can be moved around in circles without changing the position of the hand. This is because the human arm has at least seven-degrees-of-freedom. Mathematically this is described using the concept of the null space.

2.3.1 The Null Space

The null space of the Jacobian J is that set of joint coordinate velocities that satisfy the equation $J \dot{\mathbf{q}} = \mathbf{0}$, where $\mathbf{0}$ is the zero vector for specified end-effector velocities. The null space is a subspace of the domain of the Jacobian and is such that the sum of the dimensions of the null space and solution space always equal the dimension of the Jacobian. We will consider a particular case that is easy to visualize. Figure 2-5 shows a simple redundant planar manipulator (which is not the same as the manipulator used in section 2.2 to illustrate singular configurations) with three revolute joints. The Jacobian of such a manipulator has three columns since it has three joints. Motion in a plane actually requires only two joints. Because the manipulator has one extra joint we may make statements about the Jacobian without determining what it actually is. It is obvious that the columns of the Jacobian can be reduced to a linearly independent set of two vectors. The null space, therefore, can be represented in Cartesian space by a line which passes through the origin, and the solution space by a plane parallel to that line. If we let $\dot{\mathbf{p}}$ be a particular set of joint velocities from the solution space of J , and $\dot{\mathbf{q}}$ be a set of joint velocities from the

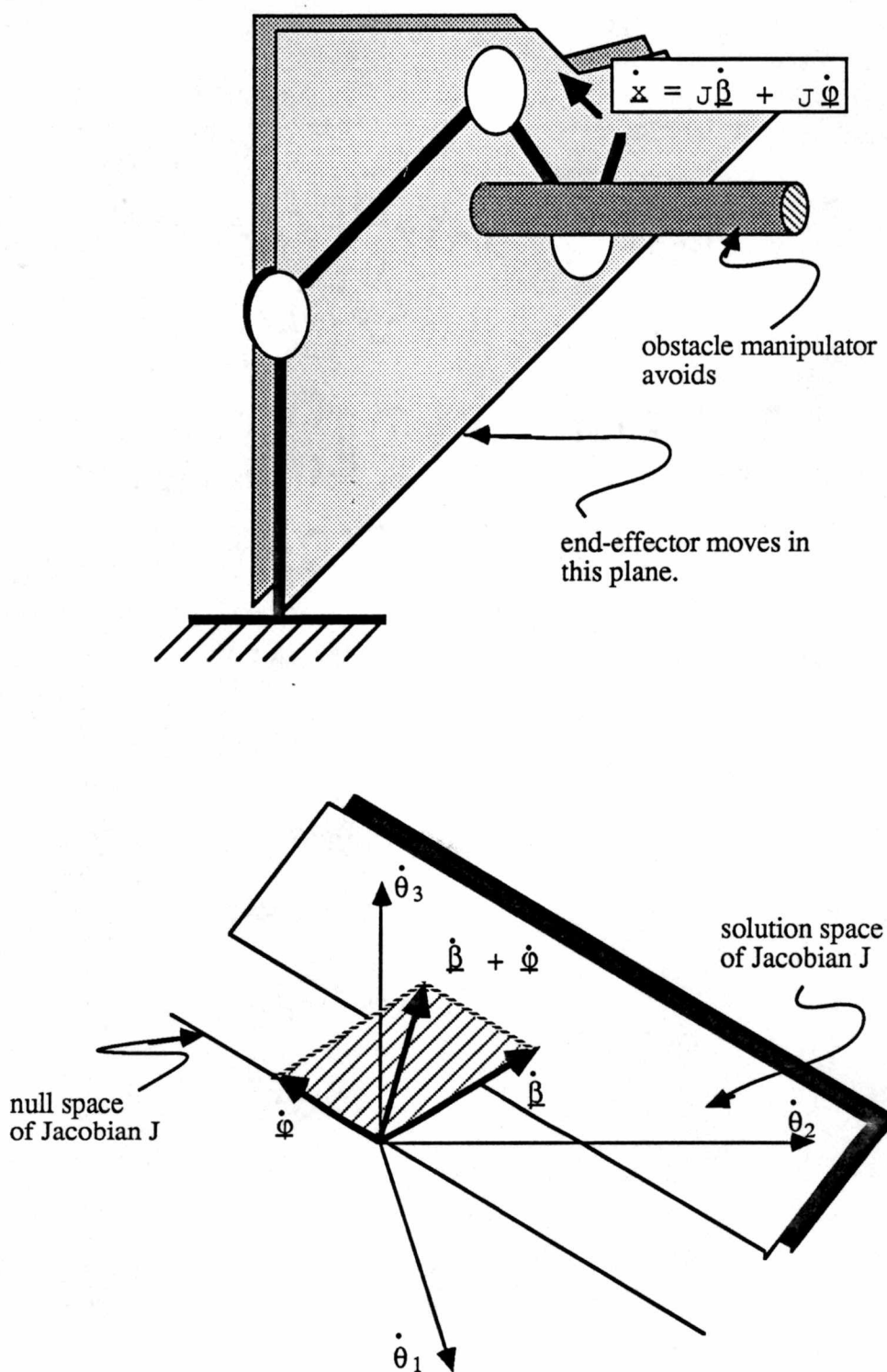


Figure 2-5: Planar manipulator utilizing the null space of manipulator Jacobian

null space of J , we may write equation (2.1) as $\dot{\mathbf{x}} = J \dot{\mathbf{p}} + J \dot{\mathbf{q}}$, where $\dot{\mathbf{q}} = \dot{\mathbf{p}} + \dot{\mathbf{q}}$. Summing the joint velocity vectors $\dot{\mathbf{p}}$ and $\dot{\mathbf{q}}$ merely serves to move the solution to some other point in the solution plane. Using $\dot{\mathbf{p}} + \dot{\mathbf{q}}$ as a solution means that the next instantaneous configuration of the manipulator will be different than if only $\dot{\mathbf{p}}$ were used as a solution to $\dot{\mathbf{x}} = J \dot{\mathbf{q}}$. If suitable values of joint velocities from the null space were chosen for $\dot{\mathbf{q}}$ such that, for instance, an obstacle was avoided as shown in Fig. 2-5, then the null space will have been well utilized.

2.3.2. Seven-Degree-of-Freedom Manipulator

In the previous section a planar redundant manipulator was discussed. This simple three-degree-of-freedom manipulator can serve as an analogy in some sense to a seven-degree-of-freedom manipulator where, like the example, the null space of a seven-degree-of-freedom manipulator can be thought of as a line in seven dimensional space and the solution space can be thought of as a hyperplane in the same seven dimensional space.

We can, therefore, use the null space of a seven-degree-of-freedom manipulator to optimize the joint velocities according to some criterion. In other words, if we are operating a seven-degree-of-freedom manipulator and the rank of the Jacobian for that manipulator is 6 at some instant in time, then we are able to move the end-effector with arbitrary translational and rotational velocities while optimizing the velocities of the joint angles.

Up until now we have assumed that we have solutions $\dot{\mathbf{p}}$ and $\dot{\mathbf{q}}$, but in the case of a seven-degree-of-freedom manipulator with a Jacobian J of rank 6, there is no inverse for J ; that is, J^{-1} does not exist. However, there are an infinite number of solutions in the solution space of J and we must first find a particular one, such as $\dot{\mathbf{p}}$ and then in order to optimize the joint velocities we must find a set of joint angles from the null space for that particular configuration. The next section shows how this can be done.

2.4. Inverse Kinematics

Even if the manipulator is non-redundant with six degrees of freedom and J is invertible, actually calculating J^{-1} is time consuming (certain manipulators having "solvable structures" allow closed-form solutions). Additionally, adding another joint as in the case of a seven-degree-of-freedom manipulator increases the complexity of the problem since there are now a boundless number of solutions when J is of rank 6. There are two

important tools that are used to overcome these difficulties: (1) the generalized inverse of a matrix, and (2) a special matrix called the null space projection matrix.

2.4.1 Generalized Inverse (Moore-Penrose pseudoinverse)

Let $A \underline{x} = \underline{b}$ be a *general* system of equations with no restrictions on consistency and where A is any $m \times n$ matrix with no restrictions on rank. Then $\underline{x} = A^+ \underline{b}$ is the solution to $A \underline{x} = \underline{b}$ which:

- (1) minimizes the sum of the squares of the residuals

$$\| \underline{r} \|^2 = \| A \underline{x} - \underline{b} \|^2, \text{ and} \quad (2.6)$$

- (2) minimizes the sum of the squares of the unknowns $\| \underline{x} \|^2$.

A^+ is called the Moore-Penrose pseudoinverse (generalized inverse) of A . It can be used with square and rectangular matrices in consistent and inconsistent sets of equations [4]. When the system of equations is inconsistent, it gives the best possible approximate solution according to the above criteria. Criterion (1) gives a solution which attempts to make $A \underline{x} - \underline{b} = 0$. Criterion (2) makes the solution unique [5]. If the system of equations is not inconsistent, then $\| \underline{r} \|^2$ is 0 since $A \underline{x}$ does equal $\underline{b}$. If its associated matrix is square and of full rank then, for a matrix A , A^+ is equal to A^{-1} , the inverse of A . This means that when it is possible to use the inverse of a matrix, the solution is always the least possible Euclidean norm. Note that other methods of minimizing $\| A \underline{x} - \underline{b} \|^2$ cannot in general be equated to A^{-1} when A is square and of full rank and, therefore, cannot be called generalized inverses.

However, in relation to the generalized inverse this paper is concerned only with an $m \times n$ rectangular matrix of full row rank with $m < n$. Such a matrix can be used to write the Jacobian of a redundant manipulator which is not in a singular configuration. What to do in the case of a singular configuration is not addressed, avoiding singular configurations is.

The pseudoinverse of a complicated 6×7 matrix, such as the Jacobian of a seven-degree-of-freedom redundant manipulator, would be extremely laborious to arrive at by hand, and to calculate it on a microcomputer such as the Motorola 68020 is a very time consuming process in relation to real-time motion. Notice, however, that the Moore-Penrose generalized inverse guarantees that the solution $\dot{\underline{q}}$ to the set of equations $\dot{\underline{x}} = J \dot{\underline{q}}$ is the *least possible* Euclidean norm $\| \dot{\underline{q}} \|_{\min}$ for that particular set of equations. Other generalized inverses do not guarantee this.

We can illustrate some interesting and useful results from considering an extremely trivial derivation of the Moore-Penrose pseudoinverse for an $m \times n$ matrix with $m = 1$ and $n = 2$. Figure 2-6 shows a Cartesian coordinate system with coordinates x_1 and x_2 . Lines CD and EF are drawn and from the graph we can see that equations for these lines can be written as

$$x_2 = \frac{60 - 3x_1}{4} \quad (2.7)$$

and

$$x_2 = \frac{0 - 3x_1}{4}. \quad (2.8)$$

But equations (2.7 and 2.8) can also be written in matrix form in the following way

$$\begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 60 \quad (2.9)$$

and

$$\begin{bmatrix} 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = 0. \quad (2.10)$$

Because line CD could have been drawn anywhere in the plane defined by axes x_1 and x_2 , we will express (2.10) in general (but with respect to this plane) as

$$A \underline{x} = \underline{y} \quad (2.11)$$

where A is a 1×2 rectangular matrix, $\underline{x}$ is a one dimensional vector of unknowns in x_1 and x_2 , and $\underline{y}$ is the desired result which is specified (refer to Fig. 2-7). Therefore the line CD represents the solution space of matrix A when $A = \begin{bmatrix} 3 & 4 \end{bmatrix}$. By definition all solutions to the equation $A \underline{x} = \underline{0}$ are members of the null space of A . Line EF, therefore, defines the null space for $A = \begin{bmatrix} 3 & 4 \end{bmatrix}$. By inspection it can be seen that, indeed, any set of points $\underline{x}$ on line EF result in a value of $\underline{0}$ for $\underline{y}$ if $\underline{x}$ is multiplied by the matrix $A = \begin{bmatrix} 3 & 4 \end{bmatrix}$. Of course, for this very simple example, $\underline{0}$ and $\underline{y}$ are each composed of 1 element.

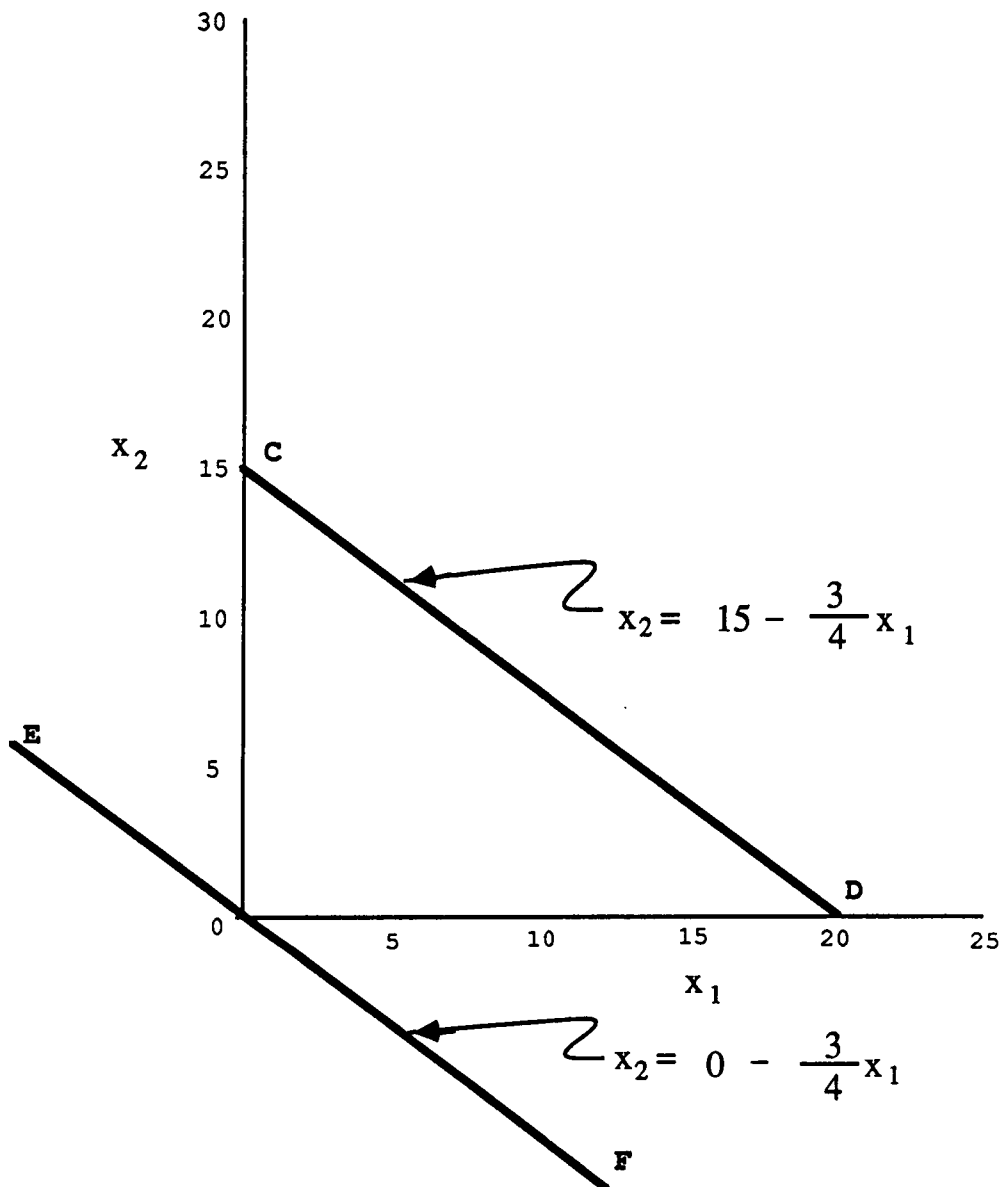


Figure 2-6 : Lines CD and EF drawn on Cartesian coordinate system.

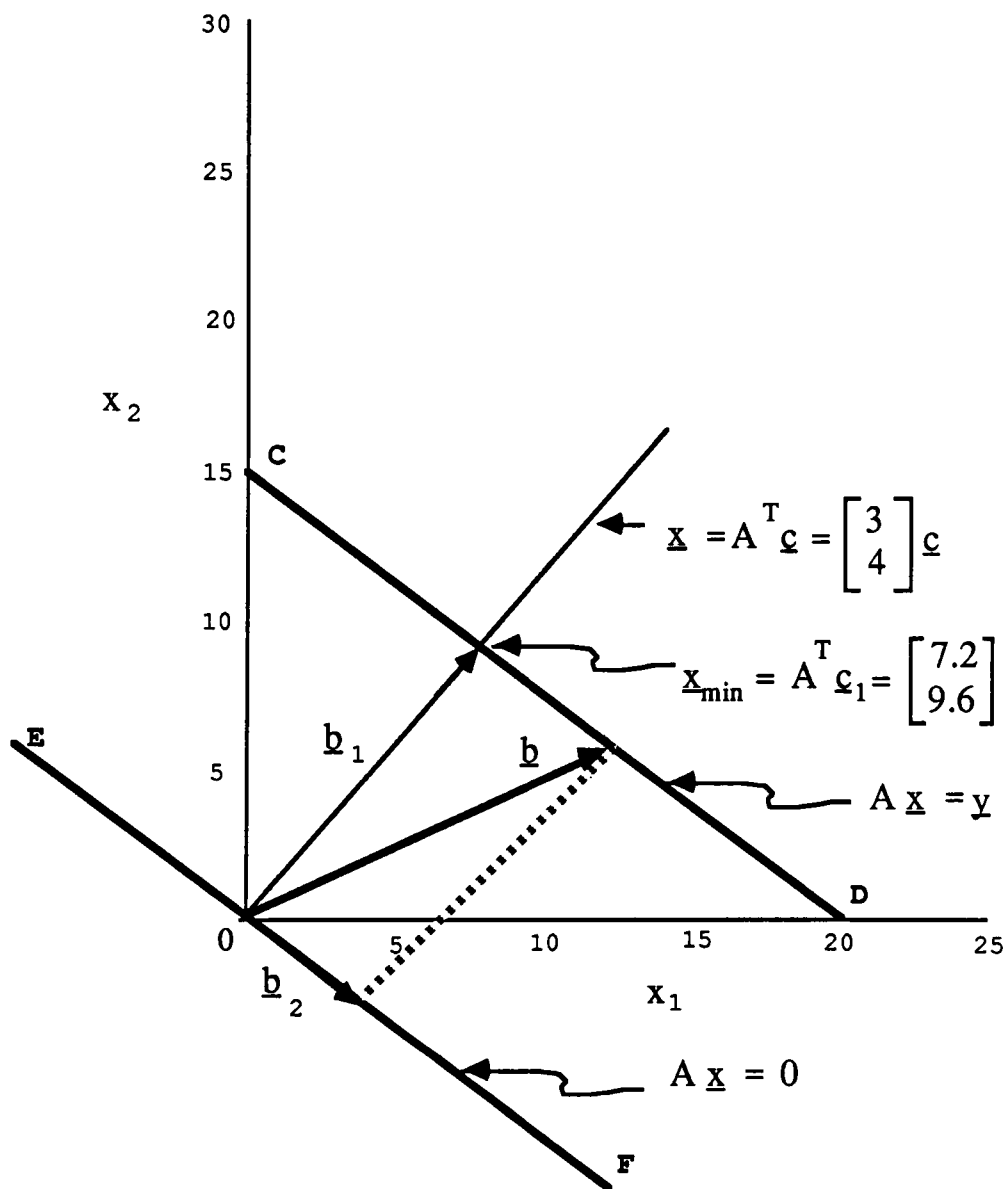


Figure 2-7 : Matrix representation of lines CD and EF

We now draw position vector $\underline{b}$ such that it extends to any solution on line CD, but is otherwise arbitrary. Next a line OG is carefully drawn from the origin so that it meets line CD at a 90° angle. And finally we project $\underline{b}$ onto the null space and the line OG, obtaining the vectors $\underline{b}_1$ and $\underline{b}_2$. Because the shortest distance between two parallel lines is another line drawn perpendicular to both of them, we see that vector $\underline{b}_1$ is the solution of (2.11) with least Euclidean norm, and therefore we can write

$$\underline{b}_1 = \begin{bmatrix} 7.2 \\ 9.6 \end{bmatrix} = A^+(60), \quad (2.12)$$

and in general we may write

$$\underline{b}_1 = \underline{x}_{\min} = A^+ \underline{y}, \quad (2.13)$$

where $\underline{x}_{\min}$ is defined as $\| \underline{x} \|_{\min}$, the minimum Euclidean norm obtained by using the Moore-Penrose pseudoinverse. If it can be shown that

$$\underline{b}_1 = G \underline{y}, \quad (2.14)$$

then obviously

$$A^+ = G. \quad (2.15)$$

Now, the *row space* of a matrix is defined using the transpose of that matrix and is always orthogonal to the null space. The row space in general is defined by the equation

$$\underline{x} = A^T \underline{c} \quad (2.16)$$

For $A = \begin{bmatrix} 3 & 4 \end{bmatrix}$ equation (2.16) becomes

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} c_1 \quad (2.17)$$

where c_1 is an arbitrary constant. If we choose c_1 to be the value 2.4, then $x_1 = 7.2$, $x_2 = 9.6$, and therefore $\underline{b}_1 = A^T \underline{c}_1$. It is also seen that the row space is a line that goes through both the origin and the point (7.2,9.6). Therefore $\underline{x}_{\min} = A^+ \underline{y}$ is in the row space of A. This should always be so since the line originating from the origin and perpendicular

to both the null space and the solution space will always contain the line of shortest distance between the two spaces. Now we may begin:

$$\underline{b} = \underline{b}_1 + \underline{b}_2, \quad (2.18)$$

$$\underline{b}_1 = A^T \underline{c}_1 \quad (2.19)$$

$$\underline{b} = A^T \underline{c}_1 + \underline{b}_2, \quad (2.20)$$

$$\underline{b}_2 = \underline{b} - A^T \underline{c}_1. \quad (2.21)$$

But $\underline{b}_2$ is in the null space of A and therefore,

$$A \underline{b}_2 = 0, \quad (2.22)$$

$$A (\underline{b} - A^T \underline{c}_1) = 0 \quad (2.23)$$

$$A A^T \underline{c}_1 = A \underline{b} \quad (2.24)$$

Because $A A^T$ is always nonsingular if A is of full rank [5] we can invert $A A^T$

$$\underline{c}_1 = (A A^T)^{-1} A \underline{b} \quad (2.25)$$

$$A^T \underline{c}_1 = A^T (A A^T)^{-1} A \underline{b} \quad (2.26)$$

$$\underline{b}_1 = A^T (A A^T)^{-1} A \underline{b} \quad (2.27)$$

But recall that the vector $\underline{b}$ has been arbitrarily drawn to any solution on line CD and therefore

$$\underline{b}_1 = A^T (A A^T)^{-1} A \underline{x} \quad (2.28)$$

$$\underline{b}_1 = A^T (A A^T)^{-1} \underline{y} \quad (2.29)$$

and therefore, according to equations (2.14) and (2.15)

$$A^+ = A^T (A A^T)^{-1}. \quad (2.30)$$

The special matrix $A^T (A A^T)^{-1}$ is the Moore-Penrose pseudoinverse A^+ . We can also find a way to project $\underline{b}$ onto the null space of matrix A :

$$\underline{b}_2 = \underline{b} - \underline{b}_1 \quad (2.31)$$

$$\underline{b}_2 = \underline{b} - A^+ A \underline{b} \quad (2.32)$$

$$\underline{b}_2 = (I - A^+ A) \underline{b} \quad (2.33)$$

We can use $(I - A^+ A)$ to project any vector $\underline{b}$ onto the null space of A . But the matrix A need not be confined to a one row matrix.

We can imagine the line CD as being just one line in a plane defining the solution space of A . Then A could be a 2×3 rectangular matrix of rank 2. If we allow the solution space of A to be termed a hyperplane (which is the term that extends the concept of a plane in three-dimensional space to spaces of greater than three dimensions), then A could be of any size with the provision that there is one more column than there are rows. A seven-degree-of-freedom manipulator falls into this category and, therefore, equations (2.13) and (2.33) also apply in the case of a seven-degree-of-freedom manipulator like the LTM.

Now, assuming that we can use the generalized inverse to find a vector of joint velocities from the solution space of the manipulator Jacobian J that will move the end-effector with the desired end-effector velocities, we must now optimize the joint velocities in a beneficial way. The method used is called the gradient projection method.

2.4.2 The Gradient Projection Method

Consider again the equation $\dot{\underline{x}} = J \dot{\underline{\beta}} + J \dot{\underline{\phi}}$, where $\dot{\underline{\beta}}$ is a particular set of joint velocities from the solution space of J , and $\dot{\underline{\phi}}$ is a set of joint velocities from the null space of J . We may use equations (2.13 and 2.33) to write the following two equations:

$$\dot{\underline{\beta}} = J^+ \dot{\underline{x}}, \quad (2.34)$$

and

$$\dot{\underline{\phi}} = (I - J^+ J) \dot{\underline{\phi}}, \quad (2.35)$$

where $\dot{\underline{\phi}}$ is an arbitrary joint velocity vector and $(I - J^+ J)$ is a special matrix called the null space projection matrix. It was shown in the last section that the null space projection

matrix can be used to project an arbitrary vector onto the null space of J . If we do so the solution to $\dot{\underline{x}} = J \dot{\underline{\theta}}$ can be expressed as

$$\dot{\underline{\theta}} = J^+ \dot{\underline{x}} + (I - J^+ J) \dot{\underline{\phi}}. \quad (2.36)$$

The arbitrary vector $\dot{\underline{\phi}}$ can be used to optimize the joint velocities of the manipulator. This is commonly done by choosing a scalar function H that is a function of the manipulator's *joint angles*, and note, *not* the manipulator's joint velocities. The gradient ∇H , is then used as $\dot{\underline{\phi}}$ in (2.36). The gradient of a scalar function always gives the direction of maximum increase of that function at any particular point. Therefore, it must be multiplied by a signed scalar constant k to ensure that maximization or minimization is occurring. Additionally, because the gradient increase may result in joint velocities unreachable by the manipulator's motors, the constant k must be scaled to ensure that joints velocities do not exceed velocity bounds. We now have the following equation:

$$\dot{\underline{\theta}} = J^+ \dot{\underline{x}} + k(I - J^+ J) \nabla H. \quad (2.37)$$

Because the second term on the right side of (2.37) projects ∇H onto the null space of J , this method of obtaining a solution to $\dot{\underline{x}} = J \dot{\underline{\theta}}$ is called the gradient projection method.

The null space of a seven-degree-of-freedom manipulator with Jacobian J of rank 6 can be thought of as a line in 7-dimensional space. The component of the gradient ∇H in the null space, therefore, "chooses" in which one of the two directions defined by that line optimization should occur. This relation is shown in Fig. 2-8.

Using (2.37) we can calculate the joint velocities necessary to move the end-effector on a particular trajectory while optimizing the configuration of the manipulator. However the necessity of calculating the generalized inverse precludes real-time calculations on a microcomputer such as the MC68020.

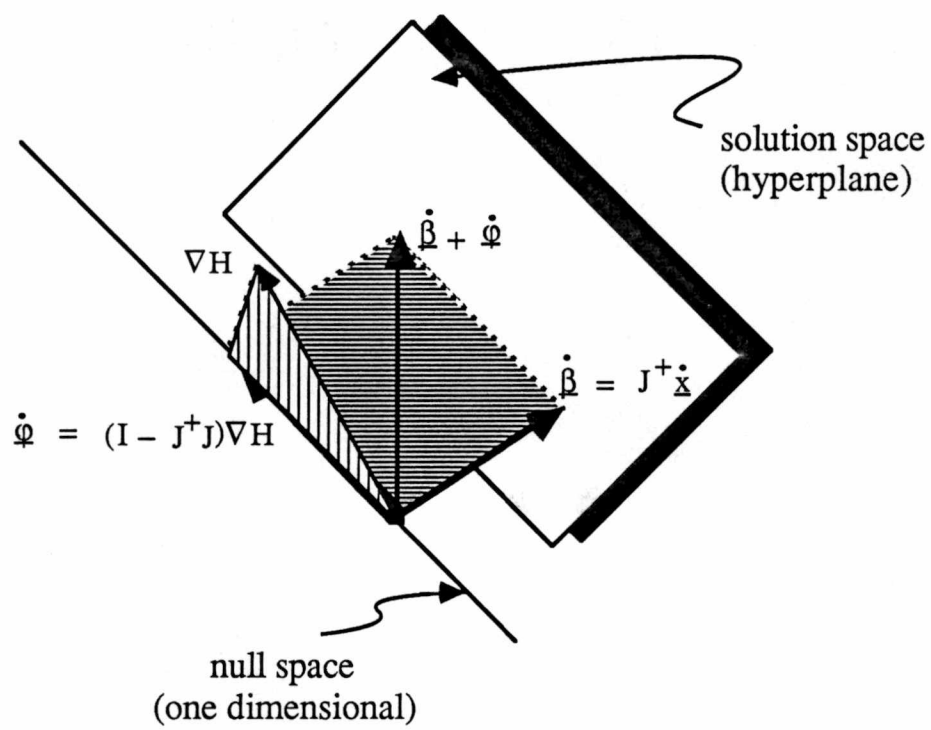


Figure 2-8: Gradient projection method for redundant manipulator with one extra degree of freedom

CHAPTER 3

AN EFFICIENT IMPLEMENTATION OF THE GRADIENT PROJECTION METHOD

Chapter 2 presented background material which led up to the well known gradient projection method of controlling redundant manipulators. The gradient projection method is expressed by the equation:

$$\dot{\mathbf{Q}} = \mathbf{J}^+ \dot{\mathbf{x}} + k(\mathbf{I} - \mathbf{J}^+ \mathbf{J}) \nabla H. \quad (3.1)$$

The problem with using (3.1) as it is written is in calculating the Moore-Penrose pseudoinverse for a full ranked 6×7 Jacobian, $\mathbf{J}^+ = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}$, which requires finding the inverse of $\mathbf{J}\mathbf{J}^T$. To move the end-effector along a specified path, a new set of joint increments must be calculated for each new increment in the path. Because the elements of the Jacobian depend on the configuration of the manipulator, the pseudoinverse $\mathbf{J}^+$ must be recalculated for each increment.. The calculations are so time consuming that real-time motion using a microprocessor such as the MC68020 becomes impractical. Instead of inverting $\mathbf{J}\mathbf{J}^T$, Klein and Huang suggested that Gaussian elimination be used [6]. Dubey et al. [7] developed a scheme which does not require calculation of the Moore-Penrose pseudoinverse. Their scheme reduces the number of calculations used in the Gaussian elimination method of Klein and Huang by at least threefold [7]. Chapter 3 presents the scheme developed by Dubey et al. and then shows a way to further reduce the number of calculations by over 50%. It is this improved version that is used for the LTM.

3.1. Avoiding the Moore-Penrose Pseudoinverse Calculations

The method used by Dubey et al. [8] which arrives at the same solution that $\mathbf{J}^+ \dot{\mathbf{x}}$ does without having to calculate the pseudoinverse, depends on the fact that the pseudoinverse of the Jacobian gives the least Euclidean norm of $\dot{\mathbf{Q}}$ when it is applied to the problem of finding the joint velocity vector which would minimize $\|\mathbf{J} \dot{\mathbf{Q}} - \dot{\mathbf{x}}\|^2$. In general other methods of minimizing $\|\mathbf{J} \dot{\mathbf{Q}} - \dot{\mathbf{x}}\|^2$ would not give the least Euclidean norm [4].

Recall that the solution to $\dot{\mathbf{x}} = \mathbf{J} \dot{\mathbf{Q}}$ can be divided into two parts called the particular and homogeneous solutions. The particular solution is a solution to $\dot{\mathbf{x}} = \mathbf{J} \dot{\mathbf{Q}}$ with $\dot{\mathbf{x}}$ arbitrary, while the homogeneous solution is the solution to $\mathbf{J} \dot{\mathbf{Q}} = \mathbf{0}$. If $\mathbf{J}$ is an 6×6 matrix of full rank, then the homogeneous solution is the zero vector and the particular

solution is the unique solution to $\dot{\underline{x}} = J \dot{\underline{\theta}}$. However, if J is a 6×7 matrix and of full rank, then there is a boundless number of solutions to both the particular and homogeneous parts of $\dot{\underline{\theta}}$. Let us express the general solution $\dot{\underline{\theta}}$ when J is a 6×7 matrix of full rank with the equation

$$\dot{\underline{\theta}} = \dot{\underline{\theta}}_p + \kappa \dot{\underline{\theta}}_h, \quad (3.2)$$

where $\dot{\underline{\theta}}_p$ designates any particular solution, $\dot{\underline{\theta}}_h$ designates any homogeneous solution, and κ is an arbitrary scalar constant. We let $\dot{\underline{\theta}}_{\min} = \|\dot{\underline{\theta}}\|_{\min}$, where $\|\dot{\underline{\theta}}\|_{\min}$ is the least Euclidean norm of all possible $\dot{\underline{\theta}}$ for a particular J and a particular $\dot{\underline{x}}$. As mentioned earlier, because the Moore-Penrose pseudoinverse always gives $\dot{\underline{\theta}}_{\min}$ as the solution to $\dot{\underline{x}} = J \dot{\underline{\theta}}$, we need to find $\dot{\underline{\theta}}_{\min}$. This is done by choosing the scalar constant κ such that the square of the Euclidean norm of (3.2) is at a minimum:

$$\|\dot{\underline{\theta}}\|^2 = \|\dot{\underline{\theta}}_p + \kappa \dot{\underline{\theta}}_h\|^2 \quad (3.3)$$

$$\|\dot{\underline{\theta}}\|^2 = (\dot{\underline{\theta}}_p + \kappa \dot{\underline{\theta}}_h)^T (\dot{\underline{\theta}}_p + \kappa \dot{\underline{\theta}}_h). \quad (3.4)$$

The operation on the right hand side of (3.4) is the dot product of two identical vectors which depend on the arbitrary constant κ ; therefore, they may be expressed as vector functions, call them $\underline{v}(\kappa)$; however, they will be written simply as $\underline{v}$. Equation (3.4) becomes

$$\|\dot{\underline{\theta}}\|^2 = \underline{v} \cdot \underline{v}. \quad (3.5)$$

Taking the derivative of (3.5) (the derivative will be indicated with the primed symbol ') with respect to κ gives

$$(\underline{v} \cdot \underline{v})' = \underline{v}' \cdot \underline{v} + \underline{v} \cdot \underline{v}' \quad (3.6)$$

$$(\underline{v} \cdot \underline{v})' = 2(\underline{v} \cdot \underline{v}') \quad (3.7)$$

$$(\underline{v} \cdot \underline{v})' = 2(\underline{v}^T \underline{v}') \quad (3.8)$$

$$(\underline{v} \cdot \underline{v})' = 2(\dot{\underline{\theta}}_p + \kappa \dot{\underline{\theta}}_h)^T \dot{\underline{\theta}}_h \quad (3.9)$$

The derivative is then set to zero and solved for κ .

$$(\dot{\underline{q}}_p + \kappa \dot{\underline{q}}_h)^T \dot{\underline{q}}_h = 0 \quad (3.10)$$

$$\dot{\underline{q}}_p^T \dot{\underline{q}}_h + \kappa \dot{\underline{q}}_h^T \dot{\underline{q}}_h = 0 \quad (3.11)$$

$$\kappa = - \frac{\dot{\underline{q}}_p^T \dot{\underline{q}}_h}{\dot{\underline{q}}_h^T \dot{\underline{q}}_h} \quad (3.12)$$

The value of κ found using (3.12) gives an extremum of (3.3). We are guaranteed that this extremum is a minimum since mathematically joint velocities may increase without bound. Any extremum, therefore, must be a minimum. When κ is substituted back into (3.2) we obtain

$$\dot{\underline{q}}_{\min} = \dot{\underline{q}}_p - \left(\frac{\dot{\underline{q}}_p^T \dot{\underline{q}}_h}{\dot{\underline{q}}_h^T \dot{\underline{q}}_h} \right) \dot{\underline{q}}_h \quad (3.13)$$

Dubey et al.[8] point out that if $\underline{u}_h$ is defined to be a unit vector in the direction of $\dot{\underline{q}}_h$, then $\dot{\underline{q}}_{\min}$ can be written as

$$\dot{\underline{q}}_{\min} = \dot{\underline{q}}_p - (\dot{\underline{q}}_p^T \underline{u}_h) \underline{u}_h \quad (3.14)$$

where

$$\underline{u}_h = \dot{\underline{q}}_h / \sqrt{\dot{\underline{q}}_h^T \dot{\underline{q}}_h} \quad (3.15)$$

and it can now be seen that the least Euclidean norm is obtained by "subtracting from a particular solution the component of the particular solution in the direction of a homogeneous solution".

This method of finding the least-norm solution to $\dot{\underline{x}} = J \dot{\underline{q}}$ can be used as long as we already have both a particular solution and a homogeneous solution. In the case of a seven-degree-of-freedom manipulator of full row rank Dubey et al. [8] propose the

following scheme to obtain a particular and a homogeneous solution to be used in finding the least-norm solution.

3.2. Particular and Homogeneous Solutions

If the Jacobian J of a seven-degree-of-freedom manipulator is of rank six, then we can always find six columns that are linearly independent. We can arrange the elements of the unknown vector $\dot{\underline{Q}}$ and the columns of J so that the six linearly independent columns are together and the remaining column is column one. If we designate the six linearly independent columns as J^* , then J^* is a 6×6 nonsingular matrix with an inverse J^{*-1} . If we designate the remaining dependent column as $\underline{\alpha}$, then we can express $\dot{\underline{x}} = J \dot{\underline{Q}}$ as

$$\dot{\underline{x}} = [\underline{\alpha} J^*] \dot{\underline{Q}} \quad (3.16)$$

In order to obtain a particular solution we solve $\dot{\underline{x}} = J \dot{\underline{Q}}_p$. This is done by arbitrarily setting the first element of $\dot{\underline{Q}}_p$ to 0, which in effect fixes the corresponding joint leaving only six degrees of freedom. A particular solution $\dot{\underline{Q}}_p$ can then be obtained using the following equation:

$$\dot{\underline{Q}}_p = \begin{bmatrix} 0 \\ J^{*-1} \dot{\underline{x}} \end{bmatrix} \quad (3.17)$$

In order to obtain a homogeneous solution we solve $J \dot{\underline{Q}}_h = 0$. This is done by arbitrarily setting the first element of $\dot{\underline{Q}}_h$ to 1, which leaves it free to be scaled by the scalar constant κ (note that when κ is 0 then the manipulator reduces to a six degree-of-freedom manipulator because the solution for $\dot{\underline{Q}}_p$ is holding one of the joint angles fixed at some value). A homogeneous solution $\dot{\underline{Q}}_h$ can then be obtained using the following equation:

$$\dot{\underline{Q}}_h = \begin{bmatrix} 1 \\ -J^{*-1} \underline{\alpha} \end{bmatrix} \quad (3.18)$$

When equations (3.17 and 3.18) are substituted into $\dot{\underline{Q}} = \dot{\underline{Q}}_p + \kappa \dot{\underline{Q}}_h$, the following arbitrary solution to $\dot{\underline{x}} = J \dot{\underline{Q}}$ is obtained :

$$\dot{\underline{Q}} = \begin{bmatrix} \kappa \\ J^{*-1}(\dot{\underline{x}} - \kappa \underline{\alpha}) \end{bmatrix}. \quad (3.19)$$

When a value for κ is chosen using (3.12), equation (3.19) becomes:

$$\dot{\underline{Q}}_{\min} = \begin{bmatrix} -\frac{\dot{\underline{Q}}_p^T \dot{\underline{Q}}_h}{\dot{\underline{Q}}_h^T \dot{\underline{Q}}_h} \\ J^{*-1}(\dot{\underline{x}} + \frac{\dot{\underline{Q}}_p^T \dot{\underline{Q}}_h}{\dot{\underline{Q}}_h^T \dot{\underline{Q}}_h} \underline{\alpha}) \end{bmatrix}. \quad (3.20)$$

In relation to a seven-degree-of-freedom manipulator with Jacobian J of full row rank, (3.20) expresses the approach presented by Dubey, Euler, and Babcock [8] for obtaining the least Euclidean norm solution to $\dot{\underline{x}} = J \dot{\underline{Q}}$ when arbitrary particular and homogeneous solutions $\dot{\underline{Q}}_p$ and $\dot{\underline{Q}}_h$ are first obtained using equations (3.18 and 3.19). Applications of this approach are currently being investigated in respect to redundant manipulators with greater than seven degrees of freedom. In addition, applications to other areas of robotics where the Moore-Penrose pseudoinverse traditionally would have to be calculated are also being investigated.

3.3. Dubey-Euler-Babcock Method for Determining Least-Norm Solutions

The method used by Dubey et al. for finding the least-norm joint velocity solution for a specified set of end-effector velocities using a 6×7 Jacobian J of full rank can be extended to any $m \times n$ matrix A of rank m with $n = m + 1$:

If we let A be such a matrix, $\underline{x}$ be an $n \times 1$ vector, and $\underline{y}$ be an $m \times 1$ vector, then the Dubey-Euler-Babcock method of finding a solution to $\underline{y} = A \underline{x}$ is to find the least Euclidean norm solution through the minimization of $\| \underline{x}_p + \kappa \underline{x}_h \|^2$ —where $\underline{x}_p$ is a particular solution to $\underline{y} = A \underline{x}$, $\underline{x}_h$ is a homogeneous solution to $A \underline{x} = 0$, and κ is an arbitrary scalar constant. Their method can be expressed symbolically as follows:

$$\underline{x}_{\min} = \| \underline{x} \|_{\min} = A^{\oplus} \underline{y}. \quad (3.21)$$

The special superscript $\oplus$ is affixed to the matrix A and introduced here for the first time. It indicates the following operations in relation to A and the vector y :

$$\underline{x}_p = \begin{bmatrix} 0 \\ A^{*-1} y \end{bmatrix} \quad (3.22)$$

$$\underline{x}_h = \begin{bmatrix} 1 \\ -A^{*-1} \underline{\alpha} \end{bmatrix} \quad (3.23)$$

$$\kappa = - \frac{\underline{x}_p^T \underline{x}_h}{\underline{x}_h^T \underline{x}_h} \quad (3.24)$$

$$\underline{x}_{min} = \begin{bmatrix} \kappa \\ A^{*-1} (\underline{x} - \kappa \underline{\alpha}) \end{bmatrix}. \quad (3.25)$$

where A^* is an $m \times m$ nonsingular submatrix of A , and $\underline{\alpha}$ is an $m \times 1$ column vector of A . Because A is of full rank $\underline{\alpha}$ must necessarily be dependent on the columns of A^* . The arrangement of A^* and $\underline{\alpha}$ must always be in the order indicated by the following equation

$$y = [\underline{\alpha} A^*] \underline{x}, \quad (3.26)$$

which may require manipulation of the columns of A . It is important to understand that $A^\oplus$ has no meaning by itself, it must always appear with a vector beside it.

Dubey et al. pointed out that the least Euclidean norm solution to $y = A \underline{x}$ is obtained by "subtracting from the particular solution the component of the particular solution in the direction of the homogeneous solution". Let us then define the operation $A^\oplus$ upon y in this alternate way:

DEFINITION 1: When a particular solution $\underline{x}_p$ and a homogeneous solution $\underline{x}_h$ are found for the system of equations $y = A \underline{x}$ (where A is an $m \times n$ matrix of rank m with $n = m + 1$, $\underline{x}$ is an $n \times 1$ vector, and y is an $m \times 1$ vector) using equations (3.22 and 3.33), then the operation of $A^\oplus$ upon y can alternately be defined in the following manner: $A^\oplus y$ indicates that the least Euclidean norm solution to $y = A \underline{x}$ can be obtained by subtracting from the particular solution $\underline{x}_p$ the component of the particular solution in the direction of

the homogeneous solution $\underline{x}_h$. This operator will be termed the Dubey-Euler-Babcock operator; it may not be separated from the vector $\underline{y}$, and it is limited in scope to the above defined type of matrix.

3.4. Relationship Between $A^\oplus \underline{y}$ and $A^+ \underline{y}$

If A is an $m \times n$ matrix of rank m with $m = n-1$ in the equation $\underline{y} = A \underline{x}$, where $\underline{y}$ is a particular $m \times 1$ specified vector, then there must be a relationship between $A^\oplus$ and A^+ with respect to $\underline{y}$ because:

$$\underline{x}_{\min} = A^\oplus \underline{y} \quad (3.27)$$

$$\underline{x}_{\min} = A^+ \underline{y} \quad (3.28)$$

Is the operation on $\underline{y}$ by $A^\oplus$ equivalent to the operation on $\underline{y}$ by A^+ ? Consider first the minimization of $\|A \underline{x} - \underline{y}\|^2$: Because A is of full row rank, not only is the augmented matrix consistent but there are an infinite number of solutions; therefore, any method which gives a solution to $\underline{y} = A \underline{x}$ gives a minimum for $\|A \underline{x} - \underline{y}\|^2$ and therefore the least squares condition is automatically satisfied by $A^\oplus \underline{y}$.

Next we consider Fig. 3-1, which is similar to Fig. 2-7. Vector $\underline{b}$, however, is indicated as being some particular solution $\underline{x}_p$ of $\underline{y} = A \underline{x}$. Additionally, a unit vector $\underline{u}_h$ in the direction of the homogeneous solution $\underline{x}_h$ is drawn. It is easy to see that the component of the particular solution $\underline{x}_p$ in the direction $\underline{u}_h$ of the homogeneous solution is the vector $\underline{b}_2$. Applying Definition 1 above and (3.27) gives

$$\underline{x}_{\min} = A^\oplus \underline{y} = \underline{x}_p - (\underline{x}_p^T \underline{u}_h) \underline{u}_h \quad (3.29)$$

and proceeding from Fig. 3-1 we have

$$A^\oplus \underline{y} = \underline{x}_p - \underline{b}_2 \quad (3.30)$$

$$A^\oplus \underline{y} = A^T \underline{c}_1 \quad (3.31)$$

$$A^\oplus \underline{y} = \underline{b}_1 = A^T (A A^T)^{-1} A \underline{x}_p = A^T (A A^T)^{-1} \underline{y} \quad (3.32)$$

Multiplying both sides of (3.32) by A gives

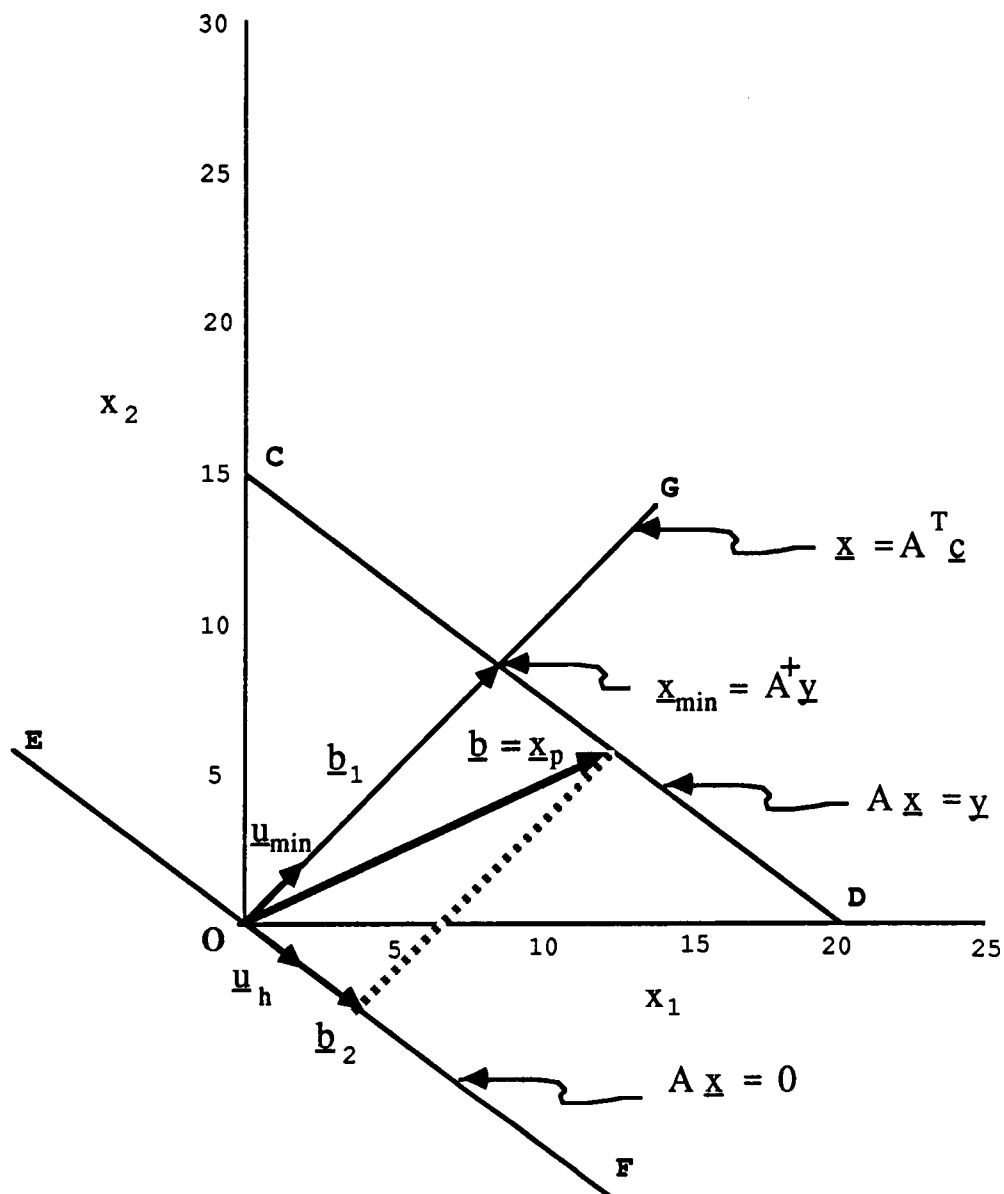


Figure 3-1: Particular solution projected onto null space and row space of matrix A

$$A A^{\oplus} y = A A^T (A A^T)^{-1} y \quad (3.33)$$

$$A A^{\oplus} y = y \quad (3.34)$$

$$A^{\oplus} y = A^+ y \quad (A^{\oplus} \neq A^+ !) \quad (3.35)$$

A few things are immediately obvious:

(1) $A^{\oplus} y$ is equivalent to $A^+ y$ for the special case when the $m \times n$ matrix A is of full row rank and $m = n - 1$; however, $A^{\oplus}$ is not equivalent to A^+ .

(2) $A^T (A A^T)^{-1}$ is the generalized inverse for a rectangular matrix of full row rank with $m < n$ (see equation (2.30) and [5]).

$$A^+ = A^T (A A^T)^{-1} \quad (3.36)$$

(3) The least Euclidean norm solution to $y = A x$ can also be obtained by the orthogonal projection of x_p onto the row space of A , which is the operation of $A^{\oplus} y$. Observe, if $u_{\min}$ is a unit vector in the direction of $x_{\min}$, then from Fig.3-1

$$\left(x_p^T u_{\min} \right) u_{\min} = x_p - \left(x_p^T u_h \right) u_h \quad (3.37)$$

$$b_1 = x_p - \left(x_p^T u_h \right) u_h \quad (3.38)$$

$$b_1 = A^T (A A^T)^{-1} A x_p \quad (3.39)$$

$$x_p - \left(x_p^T u_h \right) u_h = A^T (A A^T)^{-1} A x_p \quad (3.40)$$

The matrix $A^T (A A^T)^{-1} A$ is obviously the "row space projection matrix" of the matrix A . By rearranging (3.40) we have

$$\left(x_p^T u_h \right) u_h = x_p - A^T (A A^T)^{-1} A x_p \quad (3.41)$$

$$\left(x_p^T u_h \right) u_h = (I - A^T (A A^T)^{-1} A) x_p = (I - A^+ A) x_p. \quad (3.42)$$

The matrix $(I - A^+ A)$ is the same null space projection matrix of equation (2.33). The Dubey-Euler-Babcock method and the null space projection matrix then both rely upon the equation

$$\underline{y} = \left(\underline{y}^T \underline{u}_{\min} \right) \underline{u}_{\min} + \left(\underline{y}^T \underline{u}_h \right) \underline{u}_h \quad (3.43)$$

where $\underline{y}$ is any suitable vector. If $\underline{y}$ is any particular solution to $\underline{y} = A \underline{x}$, as obtained by DEBM, then (3.43) can be arranged to give the least Euclidean norm solution for $\underline{y} = A \underline{x}$. Doing so has been called the Dubey-Euler-Babcock Method (DEBM) for determining least-norm solutions. On the other hand, if $\underline{y}$ is an arbitrary vector such as the gradient ∇H of an optimization criterion H , then (3.43) can be used to obtain the null space projection of that vector.

(4) Note that the Dubey-Euler-Babcock method must invert the nonsingular square matrix J^* . This means that if the manipulator is in a singular configuration, DEBM would no longer be applicable even if a feasible velocity vector were specified. However, if the generalized inverse were used the best approximate solution would be obtained even if the augmented matrix of the Jacobian were inconsistent. This shows that the Dubey-Euler-Babcock method is not generalized.

(5) Equation (3.40) shows that the matrix which is more closely associated with DEBM is the row space projection matrix $A^T (A A^T)^{-1} A$ of the matrix A and not the generalized inverse $A^T (A A^T)^{-1}$. That is, one may find a particular solution, as in DEBM, and then use $A^T (A A^T)^{-1} A$ to project it onto the row space of A , giving the minimum solution.

3.5. Dubey-Euler-Babcock Approach for the Gradient Projection Method

The relationship expressed by (3.34) is important because now the Dubey-Euler-Babcock method described above can be manipulated in equations that contain the Moore-Penrose pseudoinverse as long as $A^\oplus$ stays next to its associated vector and the necessary conditions relating to the matrix A are not violated.

With the understanding that it is a seven-degree-of-freedom manipulator being discussed and that the manipulator Jacobian J is of full rank, then (3.1) can be rewritten as

$$\dot{\underline{x}} = J^\oplus \dot{\underline{x}} + k(I - J^\oplus J) \nabla H, \quad (3.44)$$

which is the Dubey-Euler-Babcock approach to the gradient projection method (DEBA). In practical applications a slightly more useful rendition of DEBA is

$$\dot{\underline{x}} = J^\oplus \dot{\underline{x}} + k(\nabla H - J^\oplus J \nabla H). \quad (3.45)$$

3.6. DEBA, Spherical Wrist, and Real Time Motion

When the joint axes of the last three joints of a manipulator intersect at a point, the manipulator is said to have a spherical wrist. Pieper [9] showed that a spherical wrist is a sufficient condition for the solvability of the kinematic equations of a six-degree-of-freedom manipulator. This stems from the fact that when a manipulator has a spherical wrist the last three columns of the first three rows of the (wrist) Jacobian are full of zeros. The inverse kinematics problem is reduced from solving six simultaneous equations in six unknowns to solving two smaller simultaneous equations in three unknowns each, resulting in closed-form solutions for the joint velocities.

A spherical wrist can lead to closed-form solutions for a seven-degree-of-freedom manipulator too. Dubey et al [7] successfully implemented their methods on the seven-degree-of-freedom CESAR manipulator developed at Oak Ridge National Laboratory using closed-form solutions.

When the CESAR manipulator is controlled using the Dubey-Euler-Babcock Approach it operates in real-time [7]. Whether or not any particular algorithm can be applied in real-time mode depends on the math processors used. DEBA was applied using the Motorola MVME133 model computer system which contains both the 16.67 MHz MC68020 microprocessor and the MC68881 math coprocessor.

3.7. Null Space Projection Function

Delving deeper into the Dubey-Euler-Babcock method gives us an even more efficient computational scheme shown below. We can define a vector function $\mathcal{P}(\underline{x})$ such that

$$\mathcal{P}(\underline{x}) = \begin{pmatrix} \underline{x}^T \underline{x}_h \\ \underline{x}_h^T \underline{x}_h \end{pmatrix} \underline{x}_h \quad (3.46)$$

where $\underline{x}$ is any $n \times 1$ vector. DEBM now has the following form:

$$\mathbf{A}^\oplus \underline{y} = \underline{x}_{\min} = \underline{x}_p - \mathcal{P}(\underline{x}_p). \quad (3.47)$$

Equations (3.42) and (3.46) can be used to write the following

$$\mathcal{P}(\underline{x}) = \begin{pmatrix} \underline{x}^T & \underline{x}_h^T \\ \underline{x}_h^T & \underline{x}_h^T \end{pmatrix} \underline{x}_h = (\underline{x}^T \underline{u}_h) \underline{u}_h = (I - A^T (A A^T)^{-1} A) \underline{x} = (I - A^+ A) \underline{x}. \quad (3.48)$$

Equation (3.48) shows that the vector function $\mathcal{P}(\underline{x})$ can be used to project any arbitrary $n \times 1$ vector $\underline{x}$ upon the null space of A since $(I - A^+ A)$ is the well known null space projection matrix. The vector function $\mathcal{P}(\underline{x})$ will be called the *null space projection function*. We now have two important relations in terms of actually computing (3.1):

$$A^{\oplus} \underline{y} = A^+ \underline{y} \quad (A^{\oplus} \neq A^+) \quad (3.35)$$

and

$$\mathcal{P}(\underline{x}) = (I - A^+ A) \underline{x}. \quad (3.49)$$

We may now write (3.1) in the following efficient way:

$$\dot{\underline{q}} = J^{\oplus} \dot{\underline{x}} + k \{ \mathcal{P}(\underline{\nabla H}) \}. \quad (3.50)$$

It is easy to see why (3.50) is more efficient than (3.45): The matrix multiplication $J \underline{\nabla H}$ does not have to be performed. Also, applying $J^{\oplus}$ to $J \underline{\nabla H}$ does not have to be done. A comparison of (3.50) to (3.45) in Chapter 5 shows that as a result of using (3.50) there is a 43% savings in multiplications and divisions, and a 53% savings in additions and subtractions. Equation (3.50) will be called the *refined DEBA* or *RDEBA*.

The refined Dubey-Euler-Babcock approach is, nevertheless, still the gradient projection method which, however, has been made a lot faster by applying both DEBM and equation (3.46) to equation (3.1).

3.8. Control Scheme Implementation for the LTM

The joint velocity vector necessary to move the end-effector at specified linear and angular velocities is given by (3.50). This equation is valid for infinitesimal angular and linear displacements during an infinitesimal time period. To implement (3.50) on the LTM we must deal with small angular and linear displacements in a small time period. This may be expressed by Eq. 3.51

$$\underline{\Delta\theta} = J^{\oplus} \underline{\Delta x} + k \{ \mathcal{P}(\underline{\nabla H}) \}, \quad (3.51)$$

where

$$\underline{\Delta\theta} = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \Delta\theta_3 \\ \Delta\theta_4 \\ \Delta\theta_5 \\ \Delta\theta_6 \\ \Delta\theta_7 \end{bmatrix}, \quad (3.52)$$

and

$$\underline{\Delta x} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta\theta_x \\ \Delta\theta_y \\ \Delta\theta_z \end{bmatrix}. \quad (3.53)$$

The $\Delta\theta_i$, $i = 1, 2, \dots, 7$, are small angular displacements in the manipulator's joint angles. The $\Delta\theta_j$, $j = x, y, z$, are small angular displacements about the axes of the world coordinate system which are specified to reorient the end-effector. And Δx , Δy , Δz are specified small linear displacements of the end-effector with respect to the world coordinate system. Though the angular displacements are not infinitesimal, they are sufficiently small to treat $\underline{\Delta\theta}$ and $\underline{\Delta x}$ as vectors.

The constant k must be scaled to stay within velocity bounds. Chapter 4 presents a method for automatically finding a value for k such that $\dot{Q}$ is as large as possible but still stays within joint velocity bounds [12]. But basing the determination of the LTM's self-motion on joint velocity bounds results in oscillation of the joint angles whenever an optimum configuration has been attained [14]. This oscillation is highly noticeable when the algorithm is run at low frequencies around 50 Hertz, but may even be noticeable at 100 Hertz. When an optimum configuration has been attained there is no null space available for use, for if there were the arm would not be at an optimum, therefore, the gradient $\underline{\nabla H}$

must be perpendicular to the null space. When close to an optimum the elements of the homogeneous vector given by $\mathcal{P}(\underline{\nabla H})$ are all very small. The scalar k then becomes very large in an effort to drive the joint velocities at their maximum possible values. This not only moves the configuration of the arm away from the optimum, but also results in numerical errors causing end-effector velocity errors. The joint angles, therefore, oscillate around their optimum values. To keep the scalar k from reaching large values, the following method is effective:

$$\Delta\theta = J^{\oplus}\Delta x + k |\cos \beta|^y \{ \mathcal{P}(\underline{\nabla H}) \}, \quad (3.54)$$

where

$$\cos \beta = \frac{\underline{\nabla H}^T \Delta\theta_h}{|\underline{\nabla H}| |\Delta\theta_h|} \quad (3.55)$$

and y is a fractional power which has been empirically determined to be 0.62. The angle β is the angle between $\Delta\theta_h$ and $\underline{\nabla H}$. When β is 90° the arm is in an optimum configuration and $\cos \beta = 0$. Thus the effect of the scalar k is diminished as the optimum is approached.

Figure 3-2 is a block diagram for implementation of the kinematic optimization scheme. Control loops are closed around the joint servos, and the optimizing inverse kinematics algorithm is implemented in an open-loop fashion.

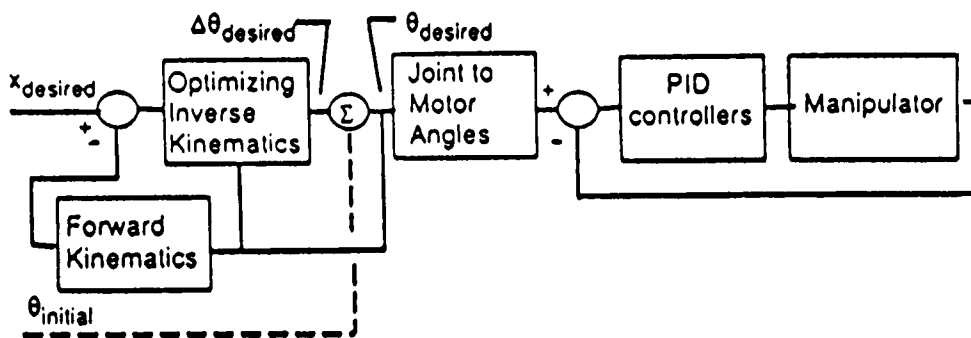


Figure 3-2: Real-time implementation block diagram [14]

CHAPTER 4

KINEMATICS OF THE LTM

4.1. Forward Kinematics of the LTM

The LTM is a seven-degree-of-freedom manipulator with a spherical wrist. The pitch-yaw-roll spherical wrist is designed so that singularities due to the wrist occur when the hand is pointing to its sides and not when it is pointing straight out, as is common in many industrial robots. Degrees of freedom of the LTM and the coordinate frames referred to in the Denavit-Hartenberg table of parameters (Table 1) for this manipulator are shown in Fig.4-1. From the parameters shown in Table 1 we can write what have historically been called the A matrices, which are the homogeneous transformation matrices which relate coordinate frame i to coordinate frame $i-1$. This relationship is denoted using subscripts and superscripts in the following manner: A_i^{i-1} . These matrices are listed in Appendix B.

What have historically been called the T matrices are those matrices which relate coordinate frames of the manipulator links to the base or world coordinate frame. The kinematic equation T for the LTM is given by the following matrix multiplication

$$T = A_1^0 A_2^1 A_3^2 A_4^3 A_5^4 A_6^5 A_7^6 . \quad (4.1)$$

The results of this multiplication is given in Appendix C. The forward kinematics subroutine written for the LTM and discussed in Chapter 5 makes use of the fact that calculations are simpler when referred to the third coordinate frame [10]. In order to obtain kinematic equations referred to the third coordinate frame we can use the following equation

$${}^3T = \begin{bmatrix} {}^3R_0 & 0^{3 \times 1} \\ 0^{1 \times 3} & 1 \end{bmatrix} T , \quad (4.2)$$

where T is the kinematic equation given by (4.1), and ${}^3T$ is that matrix referred to the third coordinate frame. ${}^3R_0$ is a 3×3 projection matrix given by:

Table 1. Denavit-Hartenberg table of link parameters

link number	d_i (m)	α_i (deg)	a_i (m)	θ_i (deg)
1	0	-90	0	θ_1
2	0	90	a_2	θ_2
3	0	90	0	θ_3
4	0	-90	a_4	θ_4
5	0	90	0	θ_5
6	0	90	0	θ_6
7	d_7	0	0	θ_7

Note: $a_2 = 23.0$ in., $a_4 = 20.0$ in., $d_7 = 12.0$ in.

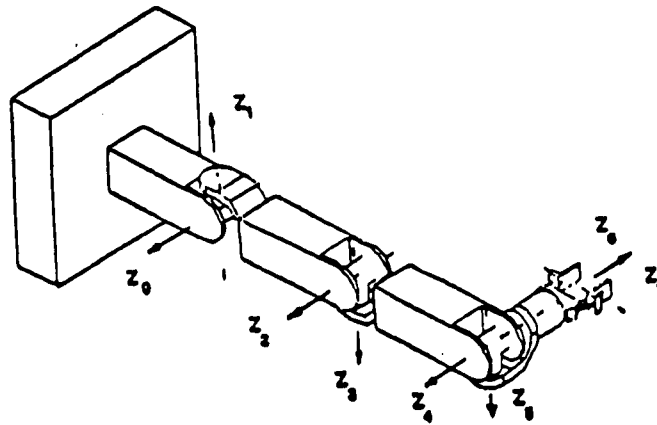


Figure 4-1: Reference configuration for kinematics of the LTM.

$${}^3R_0 = \begin{bmatrix} c_1c_2c_3-s_1s_3 & s_1c_2c_3+c_1s_3 & -s_2c_3 \\ c_1s_2 & s_1s_2 & c_2 \\ c_1c_2s_3+s_1c_3 & s_1c_2s_3-c_1c_3 & -s_2s_3 \end{bmatrix}, \quad (4.3)$$

where for equation (4.3) and *the rest of this paper* the c_i and s_i represent $\cos\theta_i$ and $\sin\theta_i$ respectively. The matrix ${}^3R_0$ is a rotation matrix which serves to change the coordinate representation of a vector from base coordinates to third coordinate frame coordinates.

When the operations indicated by (4.2) are performed we have

$${}^3T = \begin{bmatrix} c_7(c_4c_5c_6-s_4s_6)+c_4s_5s_7 & -s_7(c_4c_5c_6-s_4s_6)+c_4s_5c_7 & s_4c_6+c_4c_5s_6 & a_4c_4+a_2c_3 \\ c_7(s_4c_5c_6+c_4s_6)+s_4s_5s_7 & -s_7(s_4c_5c_6+c_4s_6)+s_4s_5c_7 & s_4c_5s_6-c_4c_6 & a_4s_4 \\ c_5s_7-s_5c_6c_7 & c_5c_7+s_5c_6s_7 & -s_5s_6 & a_2s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad -(4.4)$$

Note that the origin of the 7th coordinate frame is located at the wrist, coincident with the origin for the 6th coordinate frame. The last row of (4.4) actually gives the position of the wrist referred to the 3rd coordinate frame. To locate the hand (or a tool grasped by the hand) we must go a distance, say distance d_7 of Table 1, along the z-axis of the 7th coordinate frame. The position of the hand then becomes

$$\begin{bmatrix} h_x \\ h_y \\ h_z \end{bmatrix} = \begin{bmatrix} a_4c_4+a_2c_3 \\ a_4s_4 \\ a_2s_3 \end{bmatrix} + d_7 \begin{bmatrix} s_4c_6+c_4c_5s_6 \\ s_4c_5s_6-c_4c_6 \\ -s_5s_6 \end{bmatrix} \quad (4.5)$$

Equation (4.4) with the elements of (4.5) used in the last column of (4.4), gives the forward kinematics of the LTM referred to the 3rd coordinate frame.

4.2. Inverse Kinematics of the LTM

If the relationship between a 7×1 unknown joint velocity vector $\dot{\mathbf{Q}}$ and a 6×1 specified end-effector velocity vector $\dot{\mathbf{x}}$ is expressed using the equation

$$\dot{\mathbf{x}} = \mathbf{J} \dot{\mathbf{Q}} \quad (4.6)$$

where J is a 6×7 manipulator Jacobian, then solutions to (4.6) that allow optimization of the configuration of the manipulator while moving the end-effector with arbitrary linear and angular velocities can be obtained using:

$$\dot{\underline{Q}} = J^{\oplus} \dot{\underline{X}} + k \{ \mathcal{P}(\nabla H) \}. \quad (4.7)$$

where $J^{\oplus}$ indicates specific operations discussed in Chapter 3 involving the Jacobian J and the vector next to it. The Jacobian must be a 6×7 matrix of rank 6. The vector function $\mathcal{P}(\nabla H)$ is defined by equation (3.46). The symbol ∇H is the gradient of a scalar function H used to optimize the configuration of the arm. The optimization criterion H is a function of joint coordinates. The scalar constant k is chosen positive for maximization of H and negative for minimization of H . It is also used to scale the homogeneous part of the joint velocity vector so that the commanded joint velocity vector will not exceed velocity limits determined by hardware of the LTM. The gradient projection method is covered thoroughly in Chapter 2.

4.2.1. The Midframe Jacobian

In order to simplify the calculations we will refer the desired end-effector and wrist velocity vectors to the third coordinate frame $\underline{x}_3, \underline{y}_3, \underline{z}_3$. This results in a Jacobian that has a much simpler form and, thus, is more efficient for computation [10]. Let the desired end-effector velocity referred to the third coordinate frame be given by

$${}^3\dot{\underline{X}}_h = [{}^3\underline{V}_h^T \quad {}^3\underline{\omega}_h^T]^T = [{}^3\dot{\underline{x}}_{h1} \quad {}^3\dot{\underline{x}}_{h2} \quad {}^3\dot{\underline{x}}_{h3} \quad {}^3\dot{\underline{x}}_{h4} \quad {}^3\dot{\underline{x}}_{h5} \quad {}^3\dot{\underline{x}}_{h6}]^T, \quad (4.8)$$

where ${}^3\underline{V}_h \in \mathbb{R}^3$ and ${}^3\underline{\omega}_h \in \mathbb{R}^3$ are the linear and angular hand velocity vectors, respectively, referred to the third coordinate frame. These terms can be obtained by referring the specified end-effector velocity vector to the third coordinate frame using the following equation ${}^3\dot{\underline{X}}_h = {}^3R_0 \dot{\underline{X}}_h$, where $\dot{\underline{X}}_h$ is the hand (end-effector) velocity vector referred to the base coordinates and ${}^3R_0$ is a 3×3 projection matrix given by (4.3). Let the desired wrist velocity vector be given by

$${}^3\dot{\underline{X}}_w = [{}^3\underline{V}_w^T \quad {}^3\underline{\omega}_w^T]^T = [{}^3\dot{\underline{x}}_{w1} \quad {}^3\dot{\underline{x}}_{w2} \quad {}^3\dot{\underline{x}}_{w3} \quad {}^3\dot{\underline{x}}_{w4} \quad {}^3\dot{\underline{x}}_{w5} \quad {}^3\dot{\underline{x}}_{w6}]^T, \quad (4.9)$$

where ${}^3\mathbf{y}_w \in R^3$ and ${}^3\mathbf{\omega}_w \in R^3$ are the linear and angular wrist velocity vectors, respectively, referred to the third coordinate frame. For a given ${}^3\dot{\mathbf{x}}_h$, the terms of ${}^3\dot{\mathbf{x}}_w$ may be obtained by using the following relationships:

$${}^3\mathbf{\omega}_w = {}^3\mathbf{\omega}_h, \quad (4.10)$$

$${}^3\mathbf{y}_w = {}^3\mathbf{y}_h - {}^3\mathbf{\omega}_h \times d_7 {}^3\mathbf{z}_h, \quad (4.11)$$

where ${}^3\mathbf{z}_h$ is the unit vector $\mathbf{z}_h$ at the end-effector (see Fig. 4-1) referred to the third coordinate frame; it may be shown to be the following (see forward kinematics):

$${}^3\mathbf{z}_h = [s_4c_6 + c_4c_5s_6 \quad s_4c_5s_6 - c_4c_6 \quad -s_5s_6]^T. \quad (4.12)$$

Let ${}^3J_w$ be the midframe wrist Jacobian relating the joint velocity vector $\dot{\mathbf{q}} = [\dot{\theta}_1, \dot{\theta}_2, \dots, \dot{\theta}_7]^T$ and the wrist velocity vector ${}^3\dot{\mathbf{x}}_w$ such that

$${}^3\dot{\mathbf{x}}_w = {}^3J_w \dot{\mathbf{q}}. \quad (4.13)$$

The Jacobian ${}^3J_w$ can be shown to be the following (see Appendix E):

$${}^3J_w = \begin{bmatrix} a_4s_2s_3s_4 + a_2c_2s_3 & a_4c_3s_4 & 0 & -a_4s_4 & 0 & 0 & 0 \\ -a_4s_2s_3c_4 & -a_4c_3c_4 - a_2 & 0 & a_4c_4 & 0 & 0 & 0 \\ -a_4(s_2c_3s_4 + c_2c_4) - a_2c_2c_3 & a_4s_3s_4 & -a_4c_4 & 0 & 0 & 0 & 0 \\ -s_2c_3 & s_3 & 0 & 0 & -s_4 & c_4s_5 & c_4c_5s_6 + s_4c_6 \\ c_2 & 0 & 1 & 0 & c_4 & s_4s_5 & s_4c_5s_6 - c_4c_6 \\ -s_2s_3 & -c_3 & 0 & 1 & 0 & c_5 & -s_5s_6 \end{bmatrix} \quad -(4.14)$$

Equation (4.7) now becomes

$$\dot{\mathbf{q}} = {}^3J_w^\oplus {}^3\dot{\mathbf{x}}_w + k \{ \mathcal{P}(\nabla H) \}. \quad (4.15)$$

Equation (4.15) describes exactly what is done in the inverse kinematics subroutine written for the LTM.

4.2.2. Least Euclidean Norm Solution

The least Euclidean norm is obtained by choosing the arbitrary scalar constant κ such that it minimizes the following equation:

$$\|\dot{\underline{q}}\|^2 = \|\dot{\underline{q}}_p + \kappa \dot{\underline{q}}_h\|^2 \quad (4.16)$$

As shown by Dubey et al.[8], if κ is chosen to be $-(\dot{\underline{q}}_p^T \dot{\underline{q}}_h / \dot{\underline{q}}_h^T \dot{\underline{q}}_h)$, then (4.16) will result in the least Euclidean norm solution with respect to ${}^3\dot{\underline{x}}_w = {}^3\mathbf{J}_w \dot{\underline{q}}$.

$$\dot{\underline{q}}_{\min} = \dot{\underline{q}}_p - \left(\frac{\dot{\underline{q}}_p^T \dot{\underline{q}}_h}{\dot{\underline{q}}_h^T \dot{\underline{q}}_h} \right) \dot{\underline{q}}_h \quad (4.17)$$

Alternately we may define a vector function $\mathcal{P}(\underline{x})$ such that

$$\mathcal{P}(\underline{x}) = \left(\frac{\underline{x}^T \dot{\underline{q}}_h}{\dot{\underline{q}}_h^T \dot{\underline{q}}_h} \right) \dot{\underline{q}}_h, \quad (4.18)$$

in which case (4.17) becomes

$$\dot{\underline{q}}_{\min} = \dot{\underline{q}}_p - \mathcal{P}(\dot{\underline{q}}_p). \quad (4.19)$$

When the Dubey-Euler-Babcock method is applied to vector ${}^3\dot{\underline{x}}_w$ we write ${}^3\mathbf{J}_w^\oplus {}^3\dot{\underline{x}}_w$ which means we must first find

$$\dot{\underline{q}}_p = \begin{bmatrix} 0 \\ {}^3\mathbf{J}_w^{*-1} {}^3\dot{\underline{x}}_w \end{bmatrix}, \text{ and} \quad (4.20)$$

$$\dot{\underline{\theta}}_h = \begin{bmatrix} 1 \\ -{}^3J_w^{*-1} \underline{\alpha} \end{bmatrix}, \quad (4.21)$$

where $\dot{\underline{\theta}}_p$ and $\dot{\underline{\theta}}_h$ are arbitrary particular and homogeneous solutions to (4.6), ${}^3J_w^*$ is a 6×6 nonsingular matrix formed from 6 linearly independent columns of ${}^3J_w$, and $\underline{\alpha}$ is the remaining column of ${}^3J_w$ and is linearly dependent on the columns of ${}^3J_w^*$. The matrix ${}^3J_w^*$ and the column $\underline{\alpha}$ are arranged in the order shown by the following equation:

$${}^3\dot{\underline{x}}_w = [\underline{\alpha} \ {}^3J_w^*] \dot{\underline{\theta}}. \quad (4.22)$$

It is important to note that $\dot{\underline{\theta}}_p$ and $\dot{\underline{\theta}}_h$ are only temporary variables, they are not the particular and homogeneous parts of $\dot{\underline{\theta}}$ for (4.15). We can partition ${}^3J_w^*$ as follows:

$${}^3J_w^* = \begin{bmatrix} {}^3J_{w1}^{* \ 3 \times 3} & 0 \ 3 \times 3 \\ {}^3J_{w2}^{* \ 3 \times 3} & {}^3J_{w3}^{* \ 3 \times 3} \end{bmatrix}, \quad (4.23)$$

To determine ${}^3J_w^{*-1} {}^3\dot{\underline{x}}_w$ and ${}^3J_w^{*-1} \underline{\alpha}$ we only need to solve two sets of three simultaneous equations.

4.2.2.1. Selection of $\underline{\alpha}$

Ideally we would like to select one particular dependent column $\underline{\alpha}$ such that (4.20) and (4.21) could always be calculated for that particular $\underline{\alpha}$ except at singular configurations of the manipulator. In actuality such a choice is not always possible due to the presence of algorithmic singularities even though the manipulator is not in a singular configuration.

The determination of $\underline{\alpha}$ is done by solving (4.20) and (4.21) using each of the first four columns of ${}^3J_w$ in turn and making selections based on the fewest number of algorithmic singularities. When that was done for the LTM it was found that for each of the first four columns $\sin(\theta_6)$ always appears as a denominator, therefore, the manipulator Jacobian becomes singular whenever $\theta_6 = 0^\circ$ or $\pm 180^\circ$, which are not by themselves singular configurations for the LTM. In addition to $\sin(\theta_6)$ appearing as a denominator whenever any of the first four columns are chosen for $\underline{\alpha}$, when column 1 is chosen $\sin(\theta_4)$ and

$\cos(\theta_4)$ appear as denominators; when column 2 is chosen $\cos(\theta_2)$, $\sin(\theta_3)$, and $\cos(\theta_4)$ appear as denominators; when column 3 is chosen $\cos(\theta_3)$ and $\sin(\theta_4)$ appear as denominators; and when column 4 is chosen $\sin(\theta_3)$ and $\cos(\theta_4)$ appear as denominators.

A scheme which alternates between choosing columns 1, 3, and 4 for $\underline{\alpha}$ was decided upon. This scheme can be stated as follows: If $\cos(\theta_4)$ is zero, column 3 is chosen for $\underline{\alpha}$; if $\sin(\theta_4)$ is zero, column 4 is chosen for $\underline{\alpha}$; otherwise column 1 is chosen for $\underline{\alpha}$. Whenever $\underline{\alpha}$ is chosen in the manner just stated algorithmic singularities always occur when the manipulator is in an actual singular configuration. The LTM's redundancy can be utilized to avoid such singular configurations by improving appropriate performance criteria. The scheme for choosing $\underline{\alpha}$ is illustrated in Fig. 4-2.

4.2.2.2. When $\underline{\alpha}$ is the First Column

The elements of $\dot{\underline{\theta}}_p$ denoted by $\dot{\theta}_{p_i}$, for $i = 1$ to 7, may be obtained as follows:

$$\dot{\theta}_{p_1} = 0 \quad (4.24)$$

$$\begin{bmatrix} \dot{\theta}_{p_2} \\ \dot{\theta}_{p_4} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} a_4 c_4 & a_4 s_4 \\ a_2 + a_4 c_3 c_4 & a_4 c_3 s_4 \end{bmatrix} \begin{bmatrix} {}^3\dot{x}_{w_1} \\ {}^3\dot{x}_{w_2} \end{bmatrix}, \quad (4.25)$$

where $\Delta = -a_2 a_4 s_4$,

$$\dot{\theta}_{p_3} = \frac{(a_4 s_3 s_4) \dot{\theta}_{p_2} - {}^3\dot{x}_{w_3}}{a_4 c_4}, \quad (4.26)$$

$$\begin{bmatrix} \dot{\theta}_{p_5} \\ \dot{\theta}_{p_6} \\ \dot{\theta}_{p_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} {}^3\dot{x}_{w_4} - s_3 \dot{\theta}_{p_2} \\ {}^3\dot{x}_{w_5} - \dot{\theta}_{p_3} \\ {}^3\dot{x}_{w_6} + c_3 \dot{\theta}_{p_2} - \dot{\theta}_{p_4} \end{bmatrix}. \quad (4.27)$$

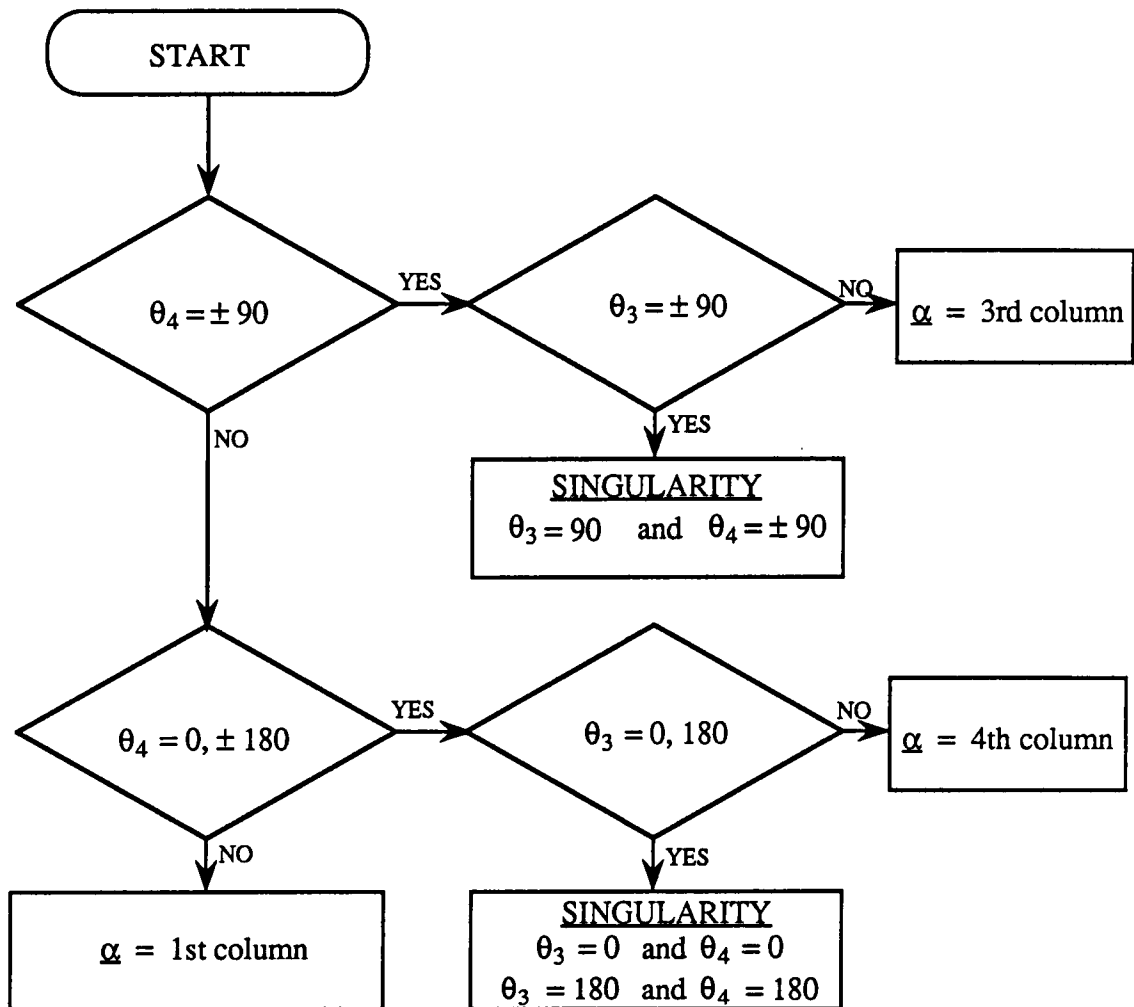


Figure 4-2 : Flowchart showing how α is chosen

The vector $\dot{\mathbf{q}}_h$ with elements $\dot{\theta}_{h_i}$, $i = 1$ to 7 may similarly be realized by setting the element $\dot{\theta}_{h_1} = 1$, and obtaining the remaining elements of $\dot{\mathbf{q}}_h$ from (4.25) through (4.27) by replacing $\dot{\theta}_{p_i}$ with $\dot{\theta}_{h_i}$ and using the i^{th} elements of vector $-\underline{\alpha}$ for ${}^3\dot{\mathbf{x}}_{w_i}$, $i = 1$ to 6. These equations are included for completeness:

$$\dot{\theta}_{h_1} = 1 \quad (4.28)$$

$$\begin{bmatrix} \dot{\theta}_{h_2} \\ \dot{\theta}_{h_4} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} a_4 c_4 & a_4 s_4 \\ a_2 + a_4 c_3 c_4 & a_4 c_3 s_4 \end{bmatrix} \begin{bmatrix} -(a_4 s_2 s_3 s_4 + a_2 c_2 s_3) \\ a_4 s_2 s_3 c_4 \end{bmatrix}, \quad (4.29)$$

where $\Delta = -a_2 a_4 s_4$,

$$\dot{\theta}_{h_3} = \frac{(a_4 s_3 s_4) \dot{\theta}_{h_2} - a_4 (s_2 c_3 s_4 + c_2 c_4) - a_2 c_2 c_3}{a_4 c_4}, \quad (4.30)$$

$$\begin{bmatrix} \dot{\theta}_{h_5} \\ \dot{\theta}_{h_6} \\ \dot{\theta}_{h_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} s_2 c_3 - s_3 \dot{\theta}_{h_2} \\ -c_2 - \dot{\theta}_{h_3} \\ s_2 s_3 + c_3 \dot{\theta}_{h_2} - \dot{\theta}_{h_4} \end{bmatrix}. \quad (4.31)$$

4.2.2.3. When $\underline{\alpha}$ is the Third Column

On the other hand, another column, say column three, may be taken to be $\underline{\alpha}$ with the remaining six columns forming the nonsingular matrix ${}^3\mathbf{J}_w^*$, in which case:

$$\dot{\theta}_{p_3} = 0, \quad (4.32)$$

$$\begin{bmatrix} \dot{\theta}_{p_1} \\ \dot{\theta}_{p_2} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} a_4 s_3 s_4 & a_4 s_4 \\ a_4 (s_2 c_3 s_4 + c_2 c_4) + a_2 c_2 c_3 & a_4 c_2 s_3 c_4 \end{bmatrix} \begin{bmatrix} c_4 {}^3\dot{x}_{w_1} + s_4 {}^3\dot{x}_{w_2} \\ {}^3\dot{x}_{w_3} \end{bmatrix}, \quad (4.33)$$

where $\Delta = -a_4 c_3 s_4 (a_4 c_2 c_3 c_4 + a_4 s_2 s_4 + a_2 c_2)$,

$$\dot{\theta}_{p_4} = \frac{s_4 {}^3\dot{x}_{w_1} - c_4 {}^3\dot{x}_{w_2} - (a_4 s_2 s_3 + a_2 c_2 s_3 s_4) \dot{\theta}_{p_1} - (a_4 c_3 + a_2 c_4) \dot{\theta}_{p_2}}{-a_4}, \quad (4.34)$$

$$\begin{bmatrix} \dot{\theta}_{p_5} \\ \dot{\theta}_{p_6} \\ \dot{\theta}_{p_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} {}^3\dot{x}_{w_4} + s_2 c_3 \dot{\theta}_{p_1} - s_3 \dot{\theta}_{p_2} \\ {}^3\dot{x}_{w_5} - c_2 \dot{\theta}_{p_1} \\ {}^3\dot{x}_{w_6} + s_2 s_3 \dot{\theta}_{p_1} + c_3 \dot{\theta}_{p_2} - \dot{\theta}_{p_4} \end{bmatrix}, \quad (4.35)$$

and similarly $\dot{\theta}_h$ may be found

$$\dot{\theta}_{h_3} = 1 \quad (4.36)$$

$$\begin{bmatrix} \dot{\theta}_{h_1} \\ \dot{\theta}_{h_2} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} a_4 s_3 s_4 & a_4 s_4 \\ a_4 (s_2 c_3 s_4 + c_2 c_4) + a_2 c_2 c_3 & a_4 c_2 s_3 c_4 \end{bmatrix} \begin{bmatrix} 0 \\ a_4 c_4 \end{bmatrix}, \quad (4.37)$$

where $\Delta = -a_4 c_3 s_4 (a_4 c_2 c_3 c_4 + a_4 s_2 s_4 + a_2 c_2)$,

$$\dot{\theta}_{p_4} = \frac{-(a_4 s_2 s_3 + a_2 c_2 s_3 s_4) \dot{\theta}_{h_1} - (a_4 c_3 + a_2 c_4) \dot{\theta}_{h_2}}{-a_4}, \quad (4.38)$$

$$\begin{bmatrix} \dot{\theta}_{h_5} \\ \dot{\theta}_{h_6} \\ \dot{\theta}_{h_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} s_2 c_3 \dot{\theta}_{h_1} - s_3 \dot{\theta}_{h_2} \\ -c_2 \dot{\theta}_{h_1} - \dot{\theta}_{h_3} \\ s_2 s_3 \dot{\theta}_{h_1} + c_3 \dot{\theta}_{h_2} - \dot{\theta}_{h_4} \end{bmatrix}. \quad (4.39)$$

4.2.2.4. When α is the Fourth Column

Column four, may also be taken to be α with the remaining six columns forming the nonsingular matrix ${}^3J_w^*$, in which case:

$$\dot{\theta}_{p_4} = 0, \quad (4.40)$$

$$\begin{bmatrix} \dot{\theta}_{p_1} \\ \dot{\theta}_{p_2} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} -(a_2 + a_4 c_3 c_4) & -a_4 c_3 s_4 \\ a_4 s_2 s_3 c_4 & a_4 s_2 s_3 s_4 + a_2 c_2 s_3 \end{bmatrix} \begin{bmatrix} {}^3\dot{x}_{w_1} \\ {}^3\dot{x}_{w_2} \end{bmatrix}, \quad (4.41)$$

where $\Delta = -a_2 s_3 (a_4 c_2 c_3 c_4 + a_4 s_2 s_4 + a_2 c_2)$,

$$\dot{\theta}_{p_3} = \frac{(a_4 s_3 s_4) \dot{\theta}_{p_2} - {}^3\dot{x}_{w_3} - [a_4 (s_2 c_3 s_4 + c_2 c_4) + a_2 c_2 c_3] \dot{\theta}_{p_1}}{a_4 c_4}, \quad (4.42)$$

$$\begin{bmatrix} \dot{\theta}_{p_5} \\ \dot{\theta}_{p_6} \\ \dot{\theta}_{p_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} {}^3\dot{x}_{w_4} + s_2 c_3 \dot{\theta}_{p_1} - s_3 \dot{\theta}_{p_2} \\ {}^3\dot{x}_{w_5} - c_2 \dot{\theta}_{p_1} - \dot{\theta}_{p_3} \\ {}^3\dot{x}_{w_6} + s_2 s_3 \dot{\theta}_{p_1} + c_3 \dot{\theta}_{p_2} \end{bmatrix}. \quad (4.43)$$

and similarly $\dot{\theta}_h$ may be found

$$\dot{\theta}_{h_4} = 1 \quad (4.44)$$

$$\begin{bmatrix} \dot{\theta}_{h_1} \\ \dot{\theta}_{h_2} \end{bmatrix} = \frac{1}{\Delta} \begin{bmatrix} -(a_2 + a_4 c_3 c_4) & -a_4 c_3 s_4 \\ a_4 s_2 s_3 c_4 & a_4 s_2 s_3 s_4 + a_2 c_2 s_3 \end{bmatrix} \begin{bmatrix} a_4 s_4 \\ -a_4 c_4 \end{bmatrix}, \quad (4.45)$$

where $\Delta = -a_2 s_3 (a_4 c_2 c_3 c_4 + a_4 s_2 s_4 + a_2 c_2)$,

$$\dot{\theta}_{h_3} = \frac{(a_4 s_3 s_4) \dot{\theta}_{h_2} - [a_4 (s_2 c_3 s_4 + c_2 c_4) + a_2 c_2 c_3] \dot{\theta}_{h_1}}{a_4 c_4}, \quad (4.46)$$

$$\begin{bmatrix} \dot{\theta}_{h_5} \\ \dot{\theta}_{h_6} \\ \dot{\theta}_{h_7} \end{bmatrix} = \frac{1}{s_6} \begin{bmatrix} c_4 c_5 c_6 - s_4 s_6 & s_4 c_5 c_6 + c_4 s_6 & -s_5 c_6 \\ c_4 s_5 s_6 & s_4 s_5 s_6 & c_5 s_6 \\ c_4 c_5 & s_4 c_5 & -s_5 \end{bmatrix} \begin{bmatrix} s_2 c_3 \dot{\theta}_{h_1} - s_3 \dot{\theta}_{h_2} \\ -c_2 \dot{\theta}_{h_1} - \dot{\theta}_{h_3} \\ s_2 s_3 \dot{\theta}_{h_1} + c_3 \dot{\theta}_{h_2} - \dot{\theta}_{h_4} \end{bmatrix}. \quad (4.47)$$

4.2.2.4. Wrist Singularity Case

When joint 6 is either 0 or 180°, then joint 5 is collinear with joint 7. In this case the column 5 vector of ${}^3J_w$ is equal to or the negative of the column 7 vector, which implies that the wrist is singular and cannot be moved to an arbitrary new orientation using just the wrist joint angles. In this case the first six columns of ${}^3J_w$ are independent. If it is desired to operate the LTM with a singular wrist DEBM can still be used, but to determine ${}^3J_w^{*-1} \dot{x}_w$ and ${}^3J_w^{*-1} \alpha$ we must actually invert ${}^3J_w^{*-1}$ or use some other method of solution such as Gaussian elimination. We may avoid the condition of joint 6 being either 0 or 180° by improving a suitable performance criteria.

4.2.3. Homogeneous Solution

The homogeneous solution is found as described in Section 3.8 where we pass ∇H to the null space projection function $\mathcal{P}(\nabla H)$ defined by equation (3.46). The scalar

constant k is determined based on joint velocity bounds, and is reduced in value whenever the configuration of the LTM is close to an optimum. The following two sections briefly discuss optimization criteria and the algorithm used to determine the value of the scalar constant k .

4.2.3.1. Projection of Optimization Criterion Gradient onto the Null Space

Performance criteria for the LTM may be chosen for a variety of reasons such as avoiding obstacles, improving mechanical advantage, avoiding joints limits, and avoiding singular configurations. Suitable performance criteria is currently a subject of research. To judge the functionality of the control software simple performance criteria were used based on the joint limits and singular configurations of the LTM. Tables for the singularities [11] and joints limits of the LTM are in the appendix. To stay within the joint limits of the LTM we may choose the following performance criterion:

$$H(\theta) = \sum k_1 (\theta_i - \theta_{i_{mid}})^2, \quad (4.48)$$

where the k_1 are weighting factors that may be chosen based on the range of the i th joint angle and the distance the i th joint angle is from its desired value, θ_i is the i th joint angle and $\theta_{i_{mid}}$ is the midpoint value for joint angle θ_i . To avoid singular configurations of the LTM we may improve the following performance criterion:

$$H(\theta) = k_2 \sin^2 \theta_2 + k_3 \cos^2 \theta_3 + k_4 \sin^2 \theta_4 + k_5 \cos^2 \theta_5, \quad (4.49)$$

where the k_1 are weighting factors that may be chosen based on the range of the joint angle and the distance the joint angle is from its desired value. The gradient of H , ∇H , is obtained in the following manner:

$$\nabla H = [\partial H / \partial \theta_1 \quad \partial H / \partial \theta_2 \quad \partial H / \partial \theta_3 \quad \partial H / \partial \theta_4 \quad \partial H / \partial \theta_5 \quad \partial H / \partial \theta_6 \quad \partial H / \partial \theta_7]^T. \quad (4.50)$$

To project the gradient onto the null space of the Jacobian we pass ∇H to the null space projection function $\mathcal{P}(\nabla H)$ defined by equation (3.46).

4.2.3.2. Choosing the Scalar Constant k

We would like to choose k so that hardware limits on the velocities are not exceeded. The method used is that proposed by Euler et al. [12] where it is shown that the gradient projection method can be written

$$\dot{\underline{\theta}} = \dot{\underline{\theta}}_{\min} + k\dot{\underline{\theta}}_{\text{null}} \quad (4.51)$$

where $\dot{\underline{\theta}}_{\min}$ is the minimum Euclidean norm solution to ${}^3\dot{\underline{x}}_w = {}^3J_w\dot{\underline{\theta}}$, and $\dot{\underline{\theta}}_{\text{null}}$ is the orthogonal projection of ∇H upon the null space of ${}^3J_w$. Because we do not want to exceed $\pm \dot{\underline{\theta}}_{\text{MAX}}$, the bounds on the joint velocities, we can substitute $\pm \dot{\underline{\theta}}_{\text{MAX}}$ for $\dot{\underline{\theta}}$ in equation (4.51),

$$\pm \dot{\underline{\theta}}_{\text{MAX}} = \dot{\underline{\theta}}_{\min} + k\dot{\underline{\theta}}_{\text{null}}. \quad (4.52)$$

Using (4.52) we may solve for the upper and lower bounds to k for the i th joint in the following manner:

$$k_{\max_i} = \max \left(\frac{\dot{\theta}_{\text{MAX}_i} - \dot{\theta}_{\min_i}}{\dot{\theta}_{\text{null}_i}}, \frac{-\dot{\theta}_{\text{MAX}_i} - \dot{\theta}_{\min_i}}{\dot{\theta}_{\text{null}_i}} \right) \quad (4.53)$$

and

$$k_{\min_i} = \min \left(\frac{\dot{\theta}_{\text{MAX}_i} - \dot{\theta}_{\min_i}}{\dot{\theta}_{\text{null}_i}}, \frac{-\dot{\theta}_{\text{MAX}_i} - \dot{\theta}_{\min_i}}{\dot{\theta}_{\text{null}_i}} \right) \quad (4.54)$$

where the subscript i denotes the i th component in the associated vector. The object is to find the maximum k and minimum k for each joint, and then choose the smallest maximum or the largest minimum for the scalar constant k :

$$k_{\max} = \min (k_{\max_1}, k_{\max_2}, k_{\max_3}, k_{\max_4}, k_{\max_5}, k_{\max_6}, k_{\max_7}) \quad (4.55)$$

and

$$k_{\min} = \max (k_{\min_1}, k_{\min_2}, k_{\min_3}, k_{\min_4}, k_{\min_5}, k_{\min_6}, k_{\min_7}) \quad (4.56)$$

If k is chosen to be $k_{\max}$, then the optimization criterion H will be maximized; if it is chosen to be $k_{\min}$, then it will be minimized. Choosing k in this manner for each iteration of the positioning routine will assure that requests which would exceed hardware limits on the joint velocities are never made.

Whenever it is possible for the manipulator to change its configuration towards the optimum configuration $k_{\max}$ will be positive and $k_{\min}$ will be negative. However, due to the specified trajectory $k_{\max}$ and $k_{\min}$ will sometimes have the same signs. This simply means that the manipulator must move its configuration away from optimum in order to stay on its assigned trajectory; it will be, nevertheless, optimizing the configuration in the best way it can as it is forced to move away from optimum. In other words, the rate of non-optimization will be kept to a minimum.

If it should happen that $k_{\max} < k_{\min}$, then no scalar constant k is available to insure that velocity requests are attainable. In this case even the velocities requested by the least-norm solution ${}^3J_w^{\oplus 3} \dot{\underline{x}}_w$ are not attainable due to limits on the joint motors.

CHAPTER 5

COMPUTER PROGRAM

The robotic control software for the LTM was written in two parts: motion control of the manipulator in joint space, and motion control of the manipulator in Cartesian space. In Cartesian space the end-effector follows a specified trajectory in Cartesian coordinates. The joint motion required to follow a specified trajectory is obtained through inverse kinematics. In joint space motion control the end-effector is moved from one point to another through an interpolated joint trajectory.

To accomplish real-time motion of the LTM in joint space an improved version of the efficient gradient projection scheme developed by Dubey et al. [8] was utilized. This improved version uses the gradient projection function introduced in Chapter 3.

The C programming language was chosen to write the robotic control scheme software due to its speed and portability. A global data structure, which specifies program variables to be used in a common memory buffer (Appendix H), is used to minimize the difficulties associated with the integration of different program modules. It is easy to see from Fig. 5-1 how the global variable buffer facilitates unification of the program modules. All information passed to the control routines is obtained from the global data buffer. The motion control routines are therefore isolated from the particular operator interface in use, isolated from the software emulating the LTM, and isolated from any other software necessary for running the manipulator, such as task specification.

Whether or not any particular optimization scheme can effectively operate in real-time is processor dependent. The optimizing inverse kinematics scheme is implemented on an MC68020 microprocessor with an MC68881 math coprocessor. The servo level control of the shoulder, elbow, and wrist joints is done by a separate microprocessor for each joint.

5.1. Program Overview

Figures 5-2, 5-3, and 5-4 give a function tree for all of the subroutines used in the robotic control software. Functions for the Macintosh interface are lumped together and indicated by the dotted square in Figure 5-2, but the individual interface functions have not been included in the tree. A detailed explanation of the Macintosh interface software is beyond the scope of this thesis, however, the software is listed in the appendix and the

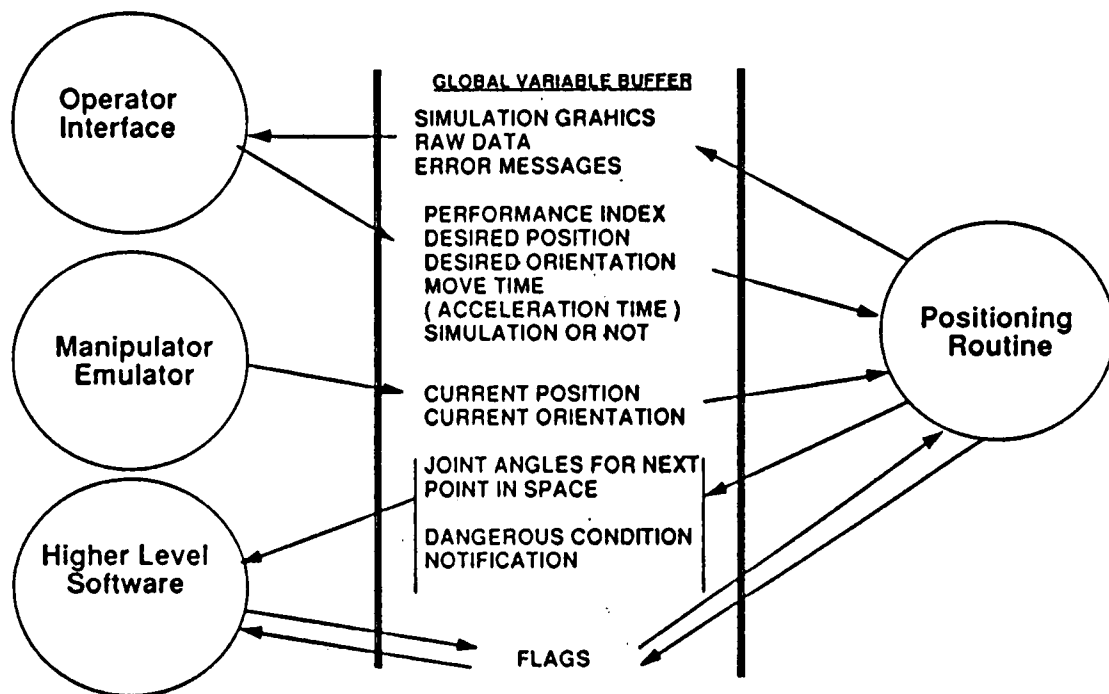


Figure 5-1 : Global variable buffer specifies a standard set of variables

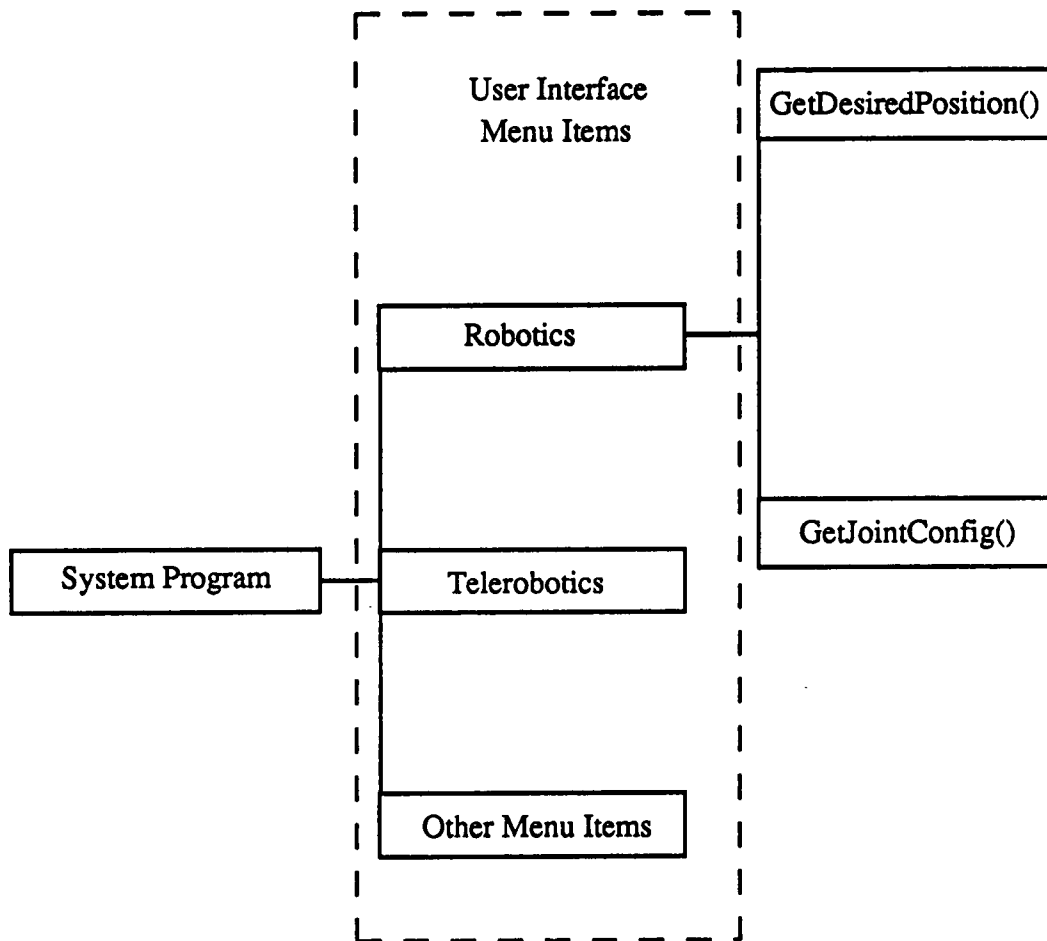


Figure 5-2 : Function tree of the robotic control software for the LTM

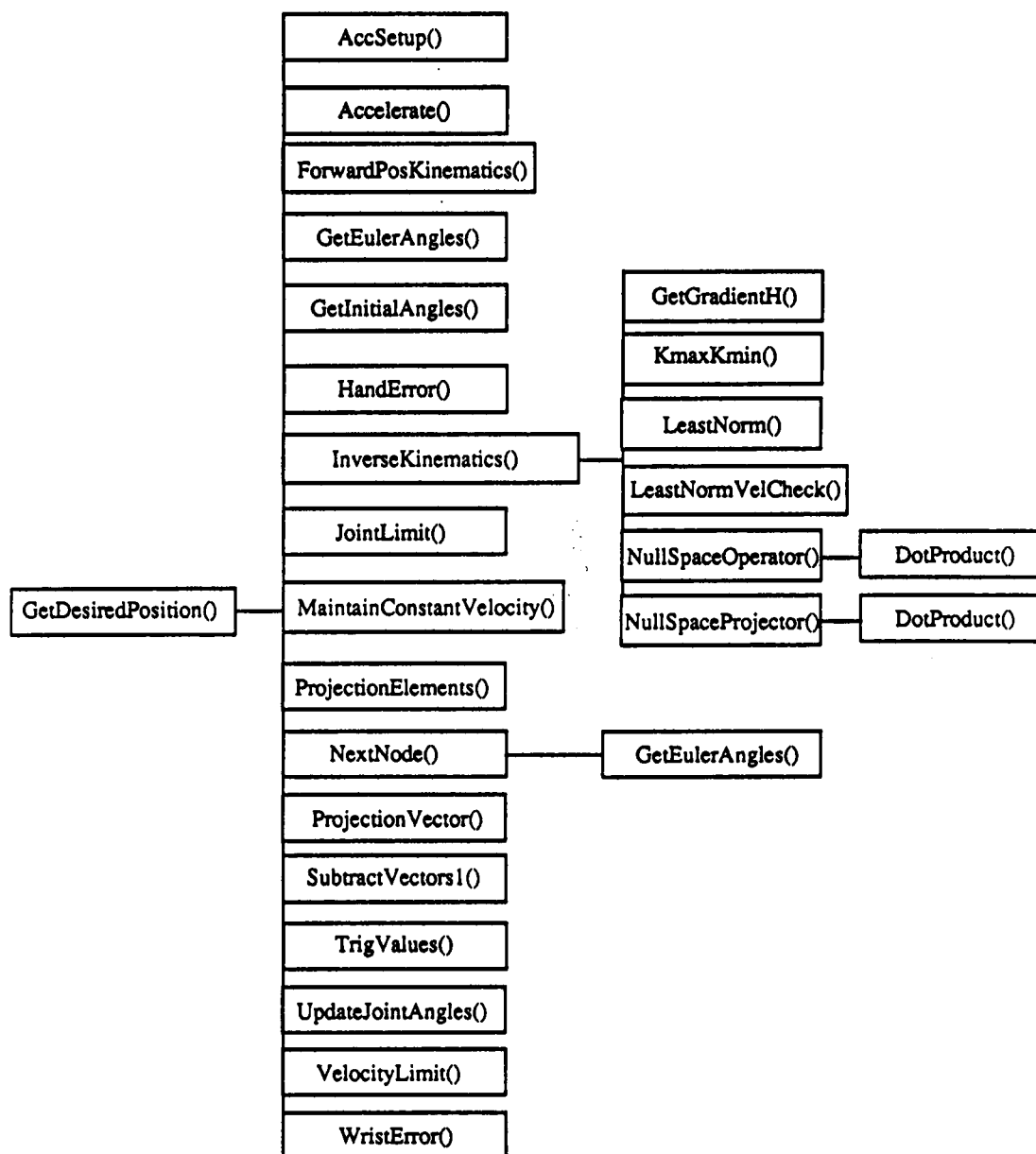


Figure 5-3 : `GetDesiredPosition()` function tree for motion in Cartesian space

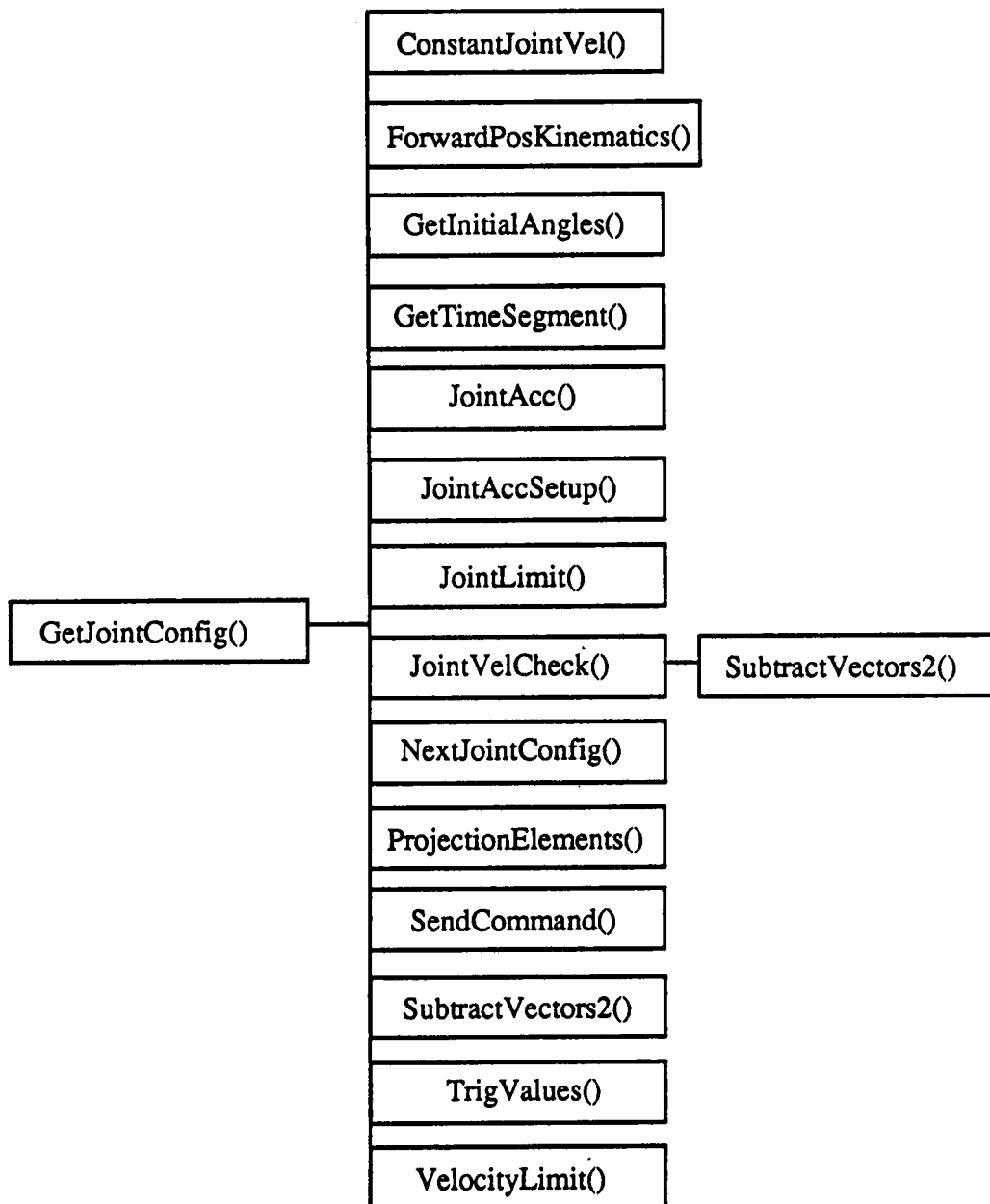


Figure 5-4 : `GetJointConfig()` function tree for motion in joint space

books *Inside Macintosh* Vols. 1-4 can be used as an aid in understanding the interface functions.

All functions necessary to move the end-effector in Cartesian space are called by the function `GetDesiredPosition()`, and all functions necessary to move the end-effector in joint space are called by the function `GetJointConfig()`. These two functions are based on the trajectory scheme chosen for robotic control. This trajectory scheme will be discussed in Section 5.3. where the details of each function is presented. The next section gives short, concise descriptions of the robotic control functions.

5.2. Concise Description of Program Functions

5.2.1. Cartesian Space Functions

The following is an alphabetical listing of the functions used for positioning the end-effector of the LTM in Cartesian space:

Accelerate(): Calculates the next desired position (in Cartesian coordinates) and orientation (in Euler angles) of the end-effector during the time the hand is accelerating or decelerating.

AccSetup(): Precalculates values used by `Accelerate()`.

DotProduct(): Returns the dot product of two vectors each of 7 elements. Used to get the dot product of two joint velocity vectors.

ForwardPosKinematics(): This function does two things (1) Calculates the current position and orientation of the hand from the current joint angles and projects them onto the base coordinate system, and (2) passes the current world Cartesian coordinates of the elbow, wrist, and hand on to graphics routines.

GetEulerAngles(): Converts the hand's orientation in matrix form (rotation matrix referenced to base coordinate system) into an Euler angle representation of the orientation.

GetGradientH(): Calculates the values for ∇H and put them in a vector to be used in `InverseKinematics()`.

GetInitialAngles(): Its only purpose is to get the initial joint angles at the beginning of a motion along a trajectory. These initial angles are used to calculate the starting position and orientation of the end-effector, which is not specified by the operator.

GetSize(): Obtains the magnitude of the seven element vector passed to it.

HandError(): Calculates the difference between the desired position and orientation of the hand, and the current position and orientation of the hand.

InverseKinematics(): Calculates the small changes in the joint angles, in a very small but finite time ΔT , necessary to follow the desired trajectory . Because the angles are so small they can be treated as vectors.

JointLimit(): Determines if a joint limit has been reached. If so, sets flags to stop the arm and causes a joint limit message to appear on the Macintosh screen.

KmaxKmin(): Determines the maximum and minimum values for the constant k , which is used to scale the homogeneous solution and determine the direction for the projection of ∇H upon the null space of the Jacobian.

LeastNorm(): Calculates the minimum Euclidian norm solution to $\dot{\mathbf{x}} = \mathbf{J} \dot{\boldsymbol{\theta}}$.

LeastNormVelCheck(): Determines if joint velocity limits will be exceeded by the least norm solution (no optimization occurring).

MaintainConstantVelocity(): Calculates the next desired position and orientation of the hand such that the hand will move with a constant velocity.

NextNode(): Gets the next node (position and orientation of the hand) in a trajectory path from the global data buffer.

NullProjector(): The null space projection function described in Chapter 3.

NullSpaceOperator(): Used to precalculate a vector for the NullProjector().

ProjectionElements(): Calculates the elements of the matrix used to project vectors from the 3rd coordinate frame onto the base coordinate frame and vice versa.

ProjectVector(): Uses the projection matrix calculated in ProjectionElements() to project a vector onto the base frame from the 3rd frame or vice versa.

SendCommand(): Puts the updated joint angles into the global data buffer to be used by the PID controller to move the arm.

SubtractVectors1(): Subtracts two vectors which both contain the position of the end-effector in Cartesian coordinates, and the orientation of the end-effector in Euler angles.

TrigValues(): Calculates the sines and cosines of the current joint angles and puts them into two structures to be used by various other functions.

UpdateJointAngles(): Updates the current joint angles with the small changes calculated by InverseKinematics().

VelocityLimit(): Causes an error message to appear on the Macintosh screen when a velocity limit has been reached.

WristError(): Calculates the difference between the current position and orientation of the wrist and the desired position and orientation of the wrist. This information is used by InverseKinematics().

5.2.2. Joint Space Functions

The following is an alphabetical listing of the functions used for moving the arm in joint space:

ConstantJointVel(): Calculates the next desired set of joint angles such that the joint angles will change with a constant angular velocity.

ForwardPosKinematics(): This function passes the current world Cartesian coordinates of the elbow, wrist, and hand on to graphics routines.

GetInitialAngles(): Its only purpose is to get the initial joint angles at the beginning of a motion along a trajectory. These initial angles are used to calculate the starting position and orientation of the end-effector, which is not specified by the operator.

GetTimeSegment(): Calculates the time between nodes.

JointAccSetup(): Precalculates values used by JointAcc().

JointLimit(): Determines if a joint limit has been reached. If so, sets flags to stop the arm and causes a joint limit message to appear on the Macintosh screen.

JointVelCheck(): Determines if joint velocity limits will be exceeded.

NextJointNode(): Gets the next node (set of joint angles) in a joint space path from the global data buffer.

ProjectionElements(): Calculates the elements of the matrix used to project vectors from the 3rd coordinate frame onto the base coordinate frame and vice versa.

SendCommand(): Puts the updated joint angles into the global data buffer to be used by the PID controller to move the arm.

SubtractVectors2(): Subtracts two vectors which both contain the angles of the joints.

TrigValues(): Calculates the sines and cosines of the current joint angles and puts them into two structures to be used by various other functions.

VelocityLimit(): Causes an error message to appear on the Macintosh screen when a velocity limit has been reached.

5.3. Detailed Description of Program Functions

The following conventions in notation and method are used in describing the program functions:

When a function (subroutine) name is used in the text of the thesis, it will be identified by a set of empty parenthesis immediately following the name. Additionally, because the C programming language is case sensitive, the exact name of the function will

be used. To illustrate these two conventions the notations `ThisIsAFunction()` and `thisIsAFunction()` would designate two different functions because, even though they both say the same thing and even though both names are followed immediately by a set of empty paranthesis, the first name starts with `T` while the second starts with `t` and so they designate different C functions.

Variable names and program code will be in bold face type to set them apart from the text of the thesis. Some examples are **theVariable** and **if(x == 1)** which represent a variable and program code respectively.

Whenever possible function names and variables were chosen to describe their purpose, therefore, lengthy discussion of names is not done as it is unnecessarily redundant to say, for instance, that the variable **JointAngles** contains the angles of the manipulator.

5.3.1. Cartesian Space Functions

5.3.1.1. **GetDesiredPosition()**

GetDesiredPosition() calls all other functions necessary for positioning the LTM in Cartesian space. However, before beginning the general description of **GetDesiredPosition()** we must first briefly discuss the method chosen to implement a complete task. Task specification involves the specification of a beginning position and orientation and an ending position and orientation of the manipulator end-effector with or without specified positions and orientations in between. Specification is done using the homogeneous transformation matrix. Each specified position and orientation of the end-effector is termed a node. In actual practice, because the LTM has absolute joint angle encoders the beginning node, or node zero, is obtainable from the initial joint angles which are transformed into the needed position and orientation. There are, however, many ways to implement motion between the beginning and ending nodes.

As described by Paul [14] motion may stop at each node as in Fig. 5-5. Such motion, however, is undesirable since we would like the manipulator to stop at the last node only and not in the midrange. Another alternative would be straight line segments between each node as in Fig. 5-6, but such motion obviously lacks continuity in velocity and acceleration and would not be possible. In order to provide for continuity in position, velocity, and acceleration (to prevent vibration and jerk), I have implemented a scheme proposed by Paul which utilizes a quartic polynomial [13]. Because the **GetNextPosition()** function involves this scheme, a general, though not in depth,

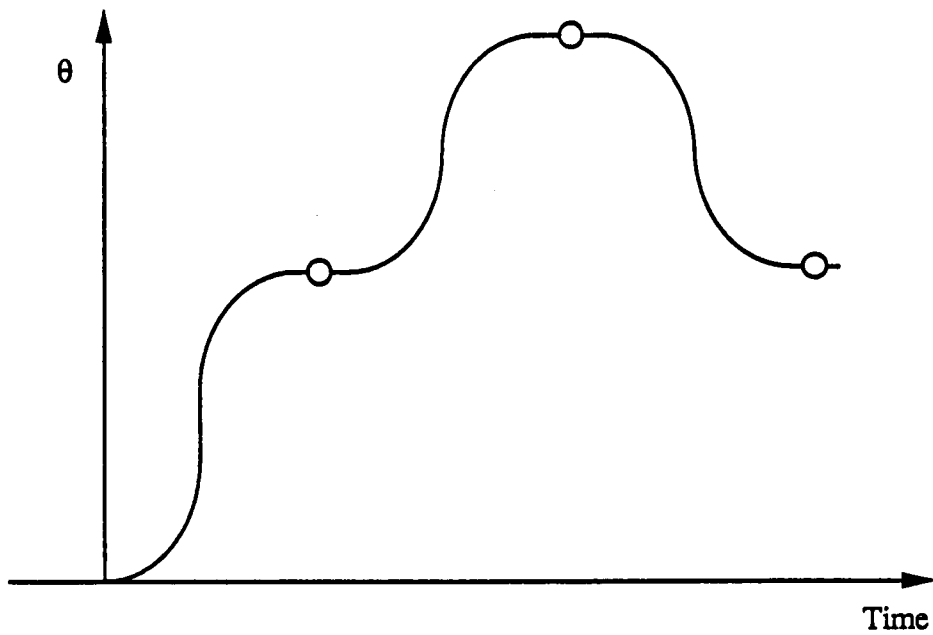


Figure 5-5 : Starting and stopping motion

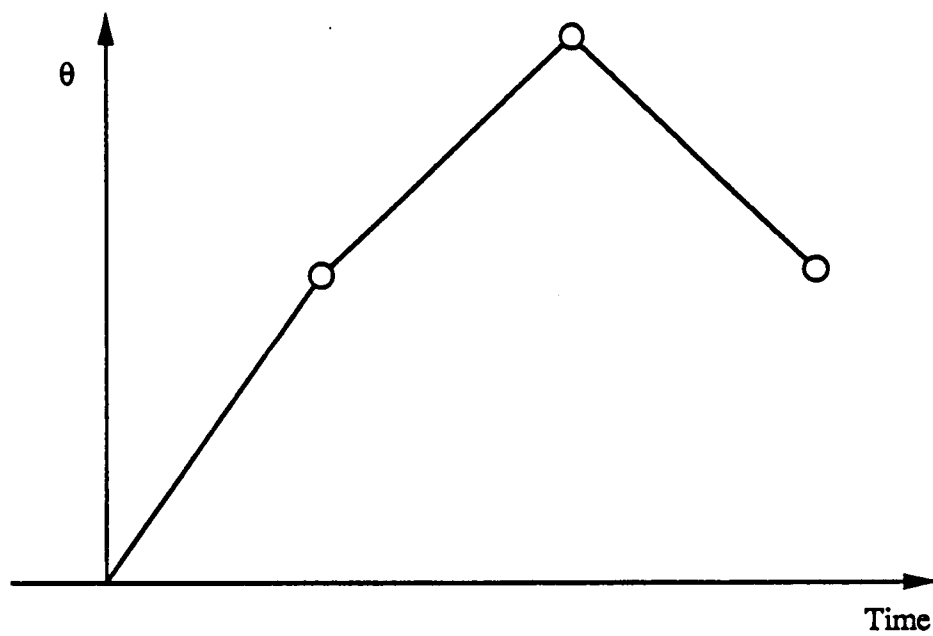


Figure 5-6 : Straight line motion

understanding of it is necessary before proceeding with the discussion of `GetNextPosition()`.

We first look at a generalized coordinate, q , shown in Fig. 5-7, where the T_i represent the total time for motion between two positions for q . For our purposes q could be a Cartesian coordinate, Euler angle, or joint angle. For the purpose of accelerating from one path segment to the other we see that at the point of acceleration A we really only need to know the next node in order that we can know where the next straight line segment is. Thusly, at position A we obtain position C in order that we can know the straight line segment between positions B and C . Knowing this we can start the acceleration at time $-t_{acc}$ and complete the acceleration at time t_{acc} . The acceleration time t_{acc} is the time to accelerate to maximum velocity. Note that because segment transition begins at time $-t_{acc}$, a full stop during segment transition is allowed for should a complete reversal in direction be necessary. This exact scheme is followed in `GetDesiredPosition()`.

The flow chart for each iteration of `GetDesiredPosition()` (see Fig. 5-8) shows that there are three different cases which must be considered for any motion between two points. The zero case obtains the initial position and orientation of the manipulator arm (the zeroth node) by passing the initial joint angles to a forward kinematics function. It then loads necessary information from the global data buffer, and finally calculates the very first desired position and orientation of the end-effector. Case 1 is concerned totally with calculating the next desired position and orientation of the end-effector during the midrange motion, and case 2 is concerned with deceleration for stopping. We will first put our attention on case 0.

case 0:

```

casenum = 1;
PerfCrit = LTM->l.s.hand.perf_crit;
TheLastNode = LTM->l.s.hand.LastNode;
NextNode (&C);
GetInitialAngles (&theta);
TrigValues (&Cos,&Sin,theta);
ProjectionElements (Cos,Sin,&P);
ViewArm0 =ForwardPosKinematics(&current_pos,Cos,Sin,P);
GetEulerAngles (current_pos,&A);
B = A;
delta_B = NULL_STRUCT;
SubtractVectors1 (C,B,&delta_C);
t = LTM->l.s.hand.move_time;
nt = t/delta_t + .5;
na = nt*LTM->l.s.hand.accFactor;
if (TheLastNode)
    nt = nt - (na + na);
else
    nt = nt - na;
i = -na;

```

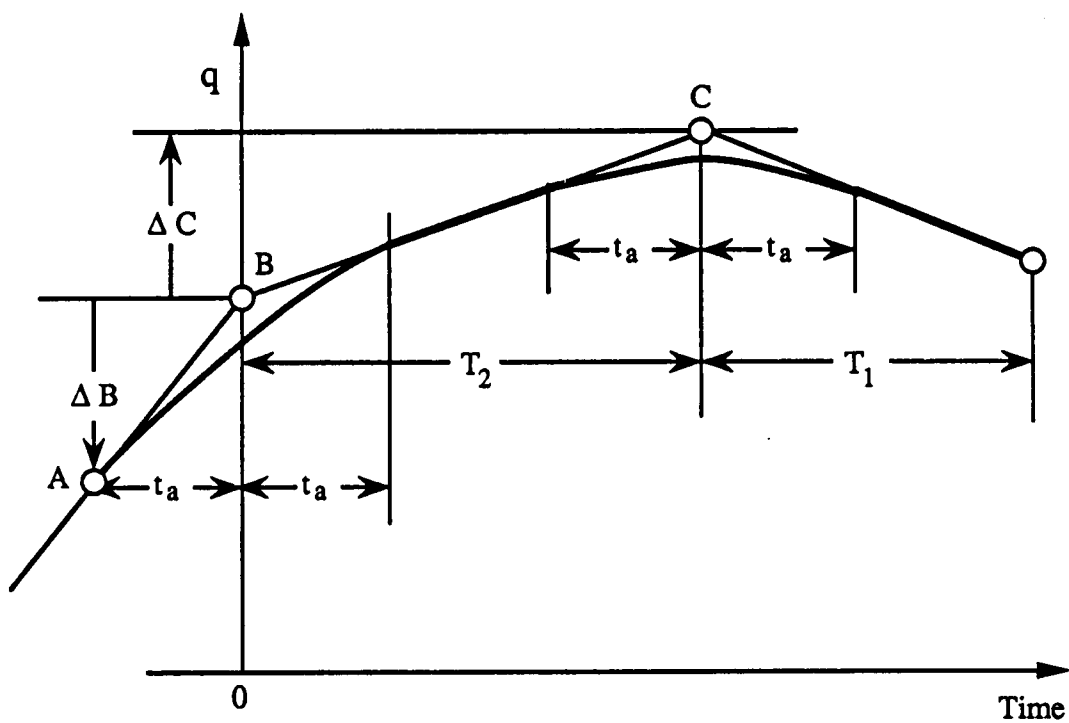


Figure 5-7 : Trajectory straight line segment transitions

(Paul R.P. "Robot Manipulators: Mathematics, Programming, and Control" Cambridge: MIT Press)

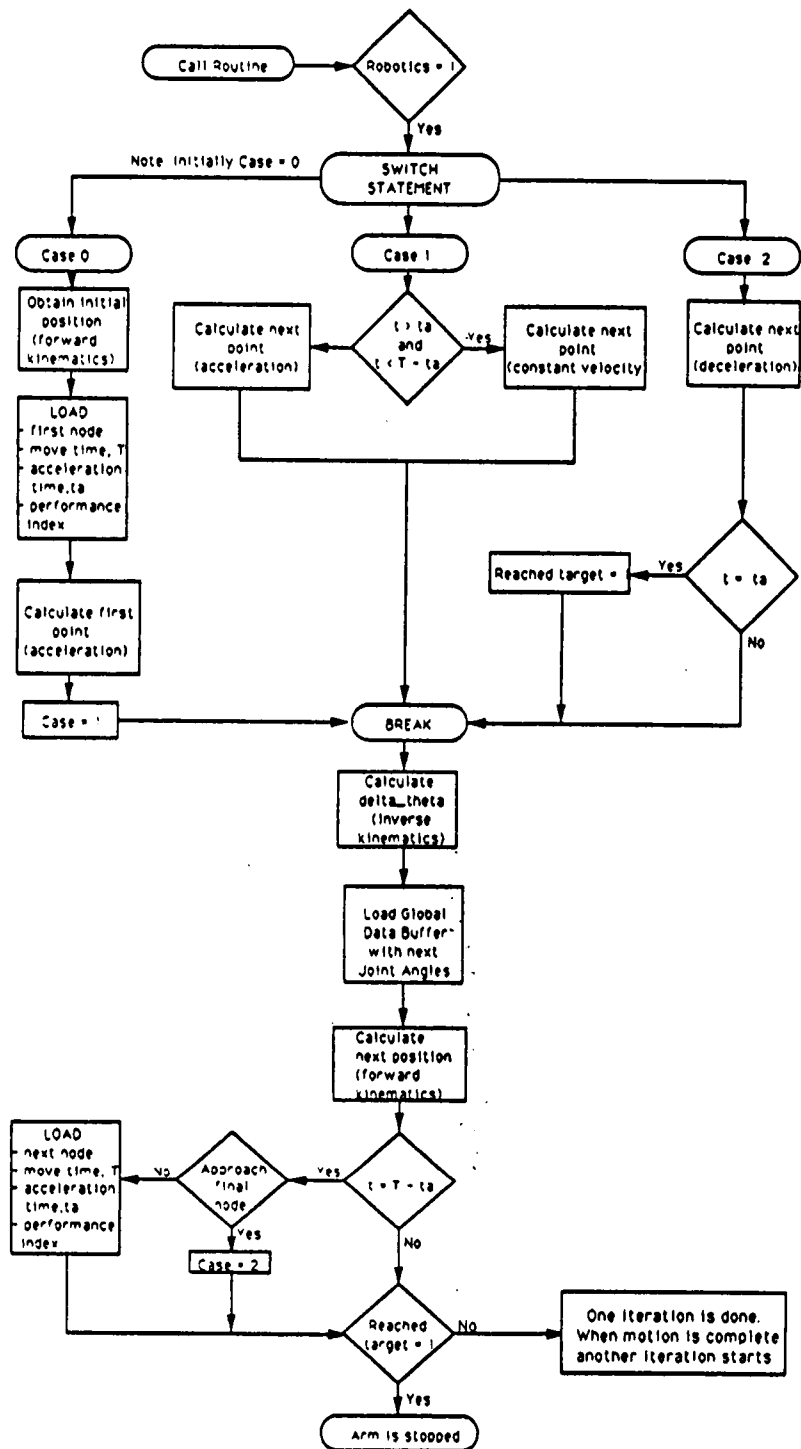


Figure 5-8 : Flowchart for GetDesiredPosition()

```

AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
break;

```

Just before the first iteration the first node was entered into the global data buffer and the manipulator is sitting at rest at the zeroth node. Case 0 is always chosen during the first iteration of `GetDesiredPosition()` and at no other time. Setting `casenum` to 1 assures that on the next iteration case 0 will be bypassed. The performance criterion and whether or not the first node is also the last node are both obtained from the global data buffer and stored into `PerfCrit` and `TheLastNode` respectively. `NextNode()` assigns the data for node 1 to `C` which is a structure (as are `A` and `B`) which contains the Cartesian coordinates for the end-effector's position and the Euler angles describing the end-effector's orientation. Note that at first glance it would seem that to be in accordance with Fig. 5-7 `NextNode()` should assign node 1 to `B`, however, because the manipulator starts from rest and must accelerate, it must be that `B` is the same position as `A`, and node 1 must be assigned to `C`.

The functions `GetInitialAngles()`, `TrigValues()`, and `ProjectionElements()` manipulate the current joint angles so that forward kinematics can be performed on them in `ForwardPosKinematics()`: `GetInitialAngles()` obtains the initial angles of the manipulator so that the variable `theta` can be passed to `TrigValues()`, which assigns the sine and cosine of the manipulator's joint angles to the structures `Sin` and `Cos`, which are then passed to `ProjectionElements()`. `ProjectionElements()` simply puts into `P` elements that are used to project or transform the sine and cosine of the joint angles from the base coordinate frame to the third coordinate frame and back again. Finally `ForwardPosKinematics()` puts the current position and orientation of the end-effector into the structure `current_pos`, whose elements are analogous to those of a homogeneous transformation matrix. `ForwardPosKinematics()` also returns the world Cartesian coordinates of the manipulator arm to `ViewArm0`, which is passed to graphics routines.

`GetEulerAngles()` simply takes the elements of the homogeneous transformation describing the current position and orientation of the end-effector and transforms the orientation submatrix to a set of Euler angles. It then places these Euler angles and the Cartesian position of the end-effector into `A`, the values of which are assigned to `B` for reasons already mentioned.

The variables `delta_B` and `delta_C` are ΔB and ΔC in Fig.5-7. Because `A` and `B` are the same in this case, all the elements in `delta_B` have been set to zero using a null structure called `NULL_STRUCT`, whose elements are all zero.

The total time to move from B to C is stored in t and then converted into the total number of steps nt in time t . The duration of acceleration t_{acc} has been specified as a fraction of the total time and stored in the global variable `LTM->l.s.hand.accFactor`, which is then used to obtain the number of steps na in acceleration time t_{acc} .

If the first node is also the last node then we must subtract two times na from nt , otherwise we only subtract one times na from nt . This is necessary if we want the actual time of travel to match the specified time of travel. As shown in Fig.5-9, in order to begin accelerating we had to start at a time $-t_{acc}$ before reaching B. In the midrange of motion this causes no problems timewise, but initially we see that in order to begin accelerating from a dead stop at the zeroth node we must, nevertheless, start at a time $-t_{acc}$ before reaching B, but since we are already at B we simply subtract the extra time from the total time, which has the effect of decreasing the duration for constant velocity motion. Later, when stopping motion is described it will be seen that an extra amount of time equal to t_{acc} is required to stop smoothly. Thusly, if the first node is also the last we must subtract two times na from nt , but if it is not the last we only subtract one times na from nt and wait for the last node to subtract the specific na for that node from the nt for that node, as they may be different from those for node 1.

`AccSetup()` does some initial computations for the function `Accelerate()`, which calculates the desired position and orientation of the end-effector in Cartesian coordinates and Euler angles respectively and assigns them to the structure `desired_pos`.

In summary, case 0 changed the initial joint angles of the manipulator into a starting position and orientation of the end-effector and used that information and the information of node 1 to obtain the very first desired position and orientation for the end-effector.

```
case 1:
    if ( i >= na+1 AND i < nt-na )
        MaintainConstantVelocity(pB,nt,delta_C,&desired_pos,i);
    else
        Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
    break;
case 2:
    Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
    if (i == na){
        LTM->flags.robotics = 0;
        casenum = 0;}
    break;
```

Cases 1 and 2 simply provide the structure `desired_pos` with the next desired position and orientation of the end-effector of the manipulator. Case 2, however, has the

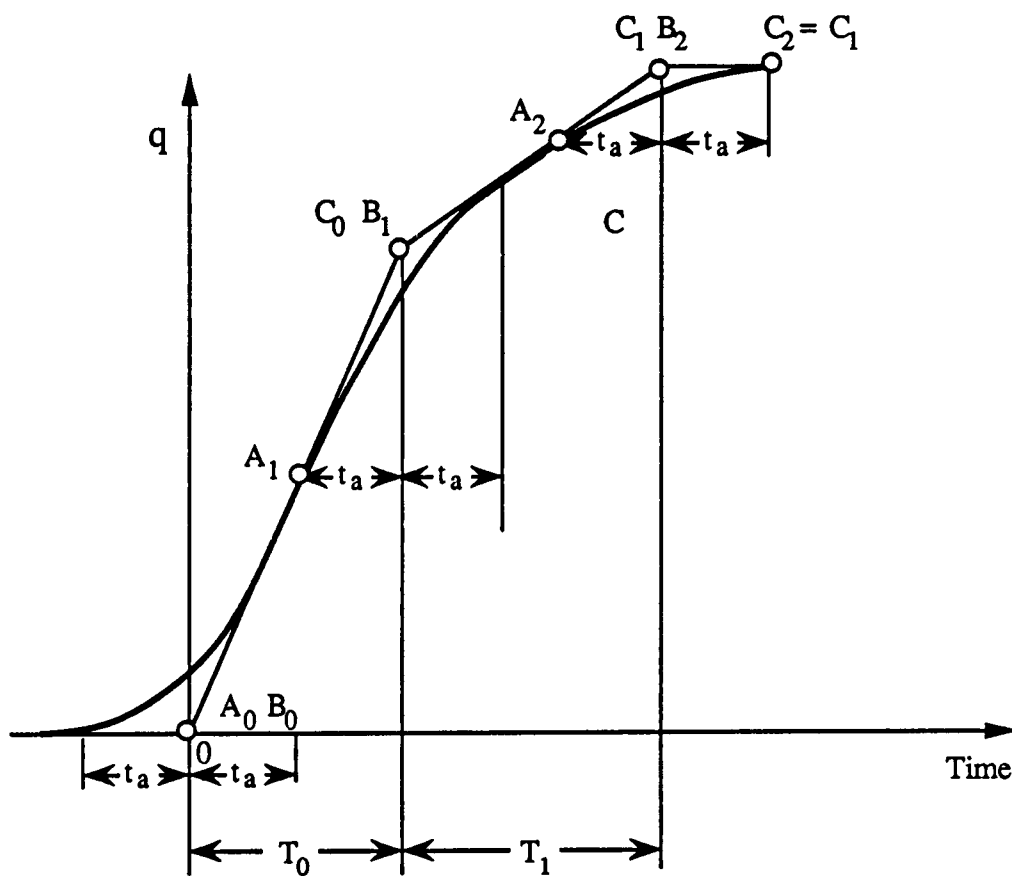


Figure 5-9 : Starting and stopping

added jobs of notifying higher level code via the global data buffer that the last node has been reached and of setting `casenum` to zero in preparation for the next move.

```

HandError (current_pos,desired_pos,&h_err);
WristError (h_err,current_pos,&w_err);
ProjectVector (&w_err.dx,&w_err.dy,&w_err.dz,P,3);
ProjectVector
(&w_err.dThetaX,&w_err.dThetaY,&w_err.dThetaZ,P,3);
if(InverseKinematics (w_err,PerfCrit,&dTheta,Cos,Sin,theta) == 1){
    VelocityLimit();
    LTM->flags.robotics = 0;
    LTM->flags.VelLimit = 1;
    casenum = 0; }
UpdateJointAngles (dTheta,&theta);
TrigValues (&Cos,&Sin,theta);
ProjectionElements (Cos,Sin,&P);
ViewArm = ForwardPosKinematics (&current_pos,Cos,Sin,P);
JointLimit (&casenum,theta);
if (!LTM->flags.simulate)
    SendCommand (theta);

```

Figure 5-10 shows a bubble diagram of the code after breaking out of the switch statement. The first function encountered is `HandError()`, which calculates the difference between the current position and orientation and the desired position and orientation of the end-effector and places the result in `h_err`. However, because `InverseKinematics()` uses the error of the wrist, `h_err` is passed to `WristError()` which calculates the difference between the desired and current positions of the wrist. This difference is placed into `w_err`. The elements of `w_err` are then passed to `ProjectVector()`, which projects them onto the third coordinate frame in preparation for use by `InverseKinematics()`. Note that for the first iteration the elements of the projection matrix have already been calculated in case 0 by `ProjectionElements()`. We will see shortly where they are calculated for the other two cases.

If `InverseKinematics()` returns the value 0 the next finite step in joint position has been successfully calculated and placed in the structure variable `dTheta`; however, if `InverseKinematics()` returns the value 1 the size of `dTheta` is such that the motor required to rotate the fastest cannot keep up with the requested size of `dTheta` and, therefore, the end-effector will not track as desired. In this case an error message is sent to the operator, the arm is stopped, flags are set, and housekeeping is done to prepare for a new attempt at moving the manipulator.

If the next finite step in joint position has been successfully calculated, the joint angles structure `theta` is updated by the amount `dTheta` through the `UpdateJointAngles()` function. New projection elements are then calculated and the new current position (to be)

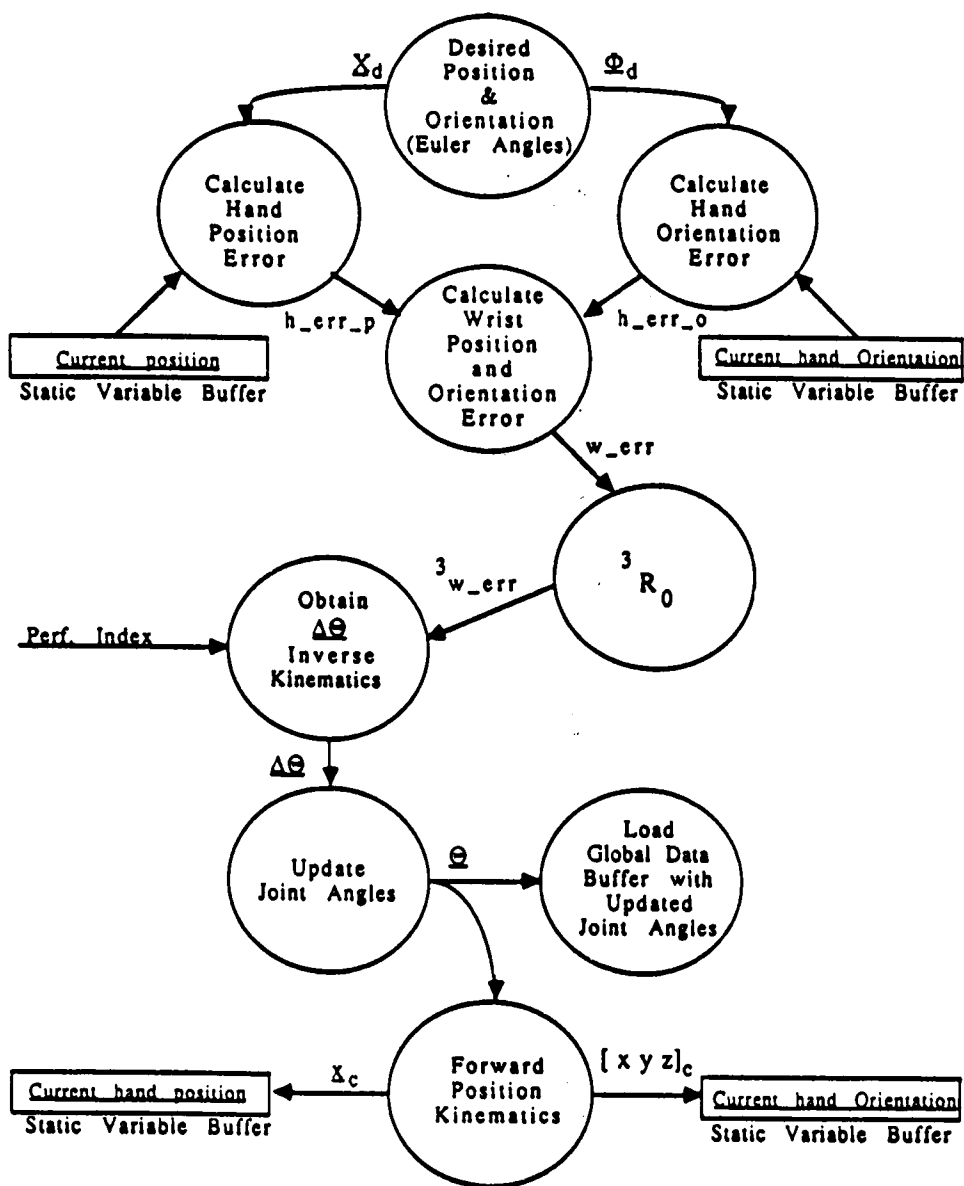


Figure 5-10 : Flow of variables in GetDesiredPosition() after switch statement

is placed into **current_pos** by **ForwardPosKinematics()**. **ForwardPosKinematics()** also returns the world coordinates of the manipulators elbow, wrist, and end-effector in the variable **ViewArm**, which is used in graphics routines. The projection elements **P** and **current_pos** will be used during the next iteration of **GetDesiredPosition()**.

Note that the new current position as calculated by **ForwardPosKinematics()** does not consult the actual position of the manipulator and, indeed, the calculation is done before the arm is even commanded to move. The loop was not closed in this case because previous work performed at Oak Ridge National Laboratory with the CESAR 7-degree-of-freedom redundant research manipulator used an open loop structure [7] which did not significantly result in positioning errors, but which did result in greater speed over closed loop operation [ORNL I&C Division]. Additionally, closed loop control has the added problem of possible instability.

Whether or not a joint limit has been reached is then checked by **JointLimit()**. If all of the joints are within the specified joint limits and a simulation is not occurring, then the command to actually move the manipulator is sent by **SendCommand()**. If **LTM->flags.simulate** is anything but a 0, the variable **theta** is simply not passed to the global data buffer, which is a further precaution to ensure that the manipulator does not move during simulation.

```

++i;
if ((i == nt-(na+1)) AND !TheLastNode)
    LTM->flags.load_next_pos=1;
if (i == nt-na AND TheLastNode){
    casenum = 2;
    GetEulerAngles (current_pos,&A);
    B = C;
    if ((fabs(B.psi - A.psi)) > PI)    A.psi = -A.psi;
    SubtractVectors1(pA,pB,&delta_B);
    delta_C = NULL_STRUCT;
    AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
    i = -na;}
else if (i == nt-na){
    GetEulerAngles (current_pos,&A);
    B = C;
    if ((fabs(B.psi - A.psi)) > PI)    A.psi = -A.psi;
    NextNode (&C);
    SubtractVectors1(A,B,&delta_B);
    SubtractVectors1 (C,B,&delta_C);
    t = LTM->l.s.hand.move_time;
    nt = t/delta_t + .5;
    na = nt*LTM->l.s.hand.accFactor;
    PerfCrit = LTM->l.s.hand.perf_crit;
    TheLastNode = LTM->l.s.hand.LastNode;
    if (TheLastNode) nt = nt - na;
    AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
    i = -na;}

```

The counter *i* is increased by one. This counter keeps track of the number of steps between each node, not over the whole path.

If if ((*i* == *nt*-(*na*+1)) AND !*TheLastNode*) is true, LTM->*flags.load_next_pos* is set so that the next node can be loaded into the global data buffer before the next iteration of the positioning function *GetDesiredPosition()*, when acceleration will begin again. At $t = -t_{acc}$, which occurs when *i* = *nt-na*, the current position of the end-effector is assigned to *A* and values of *C* are assigned to *B* in accordance with Fig. 5-7. At this time it is crucial to check if the Euler angles assigned to *A* are close to those assigned to *B*, if they are not then *A* is corrected by 360 degrees. This check and correction is necessary since the function *GetEulerAngles()* is not capable of assigning unique Euler angles to *A*, because Euler angles themselves are not unique (Even though *GetEulerAngles()* will always assign to *A* Euler angles that describe the correct orientation of the end-effector, we are looking for the set of Euler angles closest in value to *B*, otherwise *dTheta* will be of the order of hundreds of degrees rather than a very small fraction of one degree). Because we are not approaching the last node we must calculate both *delta_B* and *delta_C*, which is done by calling *SubtractVectors1()* twice. We now fill up necessary variables that will be used later, as was done in case 0. Note, however, that if the node just loaded is the last node, we subtract *na* from *nt* as was mentioned in the discussion of case 0. *AccSetup()* then assigns values to *f1*, *f2*, and *two_na* to be used later by *Accelerate()*, and finally *i* must be reset to *-na*, in accordance with Fig. 5-7.

If (*i* == *nt-na* AND *TheLastNode*) is true the final position and orientation of the end-effector is being approached. This is a special case which must be treated differently. Referring to Fig. 5-9 it can be seen that in order to come to a smooth stop the new node assigned to *C* must actually be the node previously loaded. Conceptually, we give *C* the same node twice. In actual practice, after assigning the old values of *C* to *B*, instead of loading the same node into *C* again, we simply assign the null structure *NULL_STRUCT* to *delta_C*. The variables that had to be filled up at time $-t_{acc}$ when the end-effector was not approaching its final position and orientation do not have to be filled up again, since they do not change; however, because *delta_B* and *delta_C* have changed, *AccSetup()* is called. Additionally *casenum* is set to 2. Finally *i* must be reset to *-na* as before.

5.3.1.2. Accelerate()

The subsection on GetDesiredPosition() uses Fig. 5-7 to generally describe the method implemented for moving between two points. This subsection gives further details for path travel. When the first three derivatives of the quartic polynomial

$$q = a_4t^4 + a_3t^3 + a_2t^2 + a_1t + a_0 \quad (5.1)$$

are taken and boundary conditions as defined using Fig. 5-7 are applied, we obtain the following[13]:

$$q = \left[(\Delta C \frac{t_{acc}}{T_i} + \Delta B)(2 - h)h^2 - 2\Delta B \right] h + A \quad (5.2)$$

where

$$\Delta C = C - B \quad (5.3)$$

$$\Delta B = A - B \quad (5.4)$$

$$h = \frac{t + t_{acc}}{2t_{acc}} \quad (5.5)$$

At time t_{acc} acceleration is complete and motion lies on the straight line segment and we have

$$q = \Delta Ch + B \quad (5.6)$$

$$h = \frac{t}{T_i} \quad (5.7)$$

```
Accelerate (A,na,two_na,f1,f2,desired_pos,i)
HAND_POS_OR A,f1,f2,*desired_pos;
int          na,two_na,i; {
double      h,h2,h3,two_h2,twoh2_h3;
    h = ((double) i + na)/(two_na);
    h2 = h*h;
    h3 = h2*h;
    two_h2 = h2 + h2;
    twoh2_h3 = two_h2 - h3;
    desired_pos->x = (f1.x*twoh2_h3 - f2.x)*h + A.x;
    desired_pos->y = (f1.y*twoh2_h3 - f2.y)*h + A.y;
```

```

desired_pos->z      = (f1.z*twoh2_h3 - f2.z)*h + A.z;
desired_pos->phi    = (f1.phi*twoh2_h3 - f2.phi)*h + A.phi;
desired_pos->theta  = (f1.theta*twoh2_h3 - f2.theta)*h + A.theta;
desired_pos->psi    = (f1.psi*twoh2_h3 - f2.psi)*h + A.psi; }
EndFunction

```

Accelerate() uses (5.2) to place into the structure **desired_pos** the next desired position and orientation (referred to the world coordinate frame) of the end-effector. The variable **A** is the position and orientation of the end-effector at which acceleration begins in order to execute a smooth change from a current straight line trajectory to another straight line trajectory defined by points **B** and **C** in Fig. 5-7. The variable **two_na** is merely twice the variable **na**, which is the number of steps in t_{acc} , the time to accelerate to maximum velocity. The function **AccSetup()** provides **two_na**, **f1** and **f2** for **Accelerate()**. These three variables change only when a new node is loaded into **C**. Inspection of **AccSetup()** will reveal that **f1** corresponds to $(\Delta C \frac{t_{acc}}{T_i} + \Delta B)$ of (5.2) and **f2** corresponds to $2\Delta B$ of the same equation. The variable **twoh2_h3** corresponds to $(2 - h)h^2$. **Accelerate()**, therefore, is nothing more than the implementation of equation (5.2).

5.3.1.3. AccSetup()

Inspection of equation (5.2) will show that some parts of it can be precalculated and passed to **Accelerate()** virtually as constants during the entire time of acceleration.

```

AccSetup (na,nt,two_na,delta_C,delta_B,f1,f2)
int      na,nt,*two_na;
HAND_POS_OR  delta_B,delta_C,*f1,*f2; {
double    temp;
    *two_na = na + na;
    temp    = ((double)na)/nt;
    f1->x    = (delta_C.x * temp) + delta_B.x;
    f1->y    = (delta_C.y * temp) + delta_B.y;
    f1->z    = (delta_C.z * temp) + delta_B.z;
    f1->phi  = (delta_C.phi * temp) + delta_B.phi;
    f1->theta = (delta_C.theta * temp) + delta_B.theta;
    f1->psi  = (delta_C.psi * temp) + delta_B.psi;
    f2->x    = delta_B.x + delta_B.x;
    f2->y    = delta_B.y + delta_B.y;
    f2->z    = delta_B.z + delta_B.z;
    f2->phi  = delta_B.phi + delta_B.phi;
    f2->theta = delta_B.theta + delta_B.theta;
    f2->psi  = delta_B.psi + delta_B.psi; }
EndFunction

```

AccSetup() is passed the variables **na**, **nt**, **delta_C**, and **delta_B**, and calculates values for variables **f1**, **f2** and **two_na**. As described in section **GetDesiredPosition()** the variable **na** is the number of steps in acceleration time t_{acc} and the variable **nt** is the total

time T_i for the end-effector to move from node to node . The variables **delta_C** and **delta_B** correspond to ΔC and ΔB of Fig. 5-7. This function is called only at time $T_i - t_{acc}$ when acceleration starts and only if a new node is being loaded. Comparison of (5.2) and the variable structures **f1** and **f2** will reveal the following: **f1** corresponds to $(\Delta C_{T_i}^{t_{acc}} + \Delta B)$, and **f2** corresponds to $2\Delta B$. The variables **f1**, **f2** and **two_na** are therefore precalculated for **Accelerate()**, saving computer time for other necessary calculations.

5.3.1.4. DotProduct ()

The function **DotProduct()** calculates the dot product of two vectors expressed as structures of type **JOINT_ANGLES** (see Appendix H for a description of this structure type). It is used in **InverseKinematics()** and is self-explanatory.

```
double DotProduct (x,y)
JOINT_ANGLES x,y; {
double r;
    r = x.one*y.one + x.two*y.two + x.three*y.three + x.four*y.four;
    r += ( x.five*y.five + x.six*y.six + x.seven*y.seven);
    return (r); }
EndFunction
```

5.3.1.5. ForwardPosKinematics()

Given values of the sine and cosine for the joint angles of the manipulator, **ForwardPosKinematics()** calculates the current position and orientation of the end-effector in reference to the third coordinate frame. These coordinates are then immediately transformed to the base coordinate frame. When a simulation is underway this function also returns a structure, used in graphics routines, which contains the base frame coordinates of the origins of coordinate frames attached to the elbow, wrist, and end-effector. In the graphics representation of the manipulator there is a line projecting from the origin of the end-effector frame. This line is a unit vector which aids in graphically representing the orientation of the end-effector, but is not a physical link of the manipulator.

The names of variables in **ForwardPosKinematics()** closely follow the mathematical description of the forward kinematics discussed in Chapter 4, so that by referring to Chapter 4 it is easy to follow what is being done in **ForwardPosKinematics()**.

```
struct ViewData ForwardPosKinematics (c,Cos,Sin,P)
HOMOGEN_TRANS *c;
JOINT_ANGLES Cos,Sin;
```

```

MATRIX
double
double
double
struct ViewData    view;
    C4C5C6_minus_S4S6=(Cos.four*Cos.five*Cos.six)(Sin.four*Sin.six);
    C4S5 = Cos.four * Sin.five;
    S4C5C6_plus_C4S6=(Sin.four*Cos.five*Cos.six)+(Cos.four*Sin.six);
    S4S5 = Sin.four * Sin.five;
    S5C6 = Sin.five * Cos.six;
    C5S6 = Cos.five * Sin.six;
    c->s1 = (C4C5C6_minus_S4S6)*Cos.seven+C4S5*Sin.seven;
    c->s2 = (S4C5C6_plus_C4S6)*Cos.seven +S4S5*Sin.seven;
    c->s3 = Cos.five*Sin.seven - S5C6*Cos.seven ;
    c->n = -(C4C5C6_minus_S4S6)*Sin.seven+ C4S5*Cos.seven;
    c->n2 = -(S4C5C6_plus_C4S6)*Sin.seven+ S4S5*Cos.seven ;
    c->n3 = Cos.five*Cos.seven + S5C6*Sin.seven ;
    c->a1 = Sin.four*Cos.six + Cos.four*C5S6 ;
    c->a2 = Sin.four*C5S6 - Cos.four*Cos.six ;
    c->a3 = -Sin.five*Sin.six ;
    e1 = Constant_a2 * Cos.three;
    e2 = 0;
    e3 = Constant_a2 * Sin.three;
    w1 = e1 + Constant_a4 * Cos.four;
    w2 = Constant_a4 * Sin.four;
    w3 = e3;
    c->pos1 = w1 + Constant_d7 * c->a1;
    c->pos2 = w2 + Constant_d7 * c->a2;
    c->pos3 = w3 + Constant_d7 * c->a3;
    ProjectVector (&c->pos1,&c->pos2,&c->pos3,P,0);
    ProjectVector (&c->s1,&c->s2,&c->s3,P,0);
    ProjectVector (&c->n1,&c->n2,&c->n3,P,0);
    ProjectVector (&c->a1,&c->a2,&c->a3,P,0);
    if (LTM->flags.simulate) {
        ProjectVector (&e1,&e2,&e3,P,0);
        ProjectVector (&w1,&w2,&w3,P,0);
        view.elbow.x = e1;
        view.elbow.y = e2;
        view.elbow.z = e3;
        view.wrist.x = w1;
        view.wrist.y = w2;
        view.wrist.z = w3;
        view.hand.x = c->pos1;
        view.hand.y = c->pos2;
        view.hand.z = c->pos3;
        view.orient.x = c->pos1 + c->s1;
        view.orient.y = c->pos2 + c->s2;
        view.orient.z = c->pos3 + c->s3;
        return (view);    }}
EndFunction

```

Elements of the rotation matrix which transforms vectors from the third coordinate frame of the manipulator to the base coordinate frame are passed to ForwardPosKinematics() in P. The sine and cosine of all the joint angles of the manipulator are passed to ForwardPosKinematics() in the variables Sin and Cos

respectively. Preparatory calculations designed to minimize the number of mathematical operations are then done. The names given to the variables to which the results of the preparatory calculations are assigned reflect the operations done, for instance, the value of **S4S5** is the result of multiplying the sine of joint angle 4 by the sine of joint angle 5. The orientation matrix is then computed and placed into structure **c**, which stands for current position and orientation of the end-effector. The position of the elbow and wrist, both referred to the third coordinate frame, are then computed and the homogenous transformation representing the end-effector is then completed by the computation of the position vector with elements **c->pos1**, **pos2**, and **pos3**. The structure **c** is then transformed to base coordinates by repeatedly calling the function **ProjectVector()**.

If (**LTM->flags.simulate**) is true, elbow and wrist coordinates of the LTM are converted to base coordinates by **ProjectVector()** and placed into the structure **view**, which is returned by **ForwardPosKinematics()** to be used in graphics routines. The elements **view.orient.x**, **.y**, and **.z**, are elements of a unit vector used to graphically represent the orientation of the end-effector. While this unit vector is drawn on the graphics output, it does not represent a physical link of the LTM.

5.3.1.6. GetEulerAngles()

The function **GetEulerAngles()** converts into Euler angles the orientation submatrix of the homogeneous transformation matrix specified in a path node. Because the orientation submatrix is in base coordinates, Euler angles calculated in **GetEulerAngles()** are also in base coordinates. These Euler angles are used in the trajectory scheme in the following way: When the Euler angle representation for the orientation of the end-effector at two different nodes in a path specification are known, the functions **AccSetup()**, **Accelerate()**, and **MaintainConstantVelocity()** interpolate between them to cause a smooth change in orientation for the end-effector.

Paul's method of obtaining explicit solutions to Euler angles from the rotation matrix in a homogenous transformation [13] was used in **GetEulerAngles()**. If we designate elements of the end-effector's transformation matrix in the following way:

$$T = \begin{bmatrix} nx & ox & ax & px \\ ny & oy & ay & py \\ nz & oz & az & pz \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (5.8)$$

and if the Euler angles ϕ , θ , and ψ are defined with reference to the base coordinate frame by first a positive rotation ϕ about the z-axis, then a positive rotation θ about the new y-axis, and finally a positive rotation ψ about the new z-axis, then the equations developed in [13] (see Fig. 5-11) and used in GetEulerAngles() are the following:

$$\phi = \text{atan2}(\text{ay}, \text{ax}), \quad (5.9)$$

$$\theta = \text{atan2}(\cos\phi \text{ax} + \sin\phi \text{ay}, \text{az}), \quad (5.10)$$

$$\psi = \text{atan2}(\cos\phi \text{ny} - \sin\phi \text{nx}, \cos\phi \text{oy} - \sin\phi \text{ox}). \quad (5.11)$$

The atan2() function is a function that returns angles in the range of $-\pi \leq \text{angle} < \pi$ and give the right answer when either of its arguments are zero.

```

GetEulerAngles (hand,p)
HOMOGEN_TRANS hand;
HAND_POS_OR      *p; {
double           ox,oy,nx,ny,ax,ay,az;
double           cos_phi,sin_phi;
    nx = hand.s1;
    ny = hand.s2;
    ox = hand.n1;
    oy = hand.n2;
    ax = hand.a1;
    ay = hand.a2;
    az = hand.a3;
    if (ax == 0.0 AND ay == 0.0)
        p->phi = 0.0;
    else
        p->phi = atan2(ay,ax);
    cos_phi = cos (p->phi);
    sin_phi = sin (p->phi);
    p->theta = atan2(ax*cos_phi + ay*sin_phi , az);
    p->psi = atan2(-nx*sin_phi + ny*cos_phi , -ox*sin_phi+oy*cos_phi);
    p->x = hand.pos1;
    p->y = hand.pos2;
    p->z = hand.pos3; }
EndFunction

```

The first thing done is to rename the elements of the end-effector's orientation matrix. The angle ϕ is then calculated and put into the **phi** element of structure **p**. Note that if both **ay** and **ax** are 0 then ϕ is not defined and we will arbitrarily set it to 0 degrees. In this case the hand is either pointing straight up or straight down because θ is either 0 degrees or 180 degrees. The rest of the program is self-explanatory.

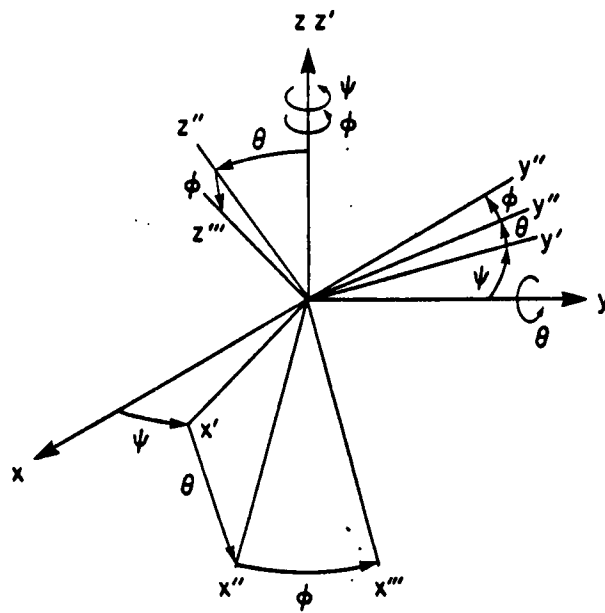
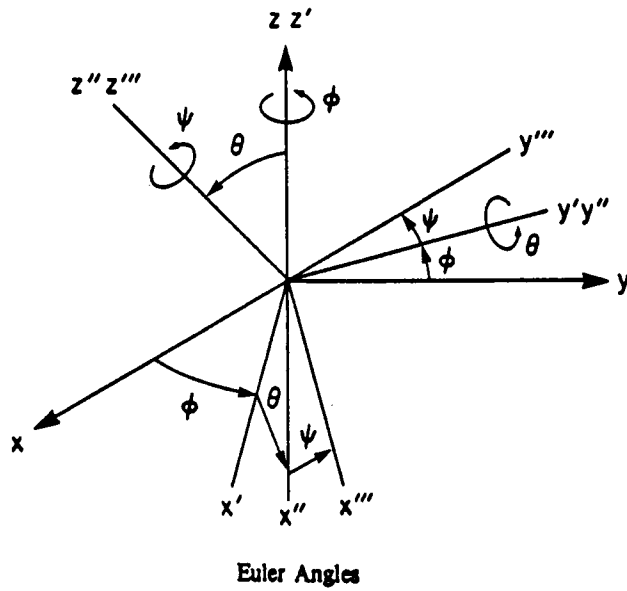


Figure 5-11 : Euler angles defined and interpreted in base coordinates.
 (Paul R.P. "Robot Manipulators: Mathematics, Programming, and Control", Cambridge, MIT Press)

5.3.1.7. GetGradientH()

GetGradientH() is used to obtain the gradient of an optimization criterion H. It is actually implemented on a module separate from the rest of the program. This is so that when the optimization criteria are changed, which may happen frequently, only this function needs to be recompiled, thusly saving time for the computer operator. This function is used in InverseKinematics().

```
GetGradientH(theta,Sin,Cos,CaseNum,kKind,gH)
JOINT_ANGLES    theta,Cos,Sin,*gH;
int CaseNum,*kKind;{
    switch (CaseNum){
        case 1: /*
                        JOINT LIMITS
                    */
                    gH->one   = 0;
                    gH->two   = 0;
                    gH->three  = 2*( theta.three - .9163 );
                    gH->four   = 0;
                    gH->five   = 2*( theta.five - .9163 );
                    gH->six    = 0;
                    gH->seven  = 0;
                    *kKind = 0;
                    break;
        case 2:
                    gH->one   = 2*( theta.one + 1.57 );
                    gH->two   = 0;
                    gH->three  = 0;
                    gH->four   = 0;
                    gH->five   = 2*( theta.five - .9163 );
                    gH->six    = 0;
                    gH->seven  = 0;
                    *kKind = 0;
                    break;}}
EndFunction
```

The vectors **theta,Sin,Cos**, are passed to GetGradientH() in order to calculate the gradient of the optimization criterion, which is chosen by passing the variable **CaseNum** to a switch statement. The person investigating criteria must keep track of which case number represents which criterion. The vector **theta** is full of the current joint angles, the vectors **Sin** and **Cos** are full of the sines and cosines of the current joint angles. The gradient of H is assigned to **gH** and **kKind** is set 0 to indicate minimization of H or 1 to indicate maximization of H.

5.3.1.8. GetInitialAngles()

GetInitialAngles() is called only once during each full path. It is called by GetDesiredPosition() on the very first iteration of any move. Its purpose is to load the current manipulator joint angles from the global data buffer into a structure recognizable by other functions called by GetDesiredPosition().

```
GetInitialAngles (theta)
JOINT_ANGLES      *theta;  {
    theta->one      =    LTM->l.s.shld.data.pitch_pos;
    theta->two       =    LTM->l.s.shld.data.yaw_pos;
    theta->three     =    LTM->l.s.elbow.data.pitch_pos;
    theta->four      =    LTM->l.s.elbow.data.yaw_pos;
    theta->five      =    LTM->l.s.wrist.data.pitch_pos;
    theta->six       =    LTM->l.s.wrist.data.yaw_pos;
    theta->seven     =    LTM->l.s.wrist.roll_data.pos;  }
EndFunction
```

5.3.1.9. GetSize()

GetSize() calculates the magnitude of the seven element vector passed to it. It is used to determine the cosine of the angle between the gradient of the optimization criterion and the homogeneous solution. It is self explanatory.

```
float GetSize(theta)
JOINT_ANGLES      theta;{
float    theSize;
    theSize = theta.one*theta.one + theta.two*theta.two + theta.three*theta.three;
    theSize += (theta.four*theta.four + theta.five*theta.five + theta.six*theta.six );
    theSize += theta.seven*theta.seven;
    return(pow(theSize,0.5);}
EndFunction
```

5.3.1.10. HandError()

HandError() calculates the extremely small but finite changes in the end-effector position and rotation, both referred to the base coordinate frame. In a small increment of time, such as the time between iterations of GetDesiredPosition(), these changes result in the linear and angular velocities of the end-effector.

The rotation necessary to carry the end-effector coordinate frame from its current orientation to its desired orientation must be derived from the current orientation contained in the homogeneous transformation passed to HandError(), and the desired orientation

contained in the set of Euler angles passed to HandError(). If the Euler angles can be transformed into an orientation matrix, then the following formula can be used:

$$\underline{d\theta} = 1/2[\underline{X_c} \times \underline{X_d} + \underline{Y_c} \times \underline{Y_d} + \underline{Z_c} \times \underline{Z_d}] \quad (5.12)$$

where $[\underline{X_c} \ \underline{Y_c} \ \underline{Z_c}]$ and $[\underline{X_d} \ \underline{Y_d} \ \underline{Z_d}]$ are the current orientation and desired orientation of the end-effector both expressed in the form of a rotation matrix, and $\underline{d\theta}$ is the three-dimensional rotation vector of the end-effector. It is easy to see why (5.12) works: Consider three separate rotations, one around each of the axes in Fig. 5-12. These rotations, if they are small enough, can be approximated by the following rotation vectors:

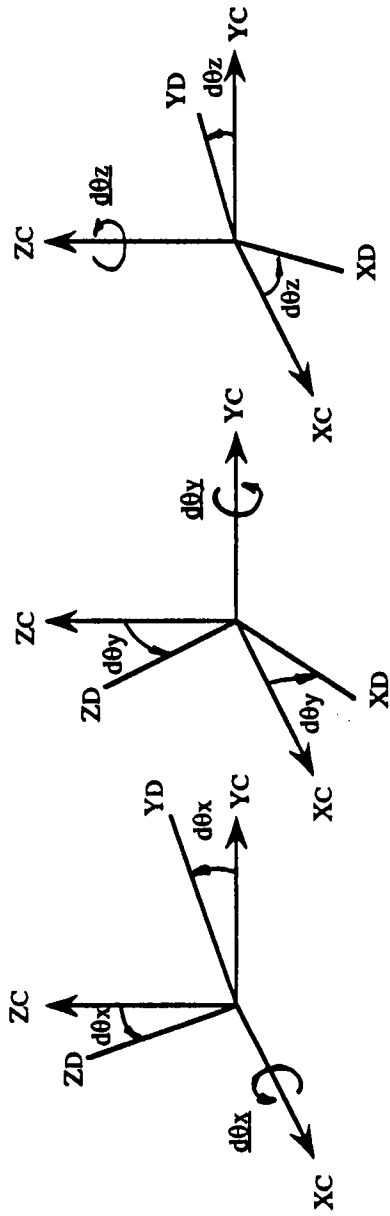
$$\underline{d\theta_x} = \begin{bmatrix} d\theta_x \\ 0 \\ 0 \end{bmatrix}, \quad \underline{d\theta_y} = \begin{bmatrix} 0 \\ d\theta_y \\ 0 \end{bmatrix}, \quad \underline{d\theta_z} = \begin{bmatrix} 0 \\ 0 \\ d\theta_z \end{bmatrix}. \quad (5.13)$$

The vector $\underline{d\theta_x}$, which represents a rotation around the X_c axis and which carries Y_c into Y_d and Z_c into Z_d , can be arrived at mathematically by either of the vector cross products $\underline{Y_c} \times \underline{Y_d}$ or $\underline{Z_c} \times \underline{Z_d}$. Vector cross products can also be found for $\underline{d\theta_y}$ and $\underline{d\theta_z}$. Additionally, because $\underline{d\theta_x}$, $\underline{d\theta_y}$, and $\underline{d\theta_z}$ are vectors we can add them which, except for the constant factor 1/2, results in (5.12) if the cross products are chosen correctly. However, it is obvious that the results of $\underline{Z_c} \times \underline{Z_d}$ when $\underline{d\theta_x}$ carries Z_c into Z_d are quite different from the results of $\underline{Z_c} \times \underline{Z_d}$ when $\underline{d\theta_y}$ carries Z_c into Z_d . Therefore, in order to add the rotation vectors we must express each individual rotation vector as the sum of its associated vector cross products. But because we are talking about unit vectors, the magnitude of that sum is now 2 instead of 1, so division of (5.12) by 2 is necessary in order to retain the resulting rotation vector as a unit vector.

As mentioned, Euler angle representation of the desired orientation must be transformed to rotation matrix form. Consider Fig. 5-11 where the Euler angles are defined. If we let $R(z,\psi)$ represent a positive rotation ψ around the z -axis, then the Euler angle representation of the desired orientation of the end-effector can be represented by

$$E(\phi, \theta, \psi) = R(z,\phi)R(y,\theta)R(z,\psi). \quad (5.14)$$

If these rotations are taken in reverse order, that is with $R(z,\psi)$ considered first, then they can be thought of as rotations in base coordinates (see Fig. 5-11). When multiplication of the rotation matrices is carried out we obtain



$$\begin{aligned}
 d\theta_x &= \begin{bmatrix} d\theta_x \\ 0 \\ 0 \end{bmatrix} = \frac{1}{2} (Y_C \times Y_D + Z_C \times Z_D) \quad d\theta_y = \begin{bmatrix} 0 \\ d\theta_y \\ 0 \end{bmatrix} = \frac{1}{2} (X_C \times X_D + Z_C \times Z_D) \quad d\theta_z = \begin{bmatrix} 0 \\ 0 \\ d\theta_z \end{bmatrix} = \frac{1}{2} (X_C \times X_D + Y_C \times Y_D)
 \end{aligned}$$

$$d\theta = d\theta_x + d\theta_y + d\theta_z = \frac{1}{2} (X_C \times X_D + Y_C \times Y_D + Z_C \times Z_D)$$

Figure 5-12 : Cross product representation of infinitesimal rotation vector

$$E(\phi, \theta, \psi) = \begin{bmatrix} \cos\phi\cos\theta\cos\psi - \sin\theta\sin\psi & -\cos\phi\cos\theta\sin\psi - \sin\theta\cos\psi & \cos\phi\sin\theta \\ \sin\phi\cos\theta\cos\psi + \cos\phi\sin\psi & -\sin\phi\cos\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\sin\theta \\ -\sin\theta\cos\psi & \sin\theta\sin\psi & \cos\theta \end{bmatrix} \quad -(5.15)$$

This matrix transforms the Euler angle representation of the orientation of the end-effector to a rotation matrix representation. Discussion of HandError() may now proceed.

```

HandError (c,desired,h)
HOMOGEN_TRANS      c;
HAND_POS_OR        desired;
INCREMENTS         *h; {
HOMOGEN_TRANS      d;
double             C_phi,S_phi,C_theta,S_theta,C_psi,S_psi;
double             Cphi_Ctheta,Sphi_Spsi;
    C_phi = cos(desired.phi);
    S_phi = sin(desired.phi);
    C_theta = cos(desired.theta);
    S_theta = sin(desired.theta);
    C_psi = cos(desired.psi);
    S_psi = sin(desired.psi);
    Cphi_Ctheta = C_phi * C_theta;
    Sphi_Spsi = S_phi * S_psi;
    d.s1 = Cphi_Ctheta*C_psi - Sphi_Spsi;
    d.s2 = S_phi*C_theta*C_psi + C_phi*S_psi;
    d.s3 = -S_theta*C_psi;
    d.n1 = -Cphi_Ctheta*S_psi - S_phi*C_psi;
    d.n2 = -Sphi_Spsi*C_theta + C_phi*C_psi;
    d.n3 = S_theta*S_psi;
    d.a1 = C_phi*S_theta;
    d.a2 = S_phi*S_theta;
    d.a3 = C_theta;
    h->dThetaX = c.s2*d.s3 - c.s3*d.s2 + c.n2*d.n3 - c.n3*d.n2;
    h->dThetaX += c.a2*d.a3 - c.a3*d.a2;
    h->dThetaX *= 0.5;
    h->dThetaY = c.s3*d.s1 - c.s1*d.s3 + c.n3*d.n1 - c.n1*d.n3;
    h->dThetaY += c.a3*d.a1 - c.a1 * d.a3;
    h->dThetaY *= 0.5;
    h->dThetaZ = c.s1*d.s2 - c.s2*d.s1 + c.n1*d.n2 - c.n2*d.n1;
    h->dThetaZ += c.a1*d.a2 - c.a2*d.a1;
    h->dThetaZ *= 0.5;
    h->dx = desired.x - c.pos1;
    h->dy = desired.y - c.pos2;
    h->dz = desired.z - c.pos3; }
EndFunction

```

First sines and cosines of the Euler angles in the structure **desired** are calculated and then the Euler angle representation of the orientation of the end-effector is transformed to a rotation matrix representation using the transformation matrix described above. Then the current orientation **c** is cross multiplied with the desired orientation **d** and the result is divided by 2. Finally the difference in the desired and current position of the end-effector is calculated.

5.3.1.11. InverseKinematics()

InverseKinematics() calculates small changes in the joint angles necessary to cause the end-effector to move in Cartesian space. Its total purpose is to implement the following equation

$$\dot{\underline{\theta}} = {}^3\mathbf{I}_w^{\oplus} {}^3\dot{\underline{x}}_w + k |\cos \beta|^y \left\{ \mathcal{P}(\underline{\nabla H}) \right\}, \quad (5.16)$$

which has been extensively discussed in Chapters 3 and 4. InverseKinematics() closely follows what was said in Chapter 4. Due to the length of this function it will be divided into relatively small parts for discussion.

```

InverseKinematics (W,CaseNum,dTheta,Cos,Sin,theta)
int                CaseNum;
INCREMENTS        W;
JOINT_ANGLES       theta,*dTheta,Cos,Sin;{
struct JOINTS      NullSpaceProjector();
struct JOINTS      LeastNorm();

                *
                *
                *

a2C2 = Constant_a2 * Cos.two;
a4S4 = Constant_a4 * Sin.four;
a4C4 = Constant_a4 * Cos.four;
S2S3 = Sin.two * Sin.three;
C4C5 = Cos.four * Cos.five;
S4C5 = Sin.four * Cos.five;
C6divS6 = Cos.six/Sin.six;
J11 = a4S4*S2S3 + a2C2*Sin.three;
J12 = a4S4*Cos.three;
J14 = -a4S4;
J21 = -a4C4*S2S3;
J22 = -a4C4*Cos.three - Constant_a2;
J24 = a4C4;
J31 = -a4S4*Sin.two*Cos.three - a4C4*Cos.two - a2C2*Cos.three;
J32 = a4S4*Sin.three;
J33= -a4C4;
J41 = -Sin.two*Cos.three;
J42 = Sin.three;
J51 = Cos.two;
J61 = -S2S3;
J62 = -Cos.three;
J3inv11 = (C4C5*C6divS6) - Sin.four;
J3inv12 = (S4C5*C6divS6) + Cos.four;
J3inv13 = -Sin.five*C6divS6;
J3inv21 = Cos.four*Sin.five;
J3inv22 = Sin.four*Sin.five;

```

```

J3inv23 = Cos.five;
J3inv31 = C4C5/Sin.six;
J3inv32 = S4C5/Sin.six;
J3inv33 = -Sin.five/Sin.six;
X1 = W.dx;
X2 = W.dy;
X3 = W.dz;
X4 = W.dThetahetaX;
X5 = W.dThetahetaY;
X6 = W.dThetahetaZ;

```

InverseKinematics() has the following structures passed to it: **W**, the desired changes in the wrist; **CaseNum**, an interger used to choose the optimization criterion; **Cos** and **Sin**, structures that contain the sines and cosines of the current angles of the joints; **theta**, a structure containing the current joint angles in rad; and **dTheta**, a structure to be filled up with the results of InverseKinematics().

The first thing done is calculate the needed elements of matrix ${}^3J_w$ and place the results into variables **J11**, **J12**, et cetera to be used later. Then the elements of ${}^3J_w^{-1}$ from (4.17) are calculated and placed in the variables **J3inv11**, **J3inv12**, et cetera to be used later (note that the variable names reflect the actual elements in the matrices and thus were easy to check during writing and debugging the program). Then the desired changes in the wrist are loaded into **X1**, **X2**, et cetera, which is helpful but not entirely necessary.

```

if (Sin.four<0.173648 AND Sin.four>-0.173648) {
    det = J11*J22 - J21*J12;
    /* finding the thetaP's for column 4 solution */
    thetaP.four = 0.0;
    thetaP.one = ( J22*X1 - J12*X2)/det;
    thetaP.two = (-J21*X1 + J11*X2)/det;
    thetaP.three = (X3 - J31*thetaP.one - J32*thetaP.two)/J33;
    X4 = X4 - J41*thetaP.one - J42*thetaP.two;
    X5 = X5 - J51*thetaP.one - thetaP.three;
    X6 = X6 - J61*thetaP.one - J62*thetaP.two;
    thetaP.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
    thetaP.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
    thetaP.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;
    /* finding the thetaH's for column 4 solution */
    X1 = -J14;
    X2 = -J24;
    thetaH.four = 1.0;
    thetaH.one = ( J22*X1 - J12*X2)/det;
    thetaH.two = (-J21*X1 + J11*X2)/det;
    thetaH.three = (-J31*thetaH.one - J32*thetaH.two)/J33;
    X4 = -J41*thetaH.one - J42*thetaH.two;
    X5 = -J51*thetaH.one - thetaH.three;
    X6 = -J61*thetaH.one - J62*thetaH.two - 1;
    thetaH.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
    thetaH.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
    thetaH.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;}
else /* Use solution that eliminates column 1 from J */{

```

```

det = -Constant_a2*a4S4;
/* finding the thetaP's for column 1 solution */
thetaP.one = 0.0;
thetaP.two = ( J24*X1 - J14*X2)/det;
thetaP.four = (-J22*X1 + J12*X2)/det;
thetaP.three = ( X3 - J32*thetaP.two)/(J33);
X4 = X4 - J42*thetaP.two;
X5 = X5 - thetaP.three;
X6 = X6 - J62*thetaP.two - thetaP.four;
thetaP.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
thetaP.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
thetaP.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;
/* finding the thetaH's for column 1 solution */
X1 = -J11;
X2 = -J21;
thetaH.one = 1.0;
thetaH.two = ( J24*X1 - J14*X2)/det;
thetaH.four = (-J22*X1 + J12*X2)/det;
thetaH.three = (-J31 - J32*thetaH.two)/J33;
X4 = -J41 - J42*thetaH.two;
X5 = -J51 - thetaH.three;
X6 = -J61 - J62*thetaH.two - thetaH.four;
thetaH.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
thetaH.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
thetaH.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

```

If joint angle 4 is within 10° of 0° then column 4 is used for $\underline{\alpha}$ and the following is done: (1) the particular solution θP associated with ${}^3I_{w1}^{\oplus} \dot{3}\underline{x}_w$ is calculated, and (2) the homogeneous solution θH associated with ${}^3I_{w1}^{\oplus} \dot{3}\underline{x}_w$ is calculated. If that condition is not present then column 1 is used for $\underline{\alpha}$ and the same thing is done. The variable $\det$ contains the determinant of the associated 3×3 nonsingular matrix ${}^3I_{w1}^{*-1}$ from (4.17).

```

NullSpaceOperator(thetaH,&thetaNullOp);
if(leastnorm)
    *dTheta = LeastNorm(thetaP,NullProjector(thetaP,thetaNullOp,thetaH));
    return(LeastNormVelCheck(dTheta));
else {
    *dTheta=LeastNorm(thetaP,NullProjector(thetaP,thetaNullOp,thetaH));
    GetGradientH(theta,Sin,Cos,CaseNum,&kKind,&gH);
    hom_sol = NullProjector(gH,thetaNullOp,thetaH);
    theDamper = fabs( (DotProduct(gH,thetaH))/(GetSize(gH)*GetSize(hom_sol)));
    theDamper = pow(theDamper,0.62);
    Kmaxmin (dThetaMax,*dTheta,hom_sol,&Kmax,&Kmin);
    if (Kmax > Kmin ) k = (kKind) ? Kmax : Kmin ;
    else
        return(1);
    dTheta->one      += k*theDamper*hom_sol.one ;
    dTheta->two      += k*theDamper*hom_sol.two ;
    dTheta->three     += k*theDamper*hom_sol.three ;
    dTheta->four      += k*theDamper*hom_sol.four ;
    dTheta->five      += k*theDamper*hom_sol.five ;
    dTheta->six       += k*theDamper*hom_sol.six ;
    dTheta->seven     += k*theDamper*hom_sol.seven ;
    return(0);}}
EndFunction

```

The function `NullSpaceOperator()` calculates the vector $(\text{thetaH}/\text{thetaH}^T\text{thetaH})$ and assigns the results to `thetaNullOp`.

If `(leastnorm)` is true optimization is not occurring and the least norm solution to $\dot{\mathbf{x}} = \mathbf{J} \dot{\boldsymbol{\theta}}$ is calculated (using ${}^3\mathbf{J}_w^\oplus {}^3\dot{\mathbf{x}}_w$) in the function `LeastNorm()` and the results assigned to `dTheta`, but before being returned a velocity check is done in `LeastNormVelCheck()`.

If `(leastnorm)` is false, then optimization is occurring. The least norm solution to $\dot{\mathbf{x}} = \mathbf{J} \dot{\boldsymbol{\theta}}$ is calculated using `LeastNorm()` and the results assigned to `dTheta`. The gradient of the optimization criterion $\mathbf{gH}$ is obtained using `GetGradientH()`, which also assigns the correct value to `kKind` so that the optimization criterion may be maximized or minimized. A homogeneous solution $\mathcal{P}(\nabla H)$ to $\dot{\mathbf{x}} = \mathbf{J} \dot{\boldsymbol{\theta}}$ is then calculated and the results are assigned to `hom_sol`, which needs to be scaled to avoid hardware limits on the joint velocities of the LTM, which is done in `KmaxKmin()`. However, $\mathbf{k}$ must be essentially damped depending on the size of the angle between the homogeneous solution and the gradient of the performance criterion. The absolute value of the cosine of that angle is computed and placed into the float `theDamper`. This value is then raised to a power of 0.62 and used to find `dTheta`.

If `(Kmax > Kmin)` is false then limits on the joint velocities cannot be met and the value 1 is returned by `InverseKinematics()`.

If `(Kmax > Kmin)` is true then commanded joint velocities are not excessive and whether or not to maximize or minimize the optimization criterion is determined by `kKind`. The structure `dTheta` is then filled up with the optimized very small but finite changes in the joint angles. These changes are such that the linear and angular velocities of the end-effector maintain the end-effector on a specified path, while at the same time the configuration of the manipulator is being optimized.

In order to compare the improved version of DEBA, to DEBA, the benchmark of number of operations in `InverseKinematics()` has been chosen. The operations in `InverseKinematics()` using the improved version break down as follows:

- (1) If column 1 is eliminated: number of multiplications is 83; number of divisions is 17; number of additions is 33; and the number of subtractions is 25.
- (2) If column 4 is eliminated: number of multiplications is 92; number of divisions is 17; number of additions is 33; and the number of subtractions is 32.

The operations for inverse kinematics using DEBA break down as follows:

- (1) If column 1 is eliminated: number of multiplications is 150; number of divisions is 29; number of additions is 71; and the number of subtractions is 63.

- (2) If column 4 is eliminated: number of multiplications is 163; number of divisions is 29; number of additions is 71; and the number of subtractions is 70.

When using the improved version there is approximately a 45% reduction in multiplications, a 40% reduction in divisions, a 54% reduction in additions, and a 55% reduction in subtractions.

5.3.1.12. JointLimit()

JointLimit() accepts a new set of joint angles and compares them to physical limits on the joints. If a joint limit would be exceeded by moving to the configuration commanded by the new joint angles, the arm is stopped and an error message is generated.

```

JointLimit (casenum,theta)
int          *casenum;
JOINT_ANGLES theta;{
char         joint[10];
char         theLimit[10];
int          LimitFlag = 0,theKind = joint_limit;
    if (theta.one > (.5235) ){
        LimitFlag = 1;
        stpcpy(joint,"1");
        CtoPstr(joint);
        stpcpy(theLimit,"120");
        CtoPstr(theLimit);}
    else if (theta.one < (-2.356) ) {
        LimitFlag = 1;
        stpcpy(joint,"1");
        CtoPstr(joint);
        stpcpy(theLimit,"-45");
        CtoPstr(theLimit);}
        *
        *
        *
    if (LimitFlag) {
        ErrorMessage(theKind,joint,theLimit);
        LTM->flags.robotics = 0; /* tell arm to stop */
        LTM->flags.jointLimit = 1;
        *casenum = 0;
        _closeall();}}
EndFunction

```

The three vertical stars (*) after CtoPstr() indicate ellipsis of the program: The if and else if statements shown for joint 1 are merely repeated for the other 6 joints. CtoPstr() converts a C type string to a pascal type string, and stpcpy() merely copies the desired string into the string variables **joint** and **theLimit**. These two functions, stpcpy() and CtoPstr(), are provided in the standard libraries of the LightSpeedC compiler. The strings

joint and theLimit are passed to the function ErrorMessage(), which expects pascal type strings due to the Macintosh environment the program is running in.

ErrorMessage() informs the operator that a joint limit has been reached when (LimitFlag) is true. In that case the flag LTM->flags.robotics is set to 0, which will ultimately cause the manipulator to stop; the global flag LTM->flags.jointLimit is set to 1 to be used by the shell program; the global variable casenum is reset to 0 in preparation for the next command to move the manipulator; and, finally, all files relating to the particular trajectory in which the joint limit occurred are closed.

5.3.1.13. KmaxKmin ()

A real scalar constant k must be found for (5.16) in order to minimize or maximize the performance criterion. The absolute value of this constant determines the rate at which optimization will occur. A larger value will optimize faster than a smaller value. Note, however, that choosing a larger value for k in order to optimize faster may result in a set of joint velocities that are impossible for at least one joint motor to attain. In such a case the end-effector will not follow the desired trajectory.

In order to avoid hardware limits on the joint velocities, Euler et al. [12] used the following method to determine the magnitude of the scalar constant k:

Let a set of maximum joint velocities determined by the joint motors be designated $\dot{q}_{\max}$, and likewise $-\dot{q}_{\max}$ for the minimum. These velocity vectors may then be used in equation (3.1), giving

$$\dot{q}_{\max} = \dot{q}_p + k\dot{q}_h \quad (5.17)$$

$$-\dot{q}_{\max} = \dot{q}_p + k\dot{q}_h \quad (5.18)$$

Using equations (5.17 and 5.18) we may solve for the upper and lower bounds to k for the ith joint in the following manner:

$$k_{\max_i} = \max \left(\frac{\dot{q}_{\max_i} - \dot{q}_{p_i}}{\dot{q}_{h_i}}, \frac{-\dot{q}_{\max_i} - \dot{q}_{p_i}}{\dot{q}_{h_i}} \right) \quad (5.19)$$

$$\text{and } k_{\min_i} = \min \left(\frac{\dot{\theta}_{\max_i} - \dot{\theta}_{p_i}}{\dot{\theta}_{h_i}}, \frac{-\dot{\theta}_{\max_i} - \dot{\theta}_{p_i}}{\dot{\theta}_{h_i}} \right) \quad (5.20)$$

where the letter i denotes the i th element in the associated vector. The object is to find the maximum k and minimum k for each joint, and then choose the smallest maximum or the largest minimum for the scalar constant k :

$$k_{\max} = \min (k_{\max_1}, k_{\max_2}, k_{\max_3}, k_{\max_4}, k_{\max_5}, k_{\max_6}, k_{\max_7}) \quad (5.21)$$

and

$$k_{\min} = \max (k_{\min_1}, k_{\min_2}, k_{\min_3}, k_{\min_4}, k_{\min_5}, k_{\min_6}, k_{\min_7}) \quad (5.22)$$

If k is chosen to be $k_{\max}$, then the optimization criterion will be maximized; if it is chosen to be $k_{\min}$, then it will be minimized. Choosing k in this manner will assure that requests which would exceed hardware limits on the joint velocities are never made.

If it should happen that $k_{\max} < k_{\min}$, then no scalar constant k is available to insure that velocity requests are attainable. In this case even the velocities requested by the least-norm solution $\underline{J}^+ \underline{\dot{x}}$ are not attainable due to limits on the joint motors.

```
KmaxKmin (tmax,tp,th,Kmax,Kmin)
JOINT_ANGLES    tmax,tp,th;
double          *Kmax,*Kmin; {
double          temp1,temp2;
  if(fabs(th.one)<.0000000001) th.one = .0000000001;
  if(fabs(th.two)<.0000000001) th.two = .0000000001;
  if(fabs(th.three)<.0000000001) th.three = .0000000001;
  if(fabs(th.four)<.0000000001) th.four = .0000000001;
  if(fabs(th.five)<.0000000001) th.five = .0000000001;
  if(fabs(th.six)<.0000000001) th.six = .0000000001;
  if(fabs(th.seven)<.0000000001) th.seven = .0000000001;
  temp1 = (tmax.one - tp.one)/th.one;
  temp2 = (-tmax.one - tp.one)/th.one;
  *Kmax = max(temp1,temp2);
  *Kmin = min(temp1,temp2);
  temp1 = (tmax.two - tp.two)/th.two;
  temp2 = (-tmax.two - tp.two)/th.two;
  *Kmax = min(max(temp1,temp2),*Kmax);
  *Kmin = max(min(temp1,temp2),*Kmin);
  *
  *
  *
  temp1 = (tmax.seven - tp.seven)/th.seven;
  temp2 = (-tmax.seven - tp.seven)/th.seven;
```

```

    *Kmax = min(max(temp1,temp2),*Kmax);
    *Kmin = max(min(temp1,temp2),*Kmin); }
EndFunction

```

A structure containing the maximum joint angles the manipulator is capable of moving in one iteration is passed to KmaxKmin() in **tmax**. The particular and homogeneous part of the joint velocity solution is passed in structures **tp** and **th** respectively.

The possibility of an element of the homogeneous solution contained in **th** being zero or virtually zero and thusly causing problems in division is avoided by comparing the absolute values of the elements to a very small number and, if the test is true for a particular element, setting that particular element to the small number. Then the maximum and minimum value of **k** for joint angle 1 are calculated and placed into **Kmax** and **Kmin** respectively. Thereafter the maximum and minimum value of **k** for each individual joint are placed in the temporary variables **temp1** and **temp2** respectively, and the minimum maximum is placed in **Kmax** and the maximum minimum is placed in **Kmin**.

5.3.1.14. LeastNorm()

This function simply accepts both a particular and homogeneous solution to ${}^3\dot{\mathbf{x}}_w = {}^3\mathbf{J}_w\dot{\mathbf{q}}$ and returns the least norm solution. It uses DEBM (${}^3\mathbf{J}_w^\oplus {}^3\dot{\mathbf{x}}_w$), but most of the calculations are done in NullSpaceOperator() and NullProjector().

```

struct JOINTS      LeastNorm(thetaP,thetaNull)
JOINT_ANGLES      thetaP,thetaNull;{
struct JOINTS      least_norm;
    least_norm.one =  thetaP.one      -  thetaNull.one;
    least_norm.two =  thetaP.two      -  thetaNull.two;
    least_norm.three = thetaP.three    -  thetaNull.three;
    least_norm.four =  thetaP.four     -  thetaNull.four;
    least_norm.five =  thetaP.five     -  thetaNull.five;
    least_norm.six  =  thetaP.six      -  thetaNull.six;
    least_norm.seven = thetaP.seven    -  thetaNull.seven;
    return(least_norm);}
EndFunction

```

The null space projection of a particular solution to ${}^3\dot{\mathbf{x}}_w = {}^3\mathbf{J}_w\dot{\mathbf{q}}$ is passed to LeastNorm() in the vector **thetaNull**. This is subtracted from the same particular solution in accordance with DEBM described in Chapter 3.

5.3.1.15. LeastNormVelCheck()

When manipulator motion is not being optimized velocity checking is done by simply comparing the requested change for each particular joint angle with the maximum change permissible for that angle. See the function KmaxKmin() for velocity checking when motion is being optimized.

```
int    LeastNormVelCheck(dTheta)
JOINT_ANGLES    *dTheta; {
    if (fabs(dTheta->one) > dThetaMax.one )
        return(1);
    else if (fabs(dTheta->two) > dThetaMax.two )
        return(1);
    else if (fabs(dTheta->three) > dThetaMax.three )
        return(1);
    else if (fabs(dTheta->four) > dThetaMax.four )
        return(1);
    else if (fabs(dTheta->five) > dThetaMax.five )
        return(1);
    else if (fabs(dTheta->six) > dThetaMax.six )
        return(1);
    else if (fabs(dTheta->seven) > dThetaMax.seven )
        return(1);
    else
        return(0); }
EndFunction
```

The requested change in joint angles is passed to LeastNormVelCheck() in the structure **dTheta**. The absolute value of each element of **dTheta** is compared to the corresponding element of the global structure **dThetaMax**, which contains the maximum allowable change in joint angles for one iteration. If a request is made to change any particular joint angle by an amount greater than the allowable maximum, the value 1 is returned, otherwise the value 0 is returned.

5.3.1.16. MaintainConstantVelocity()

Before looking at MaintainConstantVelocity() the reader must first read the discussions in GetDesiredPosition() and Accelerate(). After doing so it will be apparent that at time t_{acc} acceleration is complete and motion lies on a straight line segment (see Fig. 5-7) and we have

$$q = \Delta Ch + B \quad (5.23)$$

$$h = \frac{t}{T_i} \quad (5.24)$$

```

MaintainConstantVelocity (B,nt,delta_C,desired_pos,i)
HAND_POS_OR      *B,delta_C,*desired_pos;
int               nt,i;  {
double           h;
    h = ((double)i)/nt;
    desired_pos->x = (delta_C.x * h) + B->x;
    desired_pos->y = (delta_C.y * h) + B->y;
    desired_pos->z = (delta_C.z * h) + B->z;
    desired_pos->phi = (delta_C.phi * h) + B->phi;
    desired_pos->theta = (delta_C.theta * h) + B->theta;
    desired_pos->psi = (delta_C.psi * h) + B->psi;  }
EndFunction

```

The structures **B** and **delta_C**, corresponding to **B** and ΔC in Fig. 5-7, are passed to **MaintainConstantVelocity()** along with both the total number of steps **nt** in the particular path segment the end-effector is currently being moved along, and the current step **i**. The value of **h** is computed using (5.26) and placed in **h**. Then the desired position and the desired orientation (in Euler angles) are calculated and placed in the structure **desired_pos**, which is passed to the function **HandError()**.

5.3.1.17. NextNode ()

Referring to Fig. 5-7 we see that the next node of the end-effector, which would correspond to position **C** in Fig. 5-7, must be available before the manipulator starts accelerating to a new straight line trajectory. This next position is necessary in order to calculate ΔC .

```

NextNode (C)
HAND_POS_OR      *C;  {
HOMOGEN_TRANS hand; /* passed to GetEulerAngles */
    hand.pos1 = LTM->l.s.hand.pos.i;
    hand.pos2 = LTM->l.s.hand.pos.j;
    hand.pos3 = LTM->l.s.hand.pos.k;
    hand.s1   = LTM->l.s.hand.sliding.i;
    hand.s2   = LTM->l.s.hand.sliding.j;
    hand.s3   = LTM->l.s.hand.sliding.k;
    hand.n1   = LTM->l.s.hand.normal.i;
    hand.n2   = LTM->l.s.hand.normal.j;
    hand.n3   = LTM->l.s.hand.normal.k;
    hand.a1   = LTM->l.s.hand.approach.i;
    hand.a2   = LTM->l.s.hand.approach.j;
    hand.a3   = LTM->l.s.hand.approach.k;
    GetEulerAngles (hand,C);  }
EndFunction

```

The next node in the path has already been assigned to the structure **LTM->l.s.hand**, which is located in the global data buffer. The values for position and orientation in **LTM->l.s.hand** are then assigned to the structure **hand** in order to pass the data to **GetEulerAngles()**. **GetEulerAngles()** transforms the rotation matrix representation of the orientation contained in **hand** into an Euler angle representation and places the results into the structure **C**. Note that the structure **C** not only contains the orientation of the end-effector, but also its next position.

5.3.1.18. NullProjector()

NullProjector() is the null space projection vector function $\mathcal{P}(\underline{x})$ where

$$\mathcal{P}(\underline{x}) = \begin{pmatrix} \underline{x}^T \underline{x}_h \\ \underline{x}_h^T \underline{x}_h \end{pmatrix} \underline{x}_h. \quad (5.25)$$

It was shown in Chapter 3 that

$$\mathcal{P}(\underline{x}) = (\mathbf{I} - \mathbf{J}^+ \mathbf{J}) \underline{x}. \quad (5.26)$$

This function is used to project an arbitrary particular solution to $\dot{\underline{x}} = \mathbf{J} \underline{\theta}$ onto the null space of $\mathbf{J}$ and also to project the gradient of the optimization criterion onto the same null space to obtain a homogeneous solution to $\dot{\underline{x}} = \mathbf{J} \underline{\theta}$.

```

struct JOINTS      NullProjector(theta,thetaOp,thetaH)
  JOINT_ANGLES      theta,thetaH,thetaOp;{
double             dotProduct;
struct JOINTS      thetaNull;
  dotProduct = DotProduct (theta,thetaOperator);
  thetaNull.one = dotProduct*thetaH.one;
  thetaNull.two = dotProduct*thetaH.two;
  thetaNull.three = dotProduct*thetaH.three;
  thetaNull.four = dotProduct*thetaH.four;
  thetaNull.five = dotProduct*thetaH.five;
  thetaNull.six = dotProduct*thetaH.six;
  thetaNull.seven = dotProduct*thetaH.seven;
  return(thetaNull);}
EndFunction

```

The vector **thetaOp** is an operator equal to $(\underline{\theta}_H / \underline{\theta}_H^T \underline{\theta}_H)$ which is calculated in the function **NullSpaceOperator()**. The null space projection **thetaNull** of any vector of type **JOINT_ANGLES** passed in **theta** is then calculated using (5.25).

5.3.1.19. NullSpaceOperator()

NullSpaceOperator() calculates a vector which is part of the null space projection function $\mathbf{P}(\mathbf{x})$. This part is shown in bold type in equation (5.27).

$$\mathbf{P}(\mathbf{x}) = \left(\frac{\mathbf{x}^T \mathbf{x}_h}{\mathbf{x}_h^T \mathbf{x}_h} \right) \mathbf{x}_h. \quad (5.27)$$

```
NullSpaceOperator(thetaH,thetaN)
JOINT_ANGLES    thetaH,*thetaN;{
double          dotProduct;
    dotProduct = DotProduct (thetaH,thetaH);
    thetaN->one =  thetaH.one/dotProduct;
    thetaN->two =  thetaH.two/dotProduct;
    thetaN->three = thetaH.three/dotProduct;
    thetaN->four =  thetaH.four/dotProduct;
    thetaN->five =  thetaH.five/dotProduct;
    thetaN->six =   thetaH.six/dotProduct;
    thetaN->seven = thetaH.seven/dotProduct;}
EndFunction
```

The vector **thetaH** is passed to NullSpaceOperator() and the results of the bolded part of equation (5.27) is calculated and assigned to **thetaN**.

5.3.1.20. ProjectionElements()

At various times it is necessary to project vectors from the base coordinate frame of the manipulator to its third coordinate frame and vice-versa. In order to do this elements of a projection matrix **P** must be available at each new iteration of the positioning function GetDesiredPosition(). These elements can be obtained from the rotation submatrix of the transformation matrix $\mathbf{T}_3^0$, which results from multiplication of the first three **A** matrices in Appendix B.

$$\mathbf{T}_3^0 = \mathbf{A}_1^0 \mathbf{A}_2^1 \mathbf{A}_3^2 \quad (5.28)$$

$$\mathbf{T}_3^0 = \begin{bmatrix} c_1 c_2 c_3 - s_1 s_3 & c_1 s_2 & c_1 c_2 s_3 + s_1 c_3 & a_2 c_1 c_2 \\ s_1 c_2 c_3 + c_1 s_3 & s_1 s_2 & s_1 c_2 s_3 - c_1 c_3 & a_2 s_1 c_2 \\ -s_2 c_3 & c_2 & -s_2 s_3 & -a_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.29)$$

from which we obtain

$$\mathbf{P}_3^0 = \begin{bmatrix} c_1 c_2 c_3 - s_1 s_3 & c_1 s_2 & c_1 c_2 s_3 + s_1 c_3 \\ s_1 c_2 c_3 + c_1 s_3 & s_1 s_2 & s_1 c_2 s_3 - c_1 c_3 \\ -s_2 c_3 & c_2 & -s_2 s_3 \end{bmatrix}. \quad (5.30)$$

which is used to project elements from the third coordinate frame onto the base coordinate frame.

```

ProjectionElements (Cos,Sin,P)
JOINT_ANGLES      Cos,Sin;
MATRIX            *P; {
double            C1C2,S1C2;
    C1C2 = Cos.one * Cos.two;
    S1C2 = Sin.one * Cos.two;
    P->e11 = (C1C2 * Cos.three) - (Sin.one * Sin.three);
    P->e12 = Cos.one * Sin.two;
    P->e13 = (C1C2 * Sin.three) + (Sin.one * Cos.three);
    P->e21 = (S1C2 * Cos.three) + (Cos.one * Sin.three);
    P->e22 = Sin.one * Sin.two;
    P->e23 = (S1C2 * Sin.three) - (Cos.one * Cos.three);
    P->e31 = -Sin.two * Cos.three;
    P->e32 = Cos.two;
    P->e33 = -Sin.two * Sin.three; }
EndFunction

```

The sine and cosine of the current joint angles are passed to ProjectionElements() in the structures Sin and Cos respectively. These are then used in accordance with (5.30) to calculate the projection matrix P. Recall that even though this projection matrix is used to project vectors from the third coordinate frame onto the base coordinate frame, its transpose results in a projection matrix to project vectors from the base coordinate frame onto the third coordinate frame; therefore, new projection elements do not have to be calculated to go from the base to the third coordinate frame.

5.3.1.21. ProjectVector()

Using the projection elements calculated in ProjectionElements(), ProjectVector() projects the elements of a three element vector from the base coordinate frame onto the third coordinate frame or vice versa.

```
ProjectVector (v1,v2,v3,P,direction)
double          *v1,*v2,*v3;
MATRIX          P;
FLAG            direction;  {
double          x,y,z;
    x = *v1;
    y = *v2;
    z = *v3;
    if (direction EQUALS 0)  {
        *v1 = (P.e11 * x) + (P.e12 * y) + (P.e13 * z);
        *v2 = (P.e21 * x) + (P.e22 * y) + (P.e23 * z);
        *v3 = (P.e31 * x) + (P.e32 * y) + (P.e33 * z);    }
    else if (direction EQUALS 3 )  {
        *v1 = (P.e11 * x) + (P.e21 * y) + (P.e31 * z);
        *v2 = (P.e12 * x) + (P.e22 * y) + (P.e32 * z);
        *v3 = (P.e13 * x) + (P.e23 * y) + (P.e33 * z);    }}
EndFunction
```

Projection elements are passed to ProjectVector() in the structure P, and the three elements of the vector to be projected are passed in the variables v1,v2, and v3. The flag direction determines in which direction the projection will occur. Note that the only difference between projecting from the third coordinate frame to the base coordinate frame and projecting from the base coordinate frame to the third coordinate frame is that the transpose of the projection matrix is used. Therefore, ProjectionElements() need calculate these elements only once for each iteration of GetDesiredPosition().

5.3.1.22. SendCommand ()

SendCommand() loads the global data buffer with the next set of joints angles the manipulator should be configured to. These are loaded only during actual motion and not during simulations.

```
SendCommand (theta)
JOINT_ANGLES    theta;  {
    THETA_1      = theta.one;
    THETA_2      = theta.two;
    THETA_3      = theta.three;
    THETA_4      = theta.four;
    THETA_5      = theta.five;
```

```

    THETA_6    = theta.six;
    THETA_7    = theta.seven; }
EndFunction

```

The next set of joint angles the manipulator should configure itself to is passed in the structure **theta**. These values are then placed in the global data buffer. Appendix G shows that during compilation the terms **THETA_1**, **THETA_2**, et cetera are replaced with the structure elements **LTM->l.s.shld.setpnnts.pitch_pos**, **LTM->l.s.shld.setpnnts.yaw_pos**, et cetera.

5.3.1.23. SubtractVectors1()

SubtractVectors1() subtracts two vectors which both contain the position and orientation of the end-effector in Cartesian coordinates and Euler angles respectively. It is used in the function **GetDesiredPosition()** to calculate the differences **delta_B** and **delta_C**. These differences correspond to ΔB and ΔC in Fig. 5-7.

```

SubtractVectors1 (c,b,delta)
HAND_POS_OR      *c,*b,*delta; {
    if ((fabs(c->phi - b->phi)) > PI){
        c->phi -= PI;
        c->theta = -c->theta;}
    delta->x      = c->x - b->x;
    delta->y      = c->y - b->y;
    delta->z      = c->z - b->z;
    delta->phi    = c->phi - b->phi;
    delta->theta  = c->theta - b->theta;
    delta->psi    = c->psi - b->psi; }
EndFunction

```

5.3.1.24. TrigValues()

TrigValues() simply obtains the sines and cosines of the current set of joint angles.

```

TrigValues (Cos,Sin,theta)
JOINT_ANGLES     *Cos,*Sin,theta; {
    Cos->one = cos(theta.one);
    Sin->one = sin(theta.one);
    Cos->two = cos(theta.two);
    Sin->two = sin(theta.two);
    Cos->three = cos(theta.three);
    Sin->three = sin(theta.three);
    Cos->four = cos(theta.four);
    Sin->four = sin(theta.four);
    Cos->five = cos(theta.five);
    Sin->five = sin(theta.five);
}

```

```

    Cos->six = cos(theta.six);
    Sin->six = sin(theta.six);
    Cos->seven = cos(theta.seven);
    Sin->seven = sin(theta.seven); }
EndFunction

```

5.3.1.25. UpdateJointAngles()

UpdateJointAngles() takes the requested change in joint angles calculated in the function InverseKinematics() and adds it to the current set of joint angles.

```

UpdateJointAngles (dA,theta)
JOINT_ANGLES      dA,*theta; {
    theta->one      = theta->one + dA.one;
    theta->two       = theta->two + dA.two;
    theta->three     = theta->three + dA.three;
    theta->four      = theta->four + dA.four;
    theta->five      = theta->five + dA.five;
    theta->six       = theta->six + dA.six;
    theta->seven     = theta->seven + dA.seven; }
EndFunction

```

5.3.1.26. VelocityLimit()

This function simply notifies the operator when a velocity limit has been reached.

```

VelocityLimit(){
int    theKind = velocity_limit;
    ErrorMessage(theKind,"\p","\p");}
EndFunction

```

5.3.1.27. WristError()

The function InverseKinematics() uses the difference between the current and desired position and orientation of the wrist (which is termed the wrist error) to calculate the small changes in joint angles necessary to move the end-effector on a specified trajectory. WristError() calculates this error in the following way:

The position of the end-effector is given by a vector $\underline{r}$ extending from the origin of the seventh coordinate frame to the origin of the coordinate system assigned to the end-effector. Because the LTM has a spherical wrist the angular velocity for the end-effector is the same as that for the wrist. And recalling that the linear velocity $\underline{v}$ of a vector being

rotated at some angular velocity ω is the cross product of the angular velocity with the vector, we can state the following

$$\omega_{\text{hand}} = \omega_{\text{wrist}} \quad (5.31)$$

$$v_{\text{hand}} = v_{\text{wrist}} + \omega_{\text{wrist}} \times \mathbf{l} \quad (5.32)$$

The vector $\mathbf{l}$ can be expressed as the length d_7 of the most distal link (see Fig.4-1) multiplied by the approach vector z_{hand} of the end-effector, which is a unit vector. Using equations (5.31 and 5.32) we have

$$\omega_{\text{wrist}} = \omega_{\text{hand}} \quad (5.33)$$

$$v_{\text{wrist}} = v_{\text{hand}} - d_7(\omega_{\text{hand}} \times z_{\text{hand}}) \quad (5.34)$$

We may multiply equations (5.33 and 5.34) by a differential time and express them in terms of infinitesimal rotations and displacements.

The results of (5.2) are only correct for infinitesimal time periods dt . If, however, the iteration time for `GetDesiredPosition()` is small enough so that the resulting error in the results of (5.2) are tolerable, then we may obtain a set of very small but finite rotations and displacements that are also tolerable using the functions `Accelerate()` and `MaintainConstantVelocity()`. The function `HandError()` then transforms these small rotations (which were in Euler angles) to a set of rotations associated with the angular velocity of the hand, and these are used in `WristError()`.

Note that the value for d_7 would have to be changed for different types of end-effectors.

```
WristError (error,c,w)
INCREMENTS      error;
HOMOGEN TRANS    c;
INCREMENTS      *w; {
    w->dThetaX = error.dThetaX;
    w->dThetaY = error.dThetaY;
    w->dThetaZ = error.dThetaZ;
    w->dx = error.dx + Constant_d7*(error.dThetaY*c.a3
        - error.dThetaZ*c.a2);
    w->dy = error.dy + Constant_d7*(error.dThetaZ*c.a1
        - error.dThetaX*c.a3);
    w->dz = error.dz + Constant_d7*(error.dThetaX*c.a2
        - error.dThetaY*c.a1); }
EndFunction
```

Desired increments in the end-effector translation and angular rotation are passed to WristError() in the structure error. The unit vector in the approach direction of the end-effector is passed in the structure c. Constant_d7 is a constant defined to be the value d_7 of Fig. 4-1. The operations of (5.33) and (5.34) are then performed for the increments, which is alright to do as long as they are small enough.

5.3.2. Joint Space Functions

The joint space functions will be treated in the same manner the Cartesian space functions were, that is, GetJointConfig() will be discussed first, and then the remaining functions will be taken up in alphabetical order. Any functions that are used in both space motions are treated in section 5.3.1.

5.3.2.1. GetJointConfig()

The joint space functions are all called from the function GetJointConfig(). GetJointConfig() is the joint space counterpart to the Cartesian space function GetDesiredPosition(). Like GetDesiredPosition(), GetJointConfig() is patterned after Paul's trajectory scheme and therefore whatever is said about Paul's scheme in general in section 5.3.1.1. is applicable to GetJointConfig() too.

```
case 0:
    casenum = 1;
    TheLastNode = LTM->l.s.hand.LastNode;
    NextJointNode(&JC);
    GetInitialAngles (&theta);
    TrigValues (&Cos,&Sin,theta);
    ProjectionElements (Cos,Sin,&P);
    ViewArm0 = ForwardPosKinematics (&current_pos,Cos,Sin,P);
    JA = theta;
    JB = JA;
    delta_JB = NULL_STRUCT;
    SubtractVectors2 (pJC,pJB,&delta_JC);
    GetTimeSegment(&t,JB,JC);
    nt = t/delta_t + .5;
    na = nt*(LTM->l.s.hand.accFactor);
    i = -na;
    JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
    JointAcc (JA,na,two_na,f1,f2,&theta,i);
    firstIteration = 1;
    break;
```

Just before the first iteration the first node was entered into the global data buffer and the manipulator is sitting at rest at the zeroth node. Case 0 is always chosen during the first iteration of `GetJointConfig()` and at no other time. Setting `casenum` to 1 assures that on the next iteration case 0 will be bypassed. Whether or not the first node is also the last node is obtained from the global data buffer and stored into `TheLastNode`.

`NextJointNode()` assigns the data for node 1 to `JC` which is a structure (as are `JA` and `JB`) which contains the current joint angles of the manipulator. Note that at first glance it would seem that to be in accordance with Fig. 5-7 `NextJointNode()` should assign node 1 to `JB`, however, because the manipulator starts from rest and must accelerate, it must be that `JB` is the same position as `JA`, and node 1 must be assigned to `JC`.

The functions `GetInitialAngles()`, `TrigValues()`, and `ProjectionElements()` manipulate the current joint angles so that forward kinematics can be performed on them in `ForwardPosKinematics()`: `GetInitialAngles()` obtains the initial angles of the manipulator so that the variable `theta` can be passed to `TrigValues()`, which assigns the sine and cosine of the manipulator's joint angles to the structures `Sin` and `Cos`, which are then passed to `ProjectionElements()`. `ProjectionElements()` simply puts into `P` elements that are used to project or transform the sine and cosine of the joint angles from the base coordinate frame to the third coordinate frame and back again. `ForwardPosKinematics()` returns the world Cartesian coordinates of the manipulator arm to `ViewArm0`, which is passed to graphics routines.

The variables `delta_JB` and `delta_JC` are ΔB and ΔC in Fig.5-7. Because `JA` and `JB` are the same in this case, all the elements in `delta_JB` have been set to zero using a null structure called `NULL_STRUCT`, whose elements are all zero.

The total time to move from `JB` to `JC` is calculate in `GetTimeSegment()` and stored in `t` and then converted into the total number of steps `nt` in time `t`. The duration of acceleration t_a has been specified as a fraction of the total time and stored in the global variable `LTM->l.s.hand.accFactor`, which is then used to obtain the number of steps `na` in acceleration time t_a .

If the first node is also the last node then we must subtract two times `na` from `nt`, otherwise we only subtract one times `na` from `nt`. This is necessary of we want the actual time of travel to match the specified time of travel. As shown in Fig.5-6, in order to begin accelerating we had to start at a time $-t_a$ before reaching B. In the midrange of motion this causes no problems timewise, but initially we see that in order to begin accelerating from a dead stop at the zeroth node we must, nevertheless, start at a time $-t_a$ before reaching B, but since we are already at B we simply subtract the extra time from the total time, which has the effect of decreasing the duration for constant velocity motion. Later, when

stopping motion is described it will be seen that an extra amount of time equal to t_a is required to stop smoothly. Thusly, if the first node is also the last we must subtract two times na from nt , but if it is not the last we only subtract one times na from nt and wait for the last node to subtract the specific na for that node from the nt for that node, as they may be different from those for node 1.

JointAccSetup() does some initial computations for the function JointAcc(), which calculates the desired joint angles and assigns them to the structure **theta**.

```

case 1:
    if ( i >= na+1 AND i < nt-na )
        ConstJointVel(pJB,nt,delta_JC,&theta,i);
    else
        JointAcc (JA,na,two_na,f1,f2,&theta,i);
    break;
case 2:
    JointAcc (JA,na,two_na,f1,f2,&theta,i);
    if (i == na) {
        LTM->flags.robotics = 0;
        casenum = 0; }
    break; }

```

Cases 1 and 2 simply provide the structure **theta** with the next desired set of joint angles. Case 2, however, has the added jobs of notifying higher level code via the global data buffer that the last node has been reached and of setting **casenum** to zero in preparation for the next move.

```

if(JointVelCheck(&firstIteration,&theta)) {
    VelocityLimit();
    LTM->flags.robotics = 0; /* tell arm to stop */
    LTM->flags.VelLimit = 1;
    casenum = 0; }
TrigValues (&Cos,&Sin,theta);
ProjectionElements (Cos,Sin,&P);
ViewArm = ForwardPosKinematics(&current_pos,Cos,Sin,P);
JointLimit (&casenum,theta);
if (!LTM->flags.simulate)
    SendCommand (theta);

```

If joint velocities are not exceeded, new projection elements are calculated and used by ForwardPosKinematics() to return the world coordinates of the manipulators elbow, wrist, and end-effector in the variable **ViewArm**, which is used in graphics routines.

Whether or not a joint limit has been reached is then checked by JointLimit(). If all of the joints are within the specified joint limits and a simulation is not occurring, then the command to actually move the manipulator is sent by SendCommand(). If LTM->flags.simulate is anything but a 0, the variable theta is simply not passed to the global data buffer, which is a further precaution to ensure that the manipulator does not move during simulation.

```

++i;
if ((i == nt-(na+1)) AND !TheLastNode)
    LTM->flags.load_next_pos = 1;
if (i == nt-na AND TheLastNode) {
    casenum = 2;
    JA = theta;
    JB = JC;
    SubtractVectors2(pJA,pJB,&delta_JB);
    delta_JC = NULL_STRUCT;
    JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
    i = -na; }
else if (i == nt-na) {
    JB = JC;
    NextJointNode(&JC);
    JA = theta;
    GetTimeSegment(&t,JB,JC);
    SubtractVectors2(pJA,pJB,&delta_JB);
    SubtractVectors2 (pJC,pJB,&delta_JC);
    nt = t/delta_t + .5;
    na = nt*(LTM->l.s.hand.accFactor); /* number of steps in na */
    TheLastNode = LTM->l.s.hand.LastNode;
    if (TheLastNode) nt = nt - na;
    JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
    i = -na;}}
EndFunction

```

The counter *i* is increased by one. This counter keeps track of the number of steps between each node, not over the whole path.

If if ((i == nt-(na+1)) AND !TheLastNode) is true, LTM->flags.load_next_pos is set so that the next node can be loaded into the global data buffer before the next iteration of GetJointConfig(), when acceleration will begin again. At $t = -t_a$, which occurs when $i = nt-na$, the values of JC are assigned to JB, the next node is assigned to JC, and the current set of joint angles is assigned to JA all in accordance with Fig. 5-7. A new time between nodes is calculated in GetTimeSegment(). Because we are not approaching the last node we must calculate both delta_JB and delta_JC, which is done by calling SubtractVectors2() twice. We now fill up necessary

variables that will be used later, as was done in case 0. Note, however, that if the node just loaded is the last node, we subtract **na** from **nt** as was mentioned in the discussion of case 0. **JointAccSetup()** then assigns values to **f1**, **f2**, and **two_na** to be used later by **JointAcc()**, and finally **i** must be reset to **-na**, in accordance with Fig. 5-7.

If (**i == nt-na AND TheLastNode**) is true the final configuration of the manipulator is being approached. This is a special case which must be treated differently. Referring to Fig. 5-6 it can be seen that in order to come to a smooth stop the new node assigned to **JC** must actually be the node previously loaded. Conceptually, we give **JC** the same node twice. In actual practice, after assigning the old values of **JC** to **JB**, instead of loading the same node into **JC** again, we simply assign the null structure **NULL_STRUCT** to **delta_JC**. The variables that had to be filled up at time $-t_a$ when the end-effector was not approaching its final position and orientation do not have to be filled up again, since they do not change; however, because **delta_JB** and **delta_JC** have changed, **JointAccSetup()** is called. Additionally **casenum** is set to 2. Finally **i** must be reset to **-na** as before.

5.3.2.2. ConstantJointVel()

Refer to **MaintainConstantVelocity()** for a discussion.

```

ConstantJointVel  (B,nt,delta_C,desiredAngles,i)
JOINT_ANGLES      *B,delta_C,*desiredAngles;
int               nt,i;{
double           h;
    h = ((double)i)/((double)nt);
    desiredAngles->one      = (delta_C.one * h) + B->one;
    desiredAngles->two      = (delta_C.two * h) + B->two;
    desiredAngles->three    = (delta_C.three * h) + B->three;
    desiredAngles->four     = (delta_C.four * h) + B->four;
    desiredAngles->five     = (delta_C.five * h) + B->five;
    desiredAngles->six      = (delta_C.six * h) + B->six;
    desiredAngles->seven    = (delta_C.seven * h) + B->seven;}
EndFunction

```

The structures **B** and **delta_C**, corresponding to **B** and ΔC in Fig. 5-7, are passed to **ConstantJointVel()** along with both the total number of steps **nt** in the particular path segment the joint angles are currently being moved along, and the current step **i**. The value of **h** is computed using (5.26) and placed in **h**. Then the desired joint angles are computed and placed in **desiredAngles**.

5.3.2.3. GetTimeSegment()

This calculates the time between nodes. It depends on the maximum speed of the slowest joint motor. It also depends on what was specified as maximum speed in the dialog SetUp RunFeatures (for Joint Space).

```
GetTimeSegment(T,JC,JC)
float      *T;
JOINT_ANGLES JC,JC;{
JOINT_ANGLES t;
    t.one      =      (fabs(JC.one   -   JB.one))/MaxSpeed.one;
    t.two      =      (fabs(JC.two   -   JB.two))/MaxSpeed.two;
    t.three    =      (fabs(JC.three -   JB.three))/MaxSpeed.three;
    t.four     =      (fabs(JC.four  -   JB.four))/MaxSpeed.four;
    t.five     =      (fabs(JC.five  -   JB.five))/MaxSpeed.five;
    t.six      =      (fabs(JC.six   -   JB.six))/MaxSpeed.six;
    t.seven    =      (fabs(JC.seven -   JB.seven))/MaxSpeed.seven;
    *T = (float) (max(t.one,t.two));
    *T = (float) (max(t.three,*T));
    *T = (float) (max(t.four,*T));
    *T = (float) (max(t.five,*T));
    *T = (float) (max(t.six,*T));
    *T = (float) (max(t.seven,*T));}
EndFunction
```

The maximum speed for each joint is available in the structure MaxSpeed. The values in MaxSpeed depend on the motor ratings and on the maximum speed specified in the Set Up Run Features dialog for Joint space motion. The total time to go from each old joint value to each new joint value is then calculated and put into the structure t. The maximum time is then chosen for the time T to move all joints from their current values to the new values.

5.3.2.4. JointAcc()

The discussion in section 5.3.1.2. should be consulted for additional information regarding this function.

```
JointAcc  (JA,na,two_na,f1,f2,desiredAngles,i)
JOINT_ANGLES JA,f1,f2,*desiredAngles;
int      na,two_na,i;  {
double      h,h2,h3,two_h2,two_h3;
    h = ( (double)i + (double)na)/((double)two_na);
```

```

h2 = h*h;
h3 = h2*h;
two_h2 = h2 + h2;
twoh2_h3 = two_h2 - h3;
desiredAngles->one = (f1.one*twoh2_h3 - f2.one)*h + JA.one;
desiredAngles->two = (f1.two*twoh2_h3 - f2.two)*h + JA.two;
desiredAngles->three = (f1.three*twoh2_h3 - f2.three)*h + JA.three;
desiredAngles->four = (f1.four*twoh2_h3 - f2.four)*h + JA.four;
desiredAngles->five = (f1.five*twoh2_h3 - f2.five)*h + JA.five;
desiredAngles->six = (f1.six*twoh2_h3 - f2.six)*h + JA.six;
desiredAngles->seven = (f1.seven*twoh2_h3 - f2.seven)*h + JA.seven;}
EndFunction

```

JointAcc() uses (5.2) to place into the structure **desiredAngles** the next desired set of joint angles of the manipulator. The variable **JA** is the point in joint space at which acceleration begins in order to execute a smooth change from a current straight line trajectory, for each joint, to another straight line trajectory defined by points B and C in Fig. 5-7. The variable **two_na** is merely twice the variable **na**, which is the number of steps in t_a , the time to accelerate to maximum velocity. The function JointAccSetup() provides **two_na**, **f1** and **f2** for JointAcc(). These three variables change only when a new node is loaded into C. Inspection of AccSetup() will reveal that **f1** corresponds to $(\Delta C_{T_i}^{acc} + \Delta B)$ of (5.2) and **f2** corresponds to $2\Delta B$ of the same equation. The variable **twoh2_h3** corresponds to $(2 - h)h^2$. JointAcc(), therefore, is nothing more than the implementation of equation (5.2).

5.3.2.5. JointAccSetup()

Inspection of equation (5.2) will show that some parts of it can be precalculated and passed to JointAcc() virtually as constants during the entire time of acceleration.

```

JointAccSetup (na,nt,two_na,delta_JC,delta_JB,f1,f2)
int
    na,nt,*two_na;
JOINT_ANGLES delta_B,delta_C,*f1,*f2;{
double
    temp;
    *two_na = na + na;
    temp = ((double)na)/((double) nt);
    f1->one = (delta_JC.one * temp) + delta_JB.one;
    f1->two = (delta_JC.two * temp) + delta_JB.two;
    f1->three = (delta_JC.three * temp) + delta_JB.three;
    f1->four = (delta_JC.four * temp) + delta_JB.four;
    f1->five = (delta_JC.five * temp) + delta_JB.five;
    f1->six = (delta_JC.six * temp) + delta_JB.six;

```

```

f1->seven      =      (delta_JC.seven * temp) + delta_JB.seven;
f2->one        =      delta_JB.one + delta_JB.one;
f2->two        =      delta_JB.two + delta_JB.two;
f2->three      =      delta_JB.three + delta_JB.three;
f2->four       =      delta_JB.four + delta_JB.four;
f2->five       =      delta_JB.five + delta_JB.five;
f2->six        =      delta_JB.six + delta_JB.six;
f2->seven      =      delta_JB.seven + delta_JB.seven;
EndFunction

```

JointAccSetup() is passed the variables **na**, **nt**, **delta_JC**, and **delta_JB**, and calculates values for variables **f1**, **f2** and **two_na**. As described in section GetDesiredPosition() the variable **na** is the number of steps in acceleration time t_a and the variable **nt** is the total time T_i for the arm to move from node to node. The variables **delta_JC** and **delta_JB** correspond to ΔC and ΔB of Fig. 5-7. This function is called only at time $T_i - t_a$ when acceleration starts and only if a new node is being loaded. Comparison of (5.2) and the variable structures **f1** and **f2** will reveal the following: **f1** corresponds to $(\Delta C \frac{t_{acc}}{T_i} + \Delta B)$, and **f2** corresponds to $2\Delta B$. The variables **f1**, **f2** and **two_na** are therefore precalculated for JointAcc(), saving computer time for other necessary calculations.

5.3.2.6. JointVelCheck()

```

int  JointVelCheck(firstIteration,Angles)
int          *firstIteration;
JOINT_ANGLES      *Angles;{
static  JOINT_ANGLES      dTheta,oldAngles;
int          velerr = 0;

    if(*firstIteration){
        oldAngles.one = ThetaInit[1];
        oldAngles.two = ThetaInit[2];
        oldAngles.three = ThetaInit[3];
        oldAngles.four = ThetaInit[4];
        oldAngles.five = ThetaInit[5];
        oldAngles.six = ThetaInit[6];
        oldAngles.seven = ThetaInit[7];
        *firstIteration = 0;}
    SubtractVectors2(Angles,&oldAngles,&dTheta);
    oldAngles = *Angles;
    if (fabs(dTheta.one) > dTMax.one )
        return(1);
    else if (fabs(dTheta.two) > dTMax.two )
        return(1);
    else if (fabs(dTheta.three) > dTMax.three )
        return(1);

```

```

    else if (fabs(dTheta.four) > dTMax.four )
        return(1);
    else if (fabs(dTheta.five) > dTMax.five )
        return(1);
    else if (fabs(dTheta.six) > dTMax.six )
        return(1);
    else if (fabs(dTheta.seven) > dTMax.seven )
        return(1);
    else
        return(0);}
EndFunction

```

To check whether or not joint velocities are being exceeded the current joint angles are compared to the proposed joint angles. The current angles have been stored in the structure `oldAngles`.

If `firstIteration` is true then `oldAngles` are assigned the current joint angles in `ThetaInit[]`. The difference in the current angles and proposed angles is calculated in `SubtractVectors2()` and put into the structure `dTheta`. The elements in `dTheta` are compared to the maximum allowable values in `dTMax`. If a joint velocity would be exceeded 1 is returned, otherwise 0 is returned.

5.3.2.7. NextJointNode()

`NextJointNode()` simply places into the structure `JC` the next node in joint space.

```

NextJointNode(JC)
JOINT_ANGLES *JC;{
    JC->one = LTM->l.s.joint.J1;
    JC->two = LTM->l.s.joint.J2;
    JC->three = LTM->l.s.joint.J3;
    JC->four = LTM->l.s.joint.J4;
    JC->five = LTM->l.s.joint.J5;
    JC->six = LTM->l.s.joint.J6;
    JC->seven = LTM->l.s.joint.J7;}
EndFunction

```

5.3.2.8. SubtractVectors2()

This function is self-explanatory.

```

SubtractVectors2 (c,b,delta)
JOINT_ANGLES *c,*b,*delta;{
    delta->one = c->one - b->one;
    delta->two = c->two - b->two;

```

```
delta->three = c->three - b->three;  
delta->four  = c->four - b->four;  
delta->five  = c->five - b->five;  
delta->six   = c->six - b->six;  
delta->seven = c->seven - b->seven;}
```

EndFunction

CHAPTER 6

SIMULATIONS

6.1. Introduction

Graphical simulations were performed to test the robotic control scheme developed for the LTM. The simulations illustrate the ability of the control scheme to utilize the redundancy of the LTM for improved performance. Smooth changes in trajectory during the transition between two path segments is also illustrated. Results of each simulation are shown by a graphical representation of the manipulator, a plot of the seven joint angles with respect to time, and a plot of the Cartesian coordinates of the end-effector's position with respect to time. As an aid to understanding the simulations Figures 6-1 and 6-2 have been included. Figure 6-1 shows the LTM in the reference configuration for joint angles. Figure 6-2 shows the LTM in its anthropomorphic stance and defines the top, front, and side views of the LTM for these simulations. The anthropomorphic stance is the starting configuration for each simulation described below except the last, which demonstrates a smooth change in trajectory when the arm passes *through* the anthropomorphic stance.

6.2. Holding the End-Effector Stationary While Improving a Performance Criterion

A manipulator which is holding its end-effector stationary while performing a task may be required to change its configuration due to objects moving about in the environment, or the need for improved mechanical advantage. Figure 6-3 shows the results of a simulation in which the LTM has been commanded to improve a performance criterion while holding the end-effector stationary. The plot of the hand's position shows that it does not change from the beginning to the end of the trajectory period. The plot of the joint angles, however, shows that the configuration of the arm is changing. The performance criterion used is $(\theta_4 - 90)^2 + (\theta_7)^2$. When this criterion is minimized the elbow goes out to the side, enabling the LTM to avoid an object which might otherwise collide with it. The second term prevents joint seven from hitting a joint limit.

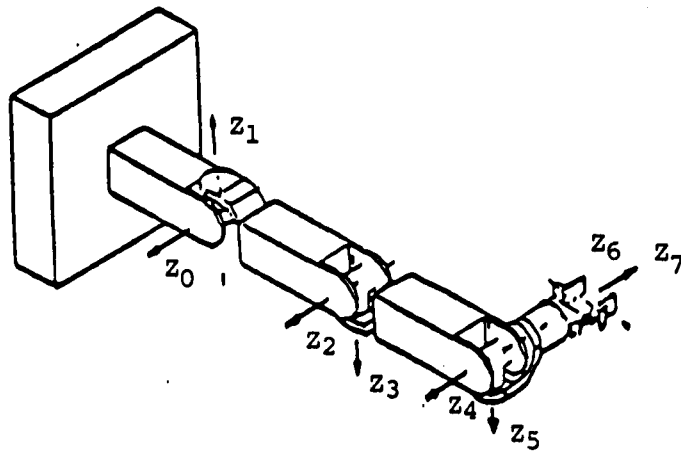


Figure 6-1: Reference configuration of LTM for simulations
(Courtesy I&C Division ORNL)

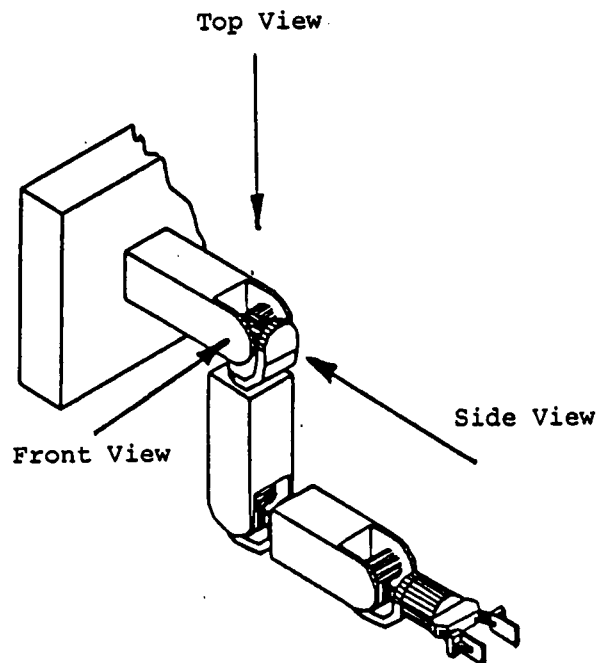
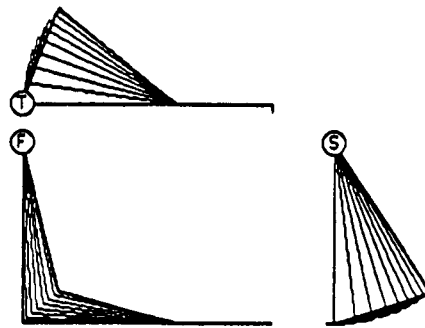
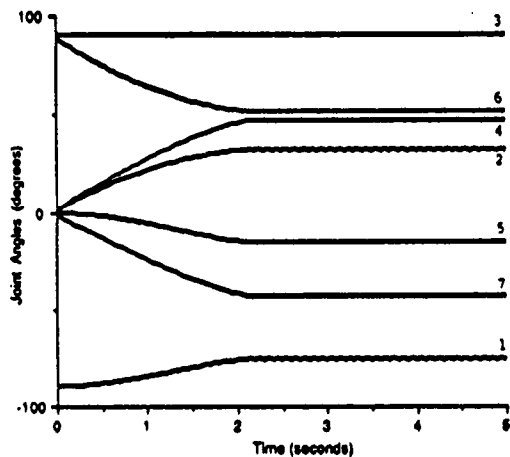


Figure 6-2: Anthropomorphic stance defining top, front, and side
views for simulations. (arm picture courtesy I&C Division ORNL)



Path Data:

Number of Nodes: 1

Starting Config: $\underline{Q} = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion: $H = (\theta_4 - 90^\circ)^2 + (\theta_7)^2$

Node #1:

Move Time : 5 seconds

$\Delta T = .05$ seconds

Acceleration Time: .5 seconds

Position of Hand: $\underline{p} = [32 \ -23 \ 0]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

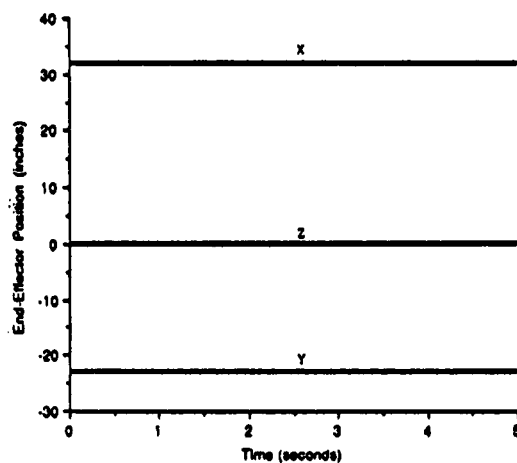


Figure 6-3: Improving a performance criterion while holding end-effector stationary
(T = top view F = front view S = side view)

6.3. Avoiding an Obstacle

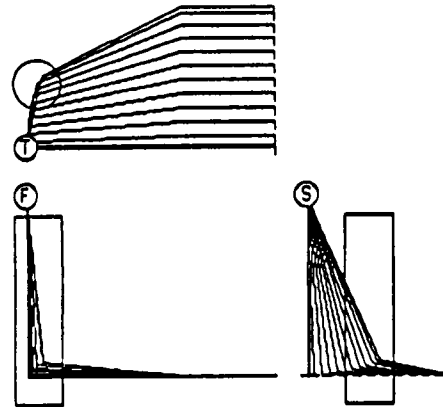
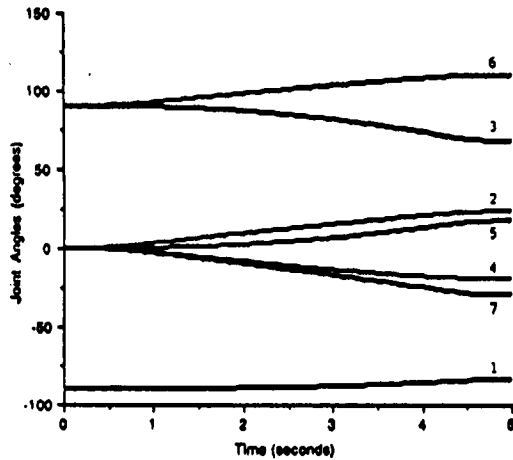
A simulation was performed to demonstrate utilization of redundancy to avoid collision of the LTM with an obstacle. A graphic representing a pole was drawn on the computer screen to represent an object in the manipulator's environment. Figure 6-4 shows the manipulator on a straight line trajectory with a fixed orientation. The starting configuration of the arm was the anthropomorphic stance shown in Fig. 6-2. The ending position is such that the z-coordinate of the end-effector has a negative value of -18 inches, but the x and y coordinates are unchanged. The least-norm solution was used. Collision with the object occurred about halfway through the trajectory.

Figure 6-5 shows the LTM on the same trajectory with the same orientation, but the manipulator now avoids the obstacle. A performance criterion which maximizes the distance of the elbow from a point on the object was used. With this performance criterion link 4 approaches the obstacle as in Fig. 6-4, but does not collide with it. The plot of the hand's position shows that the x and y coordinates are unchanging, while the z coordinate changes from a zero value at the anthropomorphic stance to a negative value of -18 inches. Thus, the manipulator was able to reach the desired position and orientation while executing a straight line trajectory with a fixed orientation, during which link 4 avoided collision with an object in the manipulator's environment.

6.4. Avoiding a Joint Limit

Figure 6-6 shows the LTM on a straight line trajectory with a fixed orientation. Motion starts from the anthropomorphic stance and ends at a position to the right and above the shoulder of the LTM. To show that a joint limit problem exists utilization of redundancy was not performed and the least-norm solution was used. Notice from the plot of joint angles that joint 5 has reached -30° , which is its lower limit. The arm is stopped when a joint limit is reached.

Figure 6-7 shows the end-effector on the same trajectory, however, due to the improvement of performance criterion $H = (\theta_3 - 52.5^\circ)^2 + (\theta_5 - 52.5^\circ)^2$, which attempts to bring joint 5 into the midrange of its limits, the elbow is being held higher. Joint 3 is also being brought to its midrange in order to avoid reaching a joint limit later in the trajectory. Error checking was done and if errors were detected, the arm would have stopped before reaching the final desired position. The plot of joint angles shows joint 5 moving away from its joint limit of -30° toward the midrange of its possible motion. Thus



Path Data:

Number of Nodes: 1

Starting: $\mathbf{Q} = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (2 - 23 \cdot C_1 C_2)^2 + (23 \cdot S_1 C_1)^2$$

$$H = H + (10 + 23 \cdot S_2)^2$$

Node #1:

Move Time : 5 seconds

$\Delta T = .01$ seconds

Acceleration time: .5 seconds

Position of Hand: $\mathbf{p} = [32 \ -23 \ -18]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

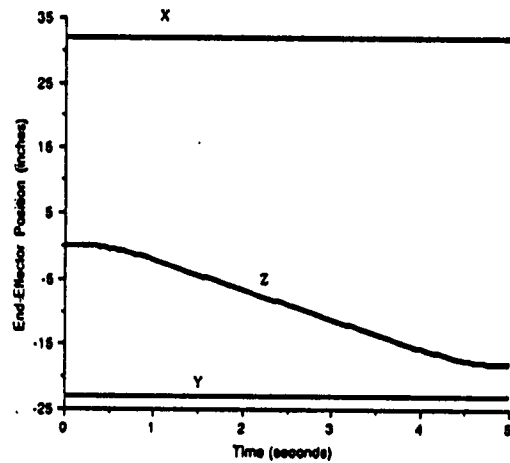
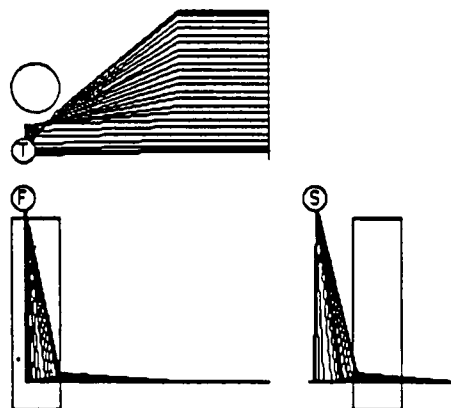
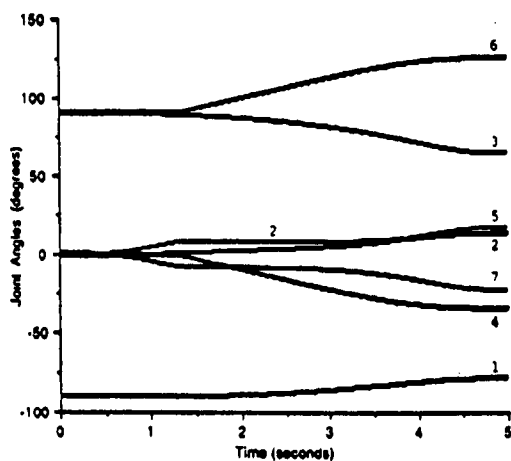


Figure 6-4: End-effector motion along a straight line with a fixed orientation.
(T = top view F = front view S = side view)
(Least-norm solution -- arm hits object)



Path Data:

Number of Nodes: 1

Starting: $\theta = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (2 - 23 \cdot C_1 C_2)^2 + (23 \cdot S_1 C_1)^2$$

$$H = H + (10 + 23 \cdot S_2)^2$$

Node #1:

Move Time : 5 seconds

$\Delta T = .01$ seconds

Acceleration time: .5 seconds

Position of Hand: $p = [32 \ -23 \ -18]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

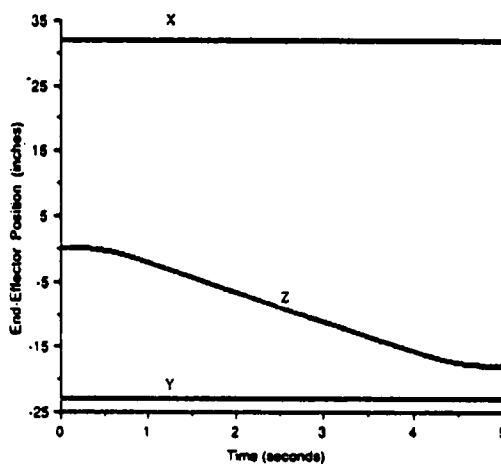
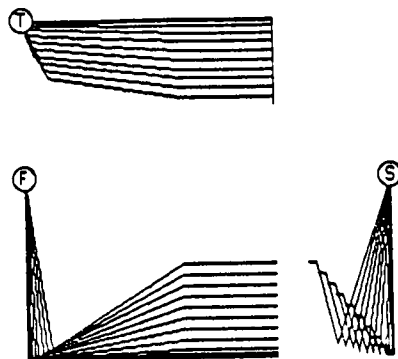
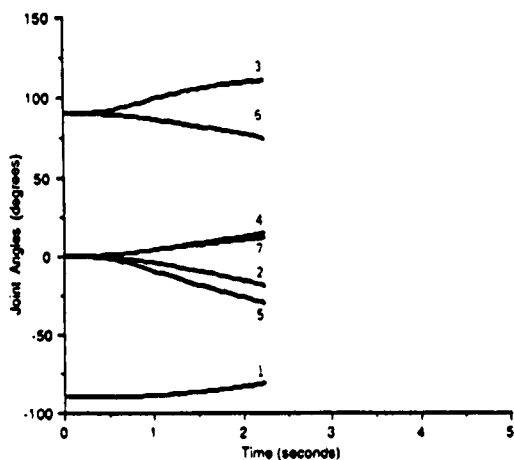


Figure 6-5: End-effector motion along a straight line with a fixed orientation.
(T = top view F = front view S = side view)
(Redundancy utilized to avoid object)



Path Data:

Number of Nodes: 1

Starting: $\underline{\theta} = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (\theta_3 - 52^\circ)^2 + (\theta_5 - 52^\circ)^2$$

Node #1:

Move Time : 5 seconds

$\Delta T = .05$ seconds

Acceleration time: .5 seconds

Position of Hand: $\underline{p} = [32 \ 5 \ 23]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

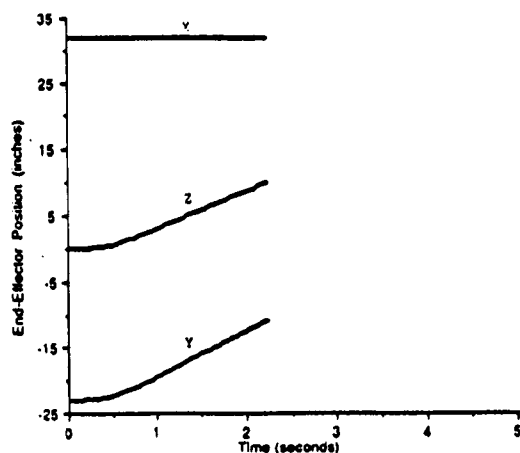
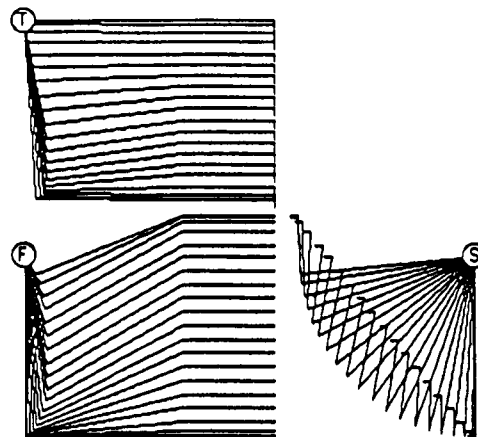
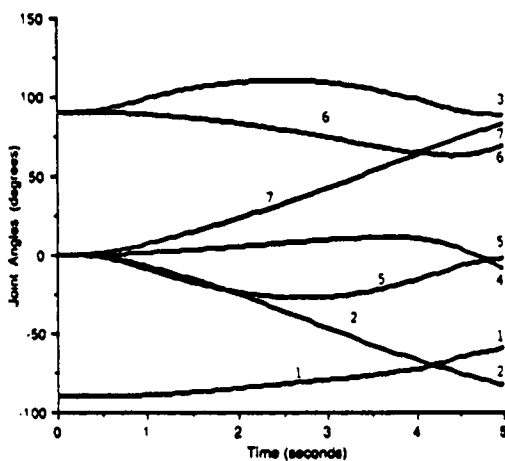


Figure 6-6: End-effector motion along a straight line with a fixed orientation.
(T = top view F = front view S = side view)
(Least-norm solution -- reaches joint 5 limit)



Path Data:

Number of Nodes: 1

Starting: $\underline{Q} = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (\theta_3 - 52^\circ)^2 + (\theta_5 - 52^\circ)^2$$

Node #1:

Move Time : 5 seconds

$\Delta T = .05$ seconds

Acceleration time: .5 seconds

Position of Hand: $\underline{p} = [32 \ 5 \ 23]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

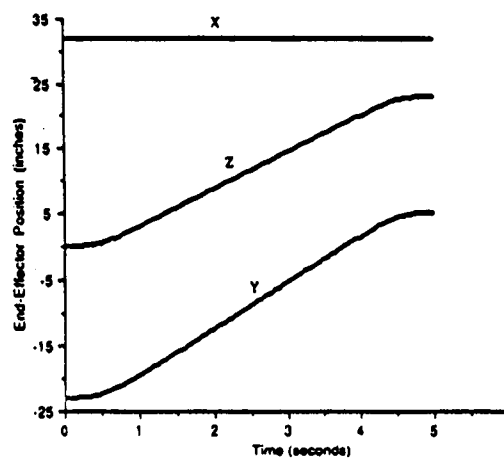


Figure 6-7: End-effector motion along a straight line with a fixed orientation.
(T = top view F = front view S = side view)
(Redundancy utilized to avoid joint limits)

the manipulator was able to reach the desired position and orientation by avoiding the lower joint limit on joint 5 through improving a performance criterion.

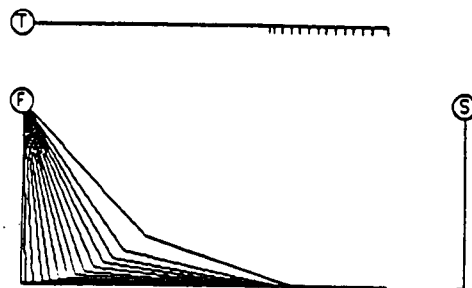
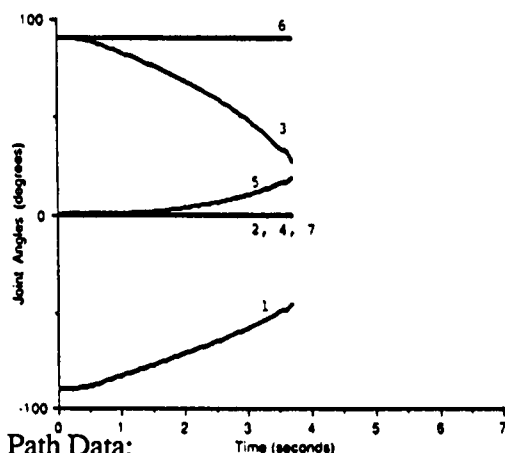
6.5. Avoiding High Joint Velocities

In Fig. 6-8 the end-effector has been directed to go from the anthropomorphic stance in the positive x direction 48.33 inches and then directly up 10 inches using only the least-norm solution but with error checking. This particular trajectory brings the arm close to a singularity at $\theta_3 = 0^\circ$ and $\theta_4 = 0^\circ$. The plot of joint angles shows that joint 3 is dropping rapidly to 0° , but even before the singularity is reached, one or more joint velocities become too great for the motors to track the trajectory and the arm is stopped.

Figure 6-9 shows the arm on the same trajectory, but performance criterion $H = (\theta_3 - 52.5^\circ)^2 + (\theta_5 - 52.5^\circ)^2$ is applied. Error checking during this simulation was done. Notice that joint angle 4 is being held away from 0° even though joint angle 3 approaches 0° . On this particular trajectory the arm must almost pass through a singular configuration and joint angle 3 and 4 perforce must both be close to 0 simultaneously, however, improving the performance criterion has kept joint velocities within joint velocity limits. When the hand starts to move up from $y = -23$ inches to $y = -18$ inches, the joint angle increments become larger. These large increments are due to the close proximity of the singularity. However, velocity error checking indicates that all joints velocities stayed within joint velocity bounds.

6.6. Smooth Changes in Trajectory

Smooth changes in trajectory require continuity in position, velocity, and acceleration. A trajectory which starts with the end-effector in a position above and to the right of the shoulder, brings it down to the position and orientation it has in the anthropomorphic stance, and then back up to the left side of the shoulder was given to demonstrate smooth changes in trajectory. The plot of the y coordinate of the end-effector in Figure 6-10 shows that at the beginning of the trajectory the end-effector smoothly accelerated to a constant velocity, during the transition between two path segments it smoothly accelerated from segment to segment, and at the end it smoothly decelerated to a stop. Thus continuity in position, velocity, and acceleration have been demonstrated.



Path Data:

Number of Nodes: 2

Starting: $\theta = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (\theta_3 - 52^\circ)^2 + (\theta_5 - 52^\circ)^2$$

Node #1:

Move Time : 4 seconds

$\Delta T = .05$ seconds

Acceleration time: .4 seconds

Position of Hand: $p = [48.33 \ -23 \ 0]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Node #2:

Move Time : 3 seconds

$\Delta T = .05$ seconds

Acceleration time: .3 seconds

Position of Hand: $p = [48.33 \ -13 \ 0]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

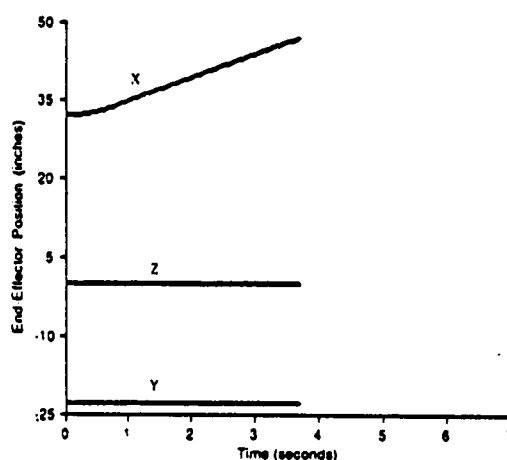
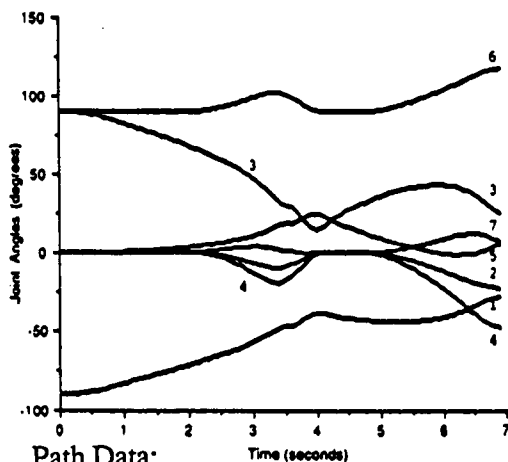


Figure 6-8: End-effector motion along Cartesian path with a fixed orientation.
(T = top view F = front view S = side view)
(Least-norm solution -- joint velocity too high)



Path Data:

Number of Nodes: 2

Starting: $\theta = [-90 \ 0 \ 90 \ 0 \ 0 \ 90 \ 0]^T$

Criterion:

$$H = (\theta_3 - 52^\circ)^2 + (\theta_5 - 52^\circ)^2$$

Node #1:

Move Time : 4 seconds

$\Delta T = .05$ seconds

Acceleration time: .4 seconds

Position of Hand: $p = [48.33 \ -23 \ 0]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Node #2:

Move Time : 3 seconds

$\Delta T = .05$ seconds

Acceleration time: .3 seconds

Position of Hand: $p = [48.33 \ -13 \ 0]^T$

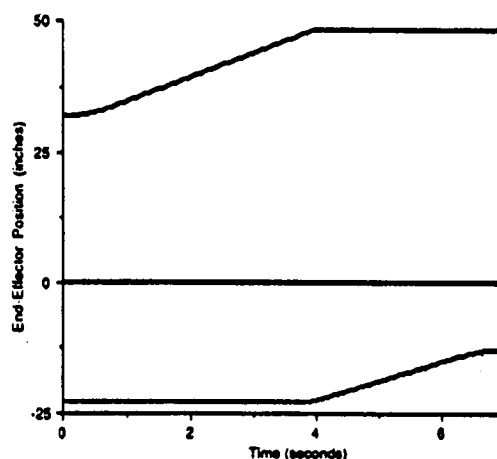
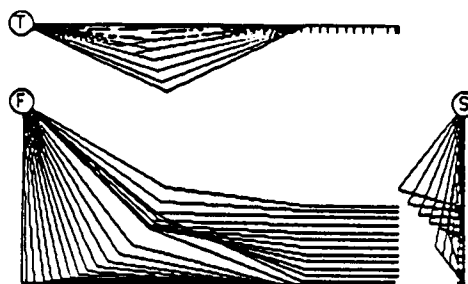
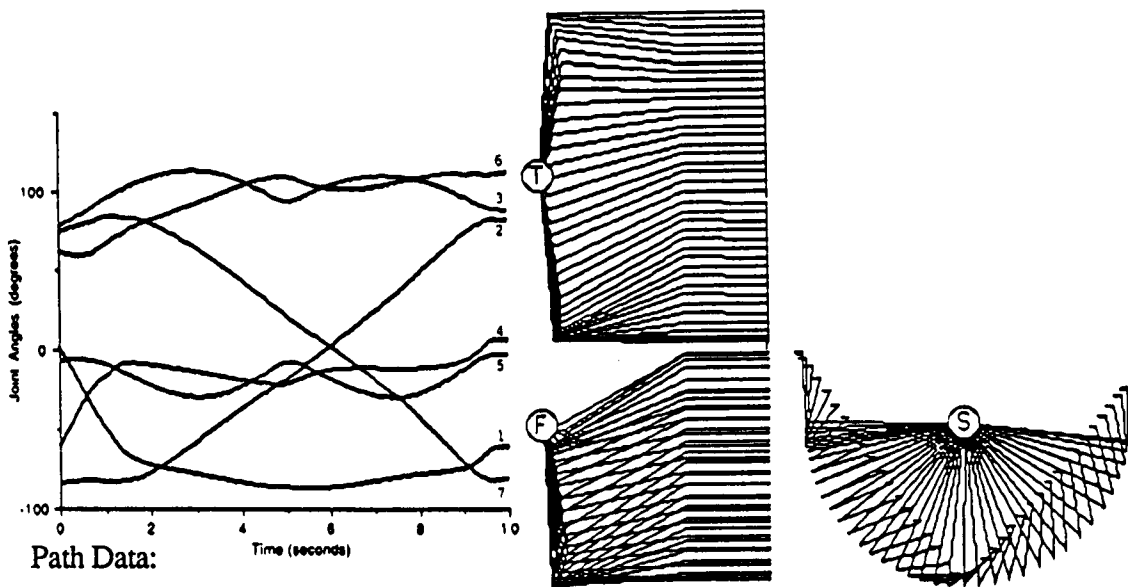


Figure 6-9: End-effector motion along Cartesian path with a fixed orientation.
(T = top view F = front view S = side view)
(Redundancy utilized to avoid singularity)



Path Data:

Number of Nodes: 2

Starting: $\theta = [4 \ -84 \ 76 \ -64 \ -8 \ 62 \ 72]^T$

Criterion:

$$H = (\theta_1 + 90^\circ)^2 + (\theta_5 - 52^\circ)^2$$

Node #1:

Move Time : 5seconds

$\Delta T = .001$ seconds

Acceleration time: .5 seconds

Position of Hand: $p = [32 \ -23 \ 0]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Node #2:

Move Time : 5 seconds

$\Delta T = .001$ seconds

Acceleration time: .5 seconds

Position of Hand: $p = [32 \ 5 \ -23]^T$

Orientation: ${}^0R_7 = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

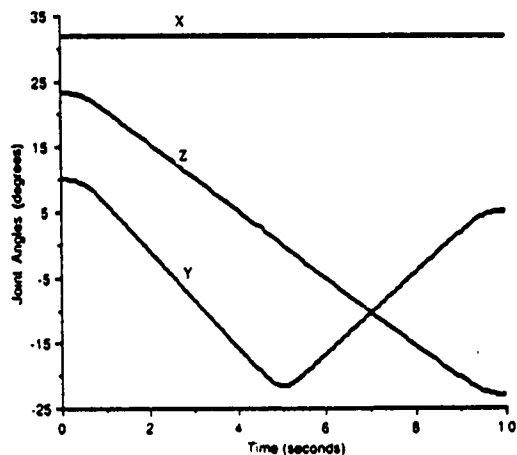


Figure 6-10: End-effector motion with smooth changes in trajectory.
(T = top view F = front view S = side view)

CHAPTER 7

USER INSTRUCTIONS

7.1. Introduction

The software for robotic control of the LTM in both Cartesian and joint space was developed using a simple interactive type of interface. An event oriented interface to be used with the Macintosh Apple computer was then developed for the user. A listing of this Macintosh interface can be found in Appendix F, but it is beyond the scope of this thesis to explain the details of the code. Familiarity with a minimum of nomenclature associated with the Macintosh is necessary before details pertaining to the interface can be discussed. These terms are enumerated directly below:

- (1) **Button (of a dialog):** A button of a dialog is an area of the dialog which, when clicked in, will cause something to happen.
- (2) **Click (on or in something):** Push the button of the mouse down and then immediately release it. The cursor arrow is in some desired place when clicking is done, such as a button of a dialog as defined above. Two quick clicks is a "double " click.
- (3) **Dialog (of the Macintosh screen):** A dialog is a window which appears on the Macintosh screen which is used to communicate with the user in some way.
- (4) **Drag (the mouse):** To move the mouse while holding down on the button of the mouse.
- (5) **Menu:** A list of items to choose from.
- (6) **Pull Down (a menu):** Move the cursor arrow to a main menu item on the menu bar across the top of the screen and push the mouse button. A menu will come down. Drag the cursor to your choice on the menu.
- (7) **Push (a button):** To click on a dialog button.

The description of the user interface is broken into five parts: (1) the robotics menu, (2) the Cartesian space dialog, (3) the joint space dialog, (4) the help menu, and (5) supplementary information. For clarity all menu and dialog items in the body of the text are in bold style in order to differentiate them from the rest of the text.

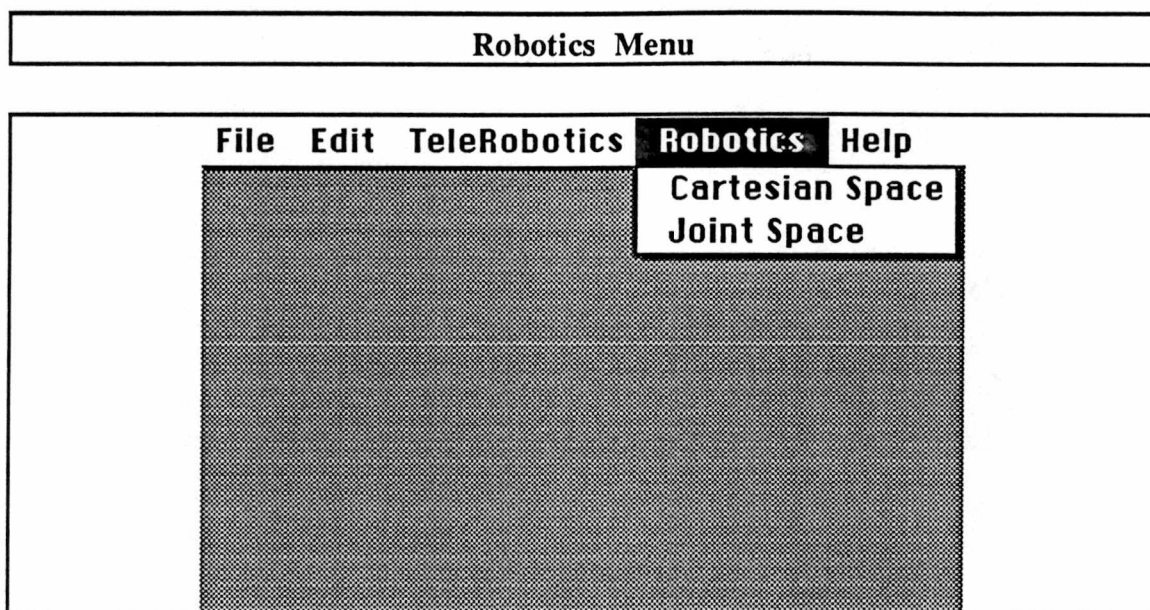


Figure 7-1 : The Robotics menu

7.2. Robotics Menu

Figure 7-1 shows the main menu bar from which the operator can choose which method to control the manipulator. He does so by positioning the cursor arrow over the menu item **Robotics** and pushing down on the mouse button, which pulls down the **Robotics** menu. The operator may drag the cursor arrow to either **Cartesian Space** or **Joint Space**.

If **Cartesian Space** is chosen the Cartesian space dialog appears on the screen. This dialog allows the user to easily specify parameters necessary to move the end-effector of the manipulator on a specified trajectory in three-dimensional Cartesian space.

If **Joint Space** is chosen the joint space dialog appears on the screen. This dialog allows the user to easily specify parameters necessary to move the manipulator arm in joint space.

Both Cartesian space and joint space dialogs may be on the screen at the same time, however, they are either both inactive or one of them is active. An inactive dialog is made active by clicking anywhere on it.

Cartesian Space Dialog

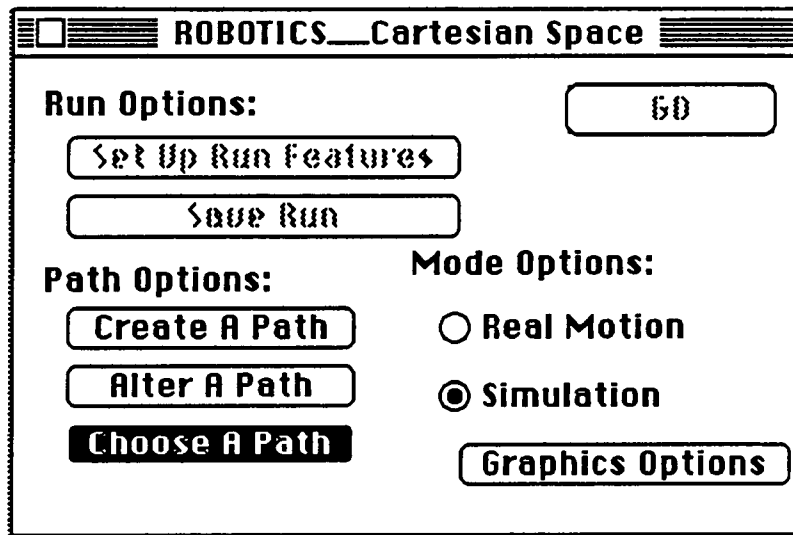


Figure 7-2 : Cartesian Space dialog

7.3. Cartesian Space Dialog

When the mouse button is released after being dragged to **Cartesian Space**, the dialog in Figure 7-2 will appear on the screen. Clicking the mouse in any of the buttons on the dialog except the **GO** button will cause another dialog to appear which must be answered.

Note the three ways a button can appear: (1) dimmed letters, which indicates the button is inactive and nothing will happen when it is clicked; (2) non-hilited (black letters with white background), which indicates the button is active; and (3) hilited (white letters with black background), which indicates that the button is active and *must be pushed and its dialog answered* before other inactive buttons will become active.

Note that the **Choose A Path** button is hilited. This is to indicate to the operator that before the **Set Up Run Features** button will become active, a path must be chosen. Additionally, the **GO** button will not become active until the **Set Up Run Features** dialog has been answered.

Each button on the **Cartesian Space Dialog** will now be discussed in alphabetical order.

Alter A Path / Choose Path

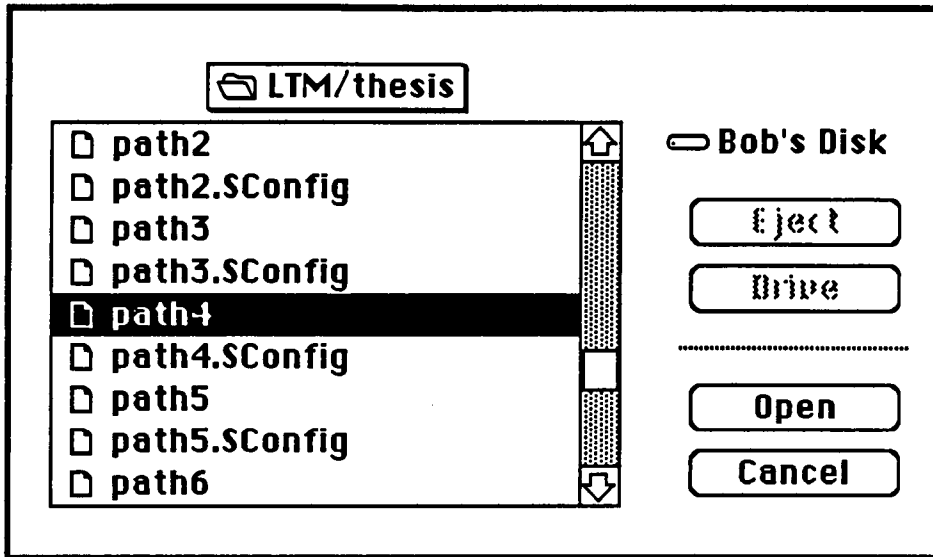


Figure 7-3 : Alter A Path dialog

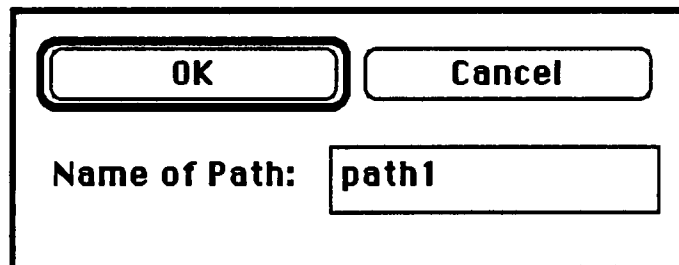


Figure 7-4 : Choose Path dialog

7.3.1. Alter a Path

When **Alter A Path** is clicked the dialog of Fig. 7-3 appears. Double clicking on any of the files will open that file. Clicking once on a file and then pushing the **Open** button will also open the file. After editing the path file, click the little box in the upper left hand corner of the file. This automatically saves and then closes the file. This option can be used to open and observe other files too.

7.3.2. Choose Path

When **Choose Path** is clicked the dialog shown in Fig. 7-4 appears. Enter the path name (created by using **Create A Path**) in the box and click **OK**.

Create A Path

Number of Nodes:

☐ **Always same Move Time**
the Move Time:

☐ **Always same Acc Factor**
the Acc Factor:

OK

Cancel

PATH NAME:

ORIENTATION MATRIX

0.0000	0.0000	1.0000
0.0000	-1.0000	0.0000
1.0000	0.0000	0.0000

POSITION

32.0000
-23.0000
0.0000

Move Time:

Acc Factor:

Node:

Node OK

Node Cancel

Figure 7-5 : The Create A Path dialog (Cartesian Space)

7.3.3. Create a Path

When the **Create A Path** button is pushed the dialog shown in Fig. 7-5 appears. The following explains each part of the **Create A Path** dialog:

Number of Nodes: Enter the number of nodes in the path you are specifying. Do not include the starting position as a node because the joint angles of the initial configuration are automatically converted into the zeroth node. A file containing the initial joint angles is automatically created and associated with a particular path whenever that path is used. For simulations the initial joint angles can be changed in the **Set Up Run Features** dialog; however, to demonstrate any particular path using the manipulator the arm must be moved in joint space in order to put the arm into the initial configuration for the path.

Always same Move Time: When this is checked the move time between nodes will be as specified in the **Move Time** in seconds. If the box is not checked the **Move Time** is ignored.

Always same Acc Factor: When checked the acceleration factor between nodes will be as specified in the **Acc Factor**. A definition for acceleration factor is given directly below.

DEFINITION: The acceleration factor is a decimal fraction of the total time between two nodes. It is used to calculate the duration of acceleration. See `GetDesiredPosition()` for more detail.

Orientation Matrix: This matrix specifies the orientation of the end-effector. The first column represents the x-axis of the end-effector's coordinate system referred to the base coordinate frame, the second column the y-axis of the end-effector's coordinate system referred to the base coordinate frame, and the third column the z-axis of the end-effector's coordinate system referred to the base coordinate frame. Each column is a unit vector of direction cosines.

Position: This column contains the x, y, z Cartesian coordinates of the origin of the coordinate system attached to the end-effector. These coordinates are expressed relative to the base coordinate system.

Move Time: If **Always same Move Time** is not checked, then a move time in seconds for each node must be specified here.

Acc. Factor: If **Always same Acc Factor** is not checked, an acceleration factor for each node must be specified here.

NODE: Tells you which node you are currently working on.

Node OK: Must be clicked for each node in a path. This indicates the node is complete. If **Node OK** is not pushed **NODE** will not advance. When specifying a path with only one node, **Node OK** must still be pushed.

Node Cancel: When **Node Cancel** is pushed everything in the current **Orientation Matrix**, and **Position** column is erased. The data in the **Move Time** and **Acc Factor** boxes is also erased. If a node has just been specified by clicking on **Node OK**, then **Node Cancel** *does not cancel that node*, it merely erases the data shown on the dialog that was used to specify that node. Using **Node Cancel** has the advantage of starting fresh for each specified node, however, the data for a series of nodes may not differ by a lot, in which case it would be easier to not use **Node Cancel**.

Path Name: The path must be named or you cannot leave the dialog except through pushing the **Cancel** button. If **OK** is pushed without a name being specified, a dialog appears which requests that the path be named.

Graphics Options

Specify A View:

☐ Top
☐ Front

☒ All Views
☐ Side

OK

Cancel

Show:

☐ Shoulder
☐ Identification of View
☐ Definition of Top, Front, Side

Adjust Image:

Size:

Location:	Top	Front	Side
Vertical.....	50	150	150
Horizontal.....	50	50	350

Figure 7-6 : Graphics Options dialog

7.3.4. Graphics Options

This dialog gives you some control over the simulation images.

Specify a View: Choose a view by clicking on one of the circles.

Shoulder: If the box next to **Shoulder** is checked, a circle will be drawn to indicate where the shoulder of the manipulator is.

Identification of View: If the box next to **Identification of View** is checked, a T will appear in a circle drawn for the shoulder of the top view. Likewise, an F will appear for the front view and an S will appear for the side view.

Definition of Top, Front, Side: If the box next to **Definition of Top, Front, Side** is checked an image of the manipulator in the anthropomorphic stance will appear as the last thing to be drawn. This image defines the top, front, and side views.

Figure 7-7 illustrates a simulation with only the **Shoulder** option checked. Figure 7-8 illustrates a simulation with all three **Show** options checked.

Size: Scales the image

Location: Locates the origin of the manipulator's shoulder on the Macintosh screen for the **Top**, **Front**, and **Side** views.

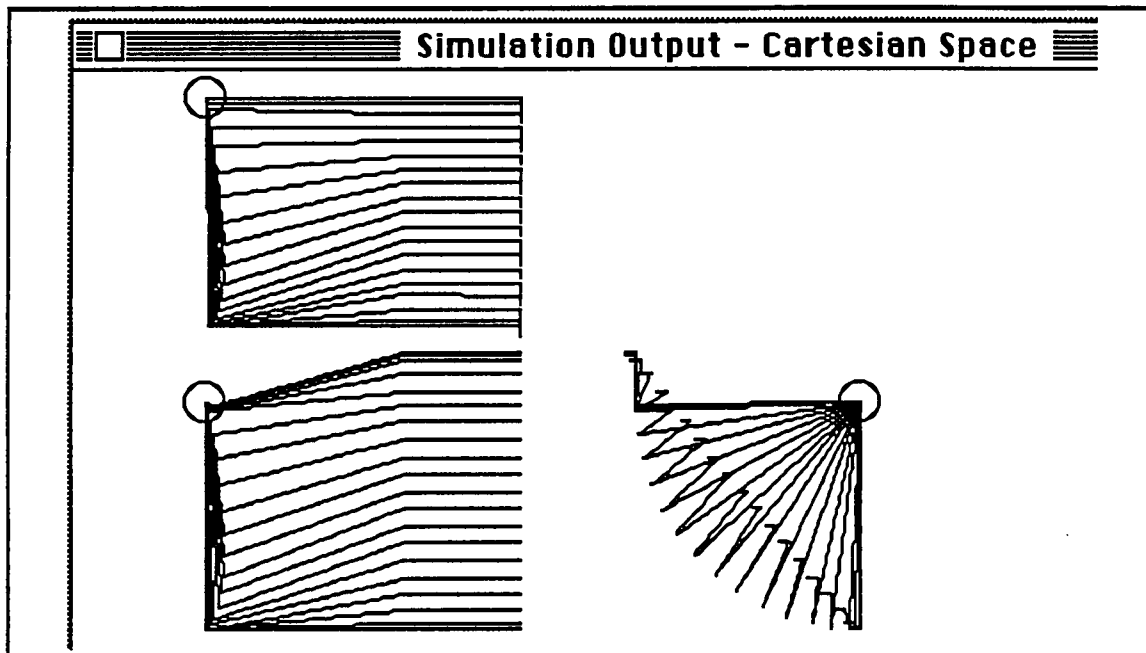


Figure 7-7 : Show Shoulder option

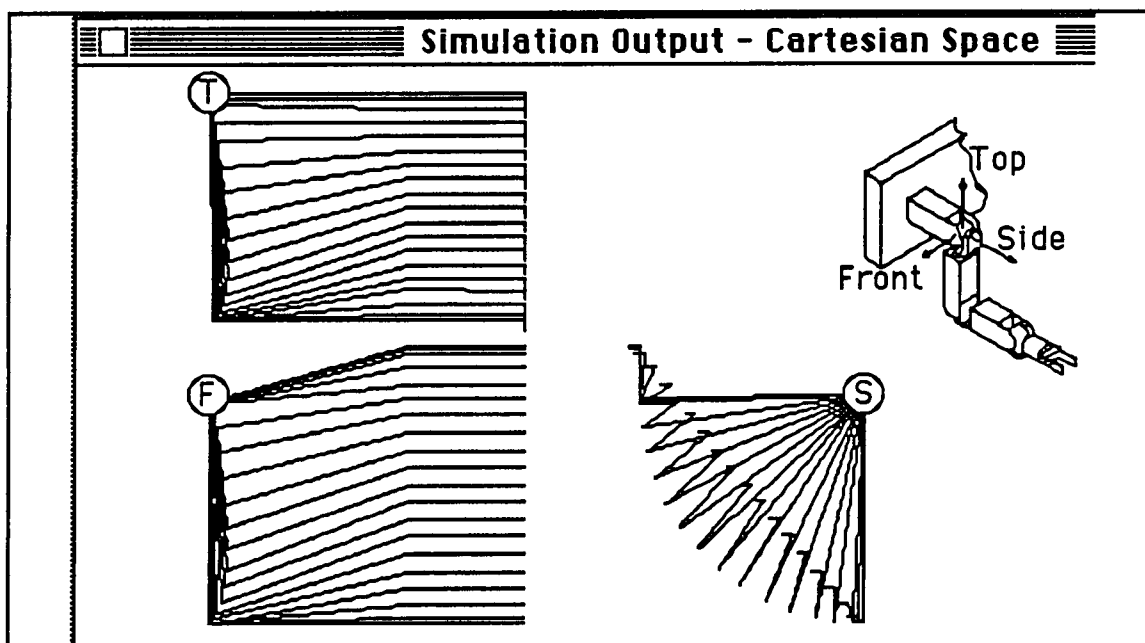


Figure 7-8 : All three Show options

Real Motion / Simulation / Save Run
Mode Options: ☐ Real Motion ☒ Simulation

Figure 7-9 : Real motion /simulation control

OK Cancel Name of Path: path1

Figure 7-10 : Save Run dialog

7.3.5. Real Motion / Simulation

The type of control shown in Fig. 7-9 allows only one option to be selected at a time. Thus when **Real Motion** is clicked, **Real Motion** becomes active and **Simulation** becomes inactive and vice versa. A circle with a black dot in it indicates the active condition.

When **Real Motion** is clicked a flag is set which takes the manipulator out of the simulation mode and into the real mode. It also informs higher level software to use the current information in the global data buffer regarding trajectory. Once the **GO** button is pushed the arm will actually move. When **Simulation** becomes active a flag is set which puts the manipulator into simulation mode.

7.3.6. Save Run

When **Save Run** is clicked the dialog shown in Fig. 7-10 appears on the screen. A name for the run just completed must be entered. The files described in the dialog **Set Up Run Features** will be appended to the name entered for the path, after which they will be saved.

Set Up Run Features (Cartesian Space Dialog)

PREFERENCES		STARTING CONFIGURATION	
Parameter		OK	Cancel
Perf.Criterion:	1	☒ Home Position ☐ Specify Degrees	
delta_t:	0.05 sec		
Parameter 3:	None		
Parameter 4:	None		
Type of output:		Theta 1	-90
☒ Graphics		Theta 2	0
Output every	5 images	Theta 3	90
☒ Numbers		Theta 4	0
Type of solution:		Theta 5	0
☐ Least Norm		Theta 6	90
☒ Optimized		Theta 7	0

Figure 7-11 : Set Up Run Features dialog (Cartesian Space)

7.3.7. Set Up Run Features

When Set Up Run Features is clicked the dialog shown in Fig. 7-11 will appear. Each section of the dialog will now be considered:

Parameter:

Perf. Criterion: The function GetGradientH(), which is located on the module GetGradientH.c, calculates the gradient of the optimization criterion. A switch statement in this function associates a case number with the gradient of a particular performance criterion. The user will have to locate that switch statement in order to use his own optimization criteria in the program. **Perf. Criterion** stands for performance criterion. Enter the case number for the desired performance criterion.

delta_t: This is the Δt in $\Delta\theta/\Delta t$, $\Delta x/\Delta t$, $\Delta y/\Delta t$, etc.

Parameter 3: Reserved for later use.

Parameter 4: Reserved for later use.

Type of Output :

Graphics: When **Graphics** is checked there will be a stick figure representation of the manipulator during simulations. The simulations are done in real-time.

Output every [] images: If a stick figure image of the manipulator was drawn for each iteration of the positioning routine, different configurations of the manipulator could not be distinguished from each other. Enter how many to skip before an image is drawn.

Numbers: If **Numbers** is checked these files (in italics) will be created during each run:

theAngles: All the joint angles for each iteration.

Example:

```
1 -90.000004 1.432395 90.000004 1.647254 0.000000 88.352750 -1.432395 1.233701
0 -90.020548 0.000533 90.002020 0.001431 0.018529 89.998059 0.000000 1.233248
0 -90.000046 -1.432923 90.015664 -1.632163 -0.015605 91.632680 1.432395 1.234558
0 -90.021538 0.000392 90.051434 0.053928 -0.029888 89.946612 0.000000 1.235931
0 -90.001229 -1.431681 90.118488 -1.526546 -0.117271 91.527088 1.432395 1.240173
1 -90.022776 0.001781 90.224639 0.231035 -0.201833 89.769502 0.000000 1.245420
```

Each line represents one configuration of the manipulator. A 1 in the first position of a line indicates that an image has been drawn on the Macintosh screen, a 0 indicates that no image was drawn; therefore, every fifth manipulator configuration is being drawn on the screen. The next seven numbers are the joint angles in degrees, starting with joint 1 and ending with joint seven, and the last number is the optimization criterion.

theCricket: This file contains information which can be put into the scrapbook (or put on the Clipboard) and transferred to the canned graphics program CricketGraph.

Example:

```
-90.000004 1.432395 90.000004 1.647254 0.000000 88.352750 -1.432395 31.991734 -23.007184
-0.000198 1.233701 0.000000 ⇐ last number is time plus delta_t.

-90.020548 0.000533 90.002020 0.001431 0.018529 89.998059 0.000000 31.991751 -23.006466
0.000179 1.233701 0.050000 ⇐ last number is time plus delta_t.

-90.000046 -1.432923 90.015664 -1.632163 -0.015605 91.632680 1.432395 31.991867 -23.001598
0.005646 1.233248 0.100000 ⇐ last number is time plus delta_t.
```

The first seven decimal numbers are the joint angles starting from joint 1 and ending with joint 7. The next three decimal numbers are the x,y,z world coordinates of the origin of the coordinate system affixed to the end-effector. The next decimal number is the value of the

optimization criterion. The last number is the time, incremented by **delta_t** as specified above. This data is tabbed especially to be accepted by CricketGraph.

thePath: This file contains the path that was specified for the end-effector.

Example:

1	Number_of_Nodes	⇐ number of nodes in full path
1	Node_number	⇐ node the data applies to
0.000000	x1	⇐ start of orientation matrix
0.000000	x2	
1.000000	x3	
0.000000	y1	
-1.000000	y2	
0.000000	y3	
1.000000	z1	
0.000000	z2	
0.000000	z3	⇐ end of orientation matrix
32.000000	p1	⇐ start of position vector
0.000000	p2	
23.000000	p3	⇐ end of position vector
5.000000	move_time	⇐ time to move to this node
0.100000	acceleration_factor	⇐ decimal fraction of move_time
1	LastNodeFlag	⇐ will be 1 if last node

Number_of_Nodes gives the number of nodes in the path. This path has 1 node (besides the starting position). Node_number gives the specific node number in a path the data applies to. The next nine numbers designated x1,x2,x3,y1,y2,y3,z1,z2, and z3 comprise the specified orientation matrix. The next three numbers, designated p1, p2, and p3 give the x,y,z location of the origin of the coordinate system of the end-effector. The move_time gives the time to move to that node. Acceleration_factor is defined in the discussion for the dialog **Create A Path**. The LastNodeFlag informs the program if the node it is on is the last node. If LastNodeFlag were 0 then a new node would be loaded at the appropriate time.

thePos: This file contains the x,y,z world coordinates of the elbow, the wrist, and the end-effector.

Example:

1	⇐ indicates image was drawn during simulation
-0.000001 -22.992813 -0.574940	⇐ elbow position
19.991734 -23.007184 -0.000199	⇐ wrist position

31.991734 -23.007184 -0.000198 $\Leftarrow$ end-effector position
31.991734 -23.007184 0.999802 $\Leftarrow$ orientation vector

0 $\Leftarrow$ no image was drawn during simulation
-0.008248 -22.999999 -0.000214
19.991751 -23.006466 0.000286
31.991751 -23.006466 0.000179
31.991760 -23.006475 1.000179

The last line gives the x, y, z coordinate of a unit vector pointing out from the hand; it is used to help indicate the orientation of the end-effector. Whether or not an image was drawn for a particular set of data is indicated by a 1 or 0 on the line immediately preceding any particular set of data. A 1 indicates an image was drawn during the iteration, while a 0 indicates that no image was drawn for that iteration.

ThePerfCrit: This file contains the performance criterion as distinguished by its place in the switch statement of the function GetGradientH() where the gradient of the performance criterion is calculated.

Example: 2

Type of Solution:

Click in one of the circles to choose either **Least Norm** or **Optimized**.

Least Norm: Choosing **Least Norm** causes only the least Euclidean norm solution to be calculated.

Optimized: Choosing **Optimized** allows for optimization of the manipulator arm.

Starting Configuration:

Click in one of the circles to choose **Home Position** or **Specify Degrees**.

Home Position: Choosing **Home Position** will start the arm from the anthropomorphic stance.

Specify Degrees: Choosing **Specify Degrees** will start the arm from the configuration dictated by the joint angles specified in **Theta1**, **Theta2**, etc. In order to run simulations in joint space the initial angles of the joint path must be specified with this option.

Joint Space Dialog

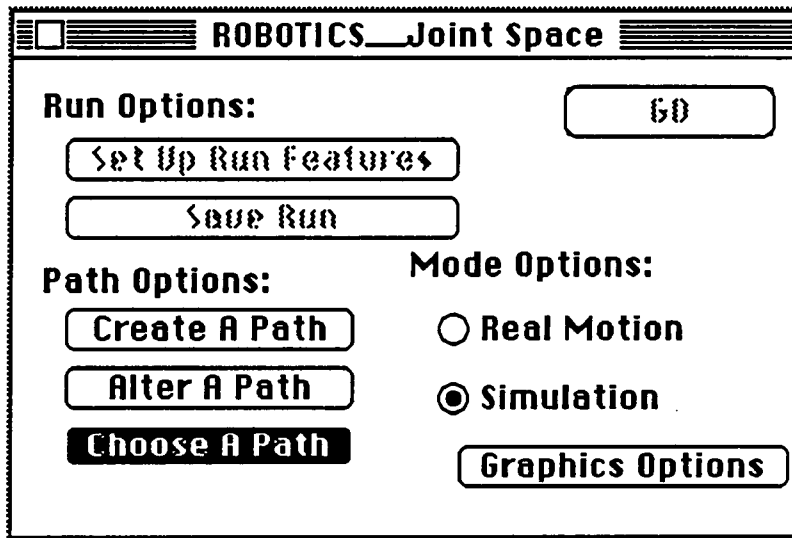


Figure 7-12 : Joint Space dialog

7.4. Joint Space Dialog

When the mouse button is released after being dragged to **Joint Space**, the dialog in Figure 7-12 will appear on the screen. Clicking the mouse in any of the buttons on the dialog except the **GO** button will cause another dialog to appear which must be answered.

Note the three ways a button can appear: (1) dimmed letters, which indicates the button is inactive and nothing will happen when it is clicked; (2) non-highlighted (black letters with white background), which indicates the button is active; and (3) highlighted (white letters with black background), which indicates that the button is active and *must be pushed and its dialog answered* before other inactive buttons will become active.

Many of the buttons on the **Cartesian Space** and **Joint Space** dialogs cause the same dialog to appear. Such buttons are discussed in Section 7.3. only. Each button on the **Joint Space** dialog will now be discussed in alphabetical order.

7.4.1. Alter a Path

See Section 7.3.1. for an explanation of **Alter A Path**.

7.4.2. Choose a Path

See Section 7.3.12. for an explanation of **Choose A Path**.

Create A Path (Joint Space)

Name Joint Path:		OK
Number of Nodes:	1	Cancel
Current Node:	1	
Theta 1 :		Theta 4 :
Theta 2 :		Theta 5 :
Theta 3 :		Theta 6 :
		Theta 7 :
		Acc. Factor: .1
		Node OK
NOTE 1: Express joint angles in degrees.		
NOTE 2: For reference stance see Help menu.		

Figure 7-13 : Create A Path dialog for joint space

7.4.3. Create a Path

When the joint space dialog is active and the **Create a Path** button is pushed the dialog shown in Fig. 7-13 appears. Enter the name of the joint space path in **Name Joint Path**. Enter the number of nodes in the joint space path in **Number of Nodes**. Do not include the initial configuration as a node. The particular node you are working on will be indicated by **Current Node**. Enter the joint angles in degrees for each node in **Theta 1**, **Theta 2**, et cetera. Enter the fraction of the total time between nodes in **Acc Factor**. The button **Node OK** must be pushed to indicate the completion of a node.

7.4.4. Graphics Options

See Section 7.3.4. for an explanation of **Graphics Options**.

7.4.5. Real Motion / Simulation

See Section 7.3.5. for an explanation of **Real Motion / Simulation**.

7.4.3. Save Run

See Section 7.3.6. for an explanation of **Save Run**.

Set Up Run Features (Joint Space Dialog)

% of Max Speed:	90	OK Cancel
Specify delta_t:	0.01	
Type of output:		
☒ Graphics		
Output every 20 images		
☒ Numbers		
Starting Configuration:		
The current joint angles are used.		

Figure 7-14 : Set Up Run Features dialog (Joint Space)

7.4.7. Set Up Run Features

When Set Up Run Features is chosen the dialog shown in Fig. 7-14 appears. The following is an explanation of its parts:

% Max Speed: Choose the speed of movement in joint space by specifying the percent of maximum speed the joint angles can move at. If 100 is chosen the program will automatically scale it down to 90%. What you are actually doing is choosing the percent of maximum of the slowest motor.

Specify delta_t: The Δt of $\Delta x/\Delta t$, $\Delta y/\Delta t$, etc.

Type of Output: As discussed for cartesian space.

Starting Configuration: Always takes the current joint angles.

NOTE: For simulations you must specify a starting configuration using the Set Up Run Features dialog of the Cartesian Space dialog.

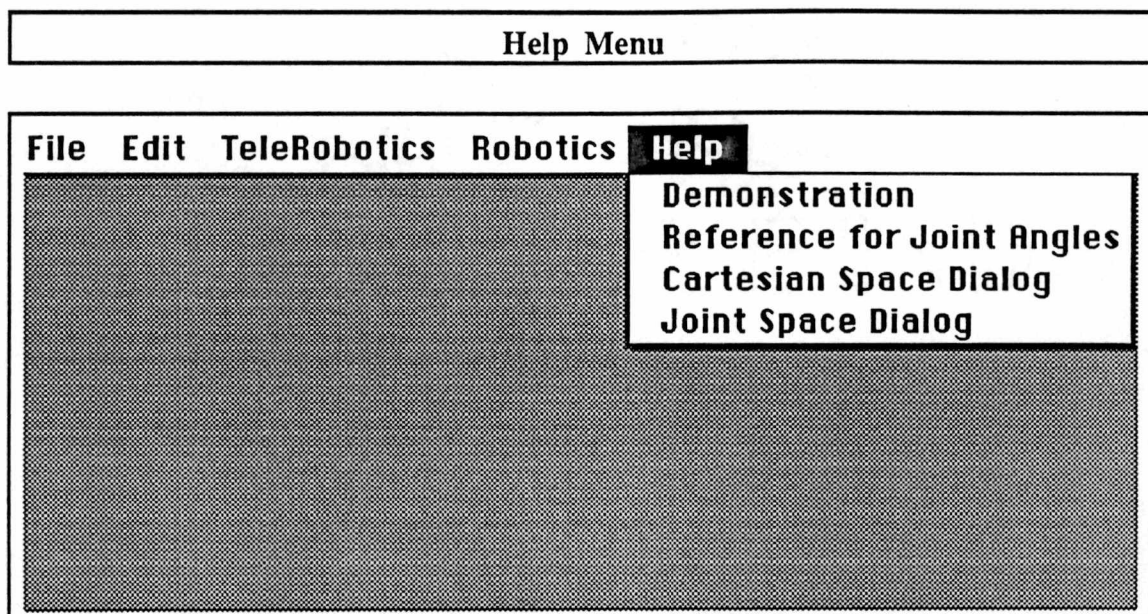


Figure 7-15 : The Help Menu

7.5. Help Menu

Figure 7-15 shows the main menu bar with the **Help** menu pulled down. The user may obtain help on the following menu items: **Demonstration**, **Reference for Joint Angles**, **Cartesian Space Dialog**, and **Joint Space Dialog**. More detail on these items is given below.

7.5.1. Help With Demonstration

There is a canned demonstration of robotic control of the LTM built into the software. When first opening the application the path **Demo1** is the default path. Choosing **Demonstration** from the **Help** menu causes a noneditable text file to appear which gives instructions on how to do the built in demonstration.

7.5.2. Reference for Joint Angles

When **Reference for Joint Angles** is selected the window shown in Fig. 7-16 appears. This window shows a rough figure of the LTM in the configuration joint angles should be referred to when specifying them. For example, joint angles for the anthropomorphic stance are $\theta_1 = -90$, $\theta_2 = 0$, $\theta_3 = 90$, $\theta_4 = 0$, $\theta_5 = 0$, $\theta_6 = 90$, and

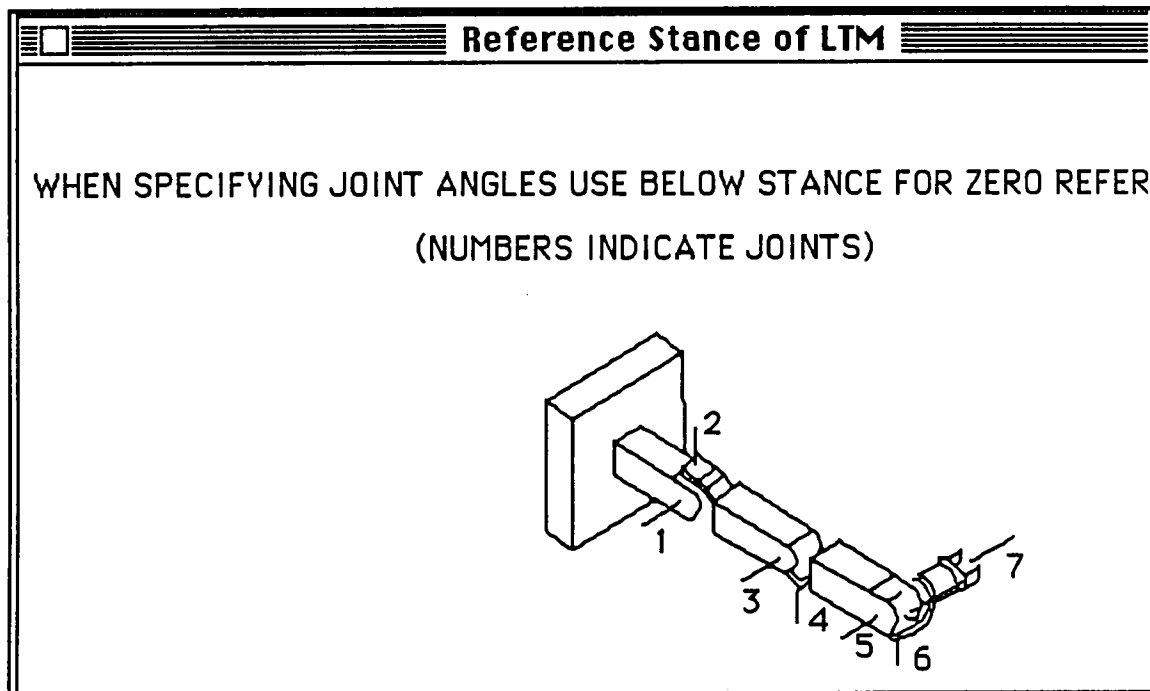


Figure 7-16 : Reference configuration of the LTM for specifying joint angles

Cartesian Space Help Dialog

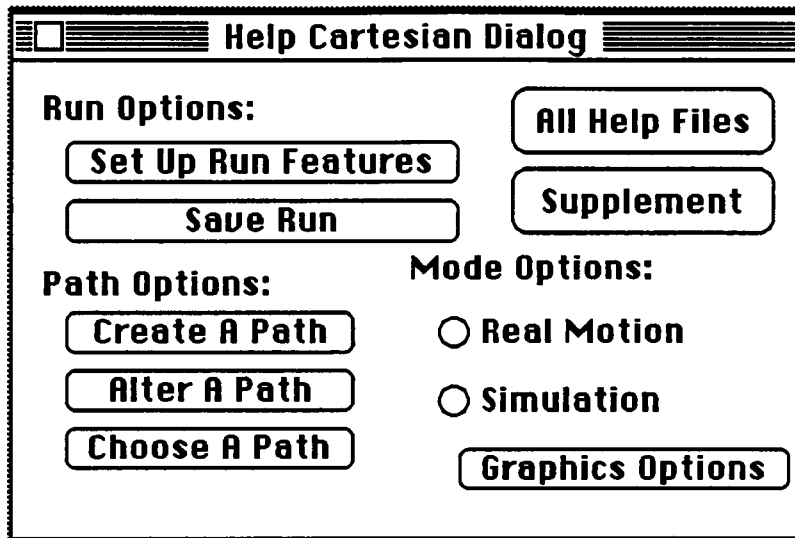


Figure 7-17 : Help dialog for Cartesian Space

$\theta_7 = 0$. The anthropomorphic stance is considered the home position of the LTM in the Cartesian space dialog Set Up Run Features.

7.5.3. Cartesian Space Help Dialog

When **Cartesian Space Dialog** is chosen from the Help menu the dialog shown in Fig. 7-17 appears. This dialog is almost identical to the dialog that comes up when **Cartesian Space** is chosen from the **Robotics** menu. To obtain help on any one of the buttons of the **Cartesian Space** dialog, click on the corresponding button of the help dialog shown in Fig. 7-17. A noneditable text file will appear with a short explanation of that button. Two buttons on the help dialog shown in Fig. 7-17 which are not on the **Cartesian Space** dialog are **All Help Files**, and **Supplement**. When **All Help Files** is pushed a window of noneditable text with all the help files will appear. When **Supplement** is pushed a noneditable text file appears which gives supplementary information, such as how to get a hard copy of simulations. This supplementary information is given in Section 7.6.

Joint Space Help Dialog

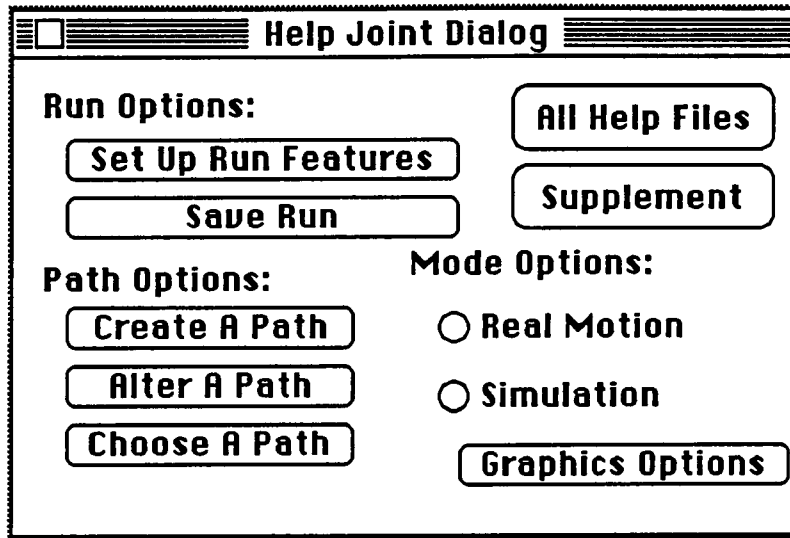


Figure 7-18 : Help dialog for joint space

7.5.4. Joint Space Help Dialog

When **Joint Space Dialog** is chosen from the Help menu the dialog shown in Fig. 7-18 appears. This dialog is almost identical to the dialog that comes up when **Joint Space** is chosen from the **Robotics** menu. To obtain help on any one of the buttons of the **Joint Space** dialog, click on the corresponding button on the help dialog shown in Fig. 7-18. A noneditable text file will appear with a short explanation of that button. Two buttons on the help dialog shown in Fig. 7-18 which are not on the **Joint Space** dialog are **All Help Files**, and **Supplement**. When **All Help Files** is pushed a window of noneditable text with all the help files will appear. When **Supplement** is pushed a noneditable text file appears which gives supplementary information, such as how to get a hard copy of simulations. This supplementary information is directly below in Section 7.6.

7.6. Supplementary Information

7.6.1. Obtaining Hard Copies

To obtain a hard copy of your simulations do these steps:

- (1) Make the simulation window the active window if it is not already so.
- (2) While holding down the Command and Shift keys of the Macintosh, push 3.
- (3) You have just made a MacPaint document of the current contents of the screen.
- (4) You may use MacPaint, FullPaint, or SuperPaint to open this document and get a hard copy. The document will appear on the desktop entitled Screen 0. If you did step 2 twice, the second MacPaint document would be called Screen 1. You may make 10 such documents before having to go to the desktop to rename them. Once you have renamed them or taken them off the desktop, you may make 10 more.

(5) The hard copy can be pasted into a text document by using an application like MacPaint and selecting the part of the MacPaint document you are interested in (remember it is of the whole screen, not just your simulation), and choosing **Copy** from the Edit menu. Then choose **ScrapBook** from the Apple menu, then **Paste** from the edit menu. You have just transferred the simulation to what is called the Scrapbook. You may now open up a word processing package, like Microsoft Word, and transfer the simulation from the Scrapbook to a document in that package.

7.6.2. Multiple Dialogs/Multiple Windows

The **Cartesian Space** dialog and the **Joint Space** dialog may be on the screen at the same time. Also, the output windows for both spaces may be on the screen at the same time. The location of the dialogs and windows on the Macintosh screen may be changed by moving the cursor arrow to the top of the dialog or window and dragging from there to the desired location. However, only one window is active at a time.

To make an inactive dialog the active dialog one can click anywhere within the inactive dialog. The inactive dialog will move to the front and become active, the formerly active dialog will move behind the newly activated dialog. Dialogs can also be made active by choosing them from the **Robotics** menu, which is handy when the dialog is open but another dialog or window completely covers it. Alternatively, instead of choosing the dialog from the menu, the window which covers it can be moved so as to uncover it. Once it is uncovered it can be made active by clicking on it.

Output windows can be made active by clicking on them also. But because the content of an output window is totally controlled by the robotics program, their content cannot be directly changed by the operator. Figure 7-19 shows an example of multiple windows.

7.6.3. Hypothetical Run

When using this program a user will be working in the LightSpeedC environment and will need to be familiar with LightSpeedC. LightSpeedC is arranged so that each program written in C is contained in a *project*. A project has all the modules, libraries, et cetera needed for the particular program it is associated with. The software was written so that the user would only have to change one or two functions on one module. That module is called **GetGradientH.c**. On it is the one or possibly two functions which must be changed from time to time, **GetGradientH()** and **GetPerfCrit()**. This was done for two reasons: (1) There is less chance other parts of the program will be unintentionally changed when adjusting performance criteria, and (2) compilation time is greatly lessened because only the small files contained on **GetGradientH.c** have to be compiled. The following steps are typical while investigating optimization criteria:

- (1) Double click on the LightSpeedC project icon (see Fig. 7-20).
- (2) Double click on the module **GetGradientH.c** to see if the performance criteria you plan to investigate are entered properly. Record which performance criteria goes with which case number in the switch statements. Make sure the gradient is correct. Enter any criteria (and their gradients) you plan on investigating.
- (3) Pull down the **Project** menu and drag the cursor to **Run**. If **GetGradientH.c** compiles, the main menu bar shown in Fig. 7-1 will appear.
- (4) Pull down the **Robotics** menu and drag the cursor to the **Cartesian Space** item.
- (5) If the path you are interested in has already been created click on **Choose Path** and enter the path name and then click on **OK**.
- (6) If the path has not been created, click on **Create A Path** and do the following:
 - (a) Enter a path name.
 - (b) Enter the number of nodes in the path, not including the starting node.
 - (c) Fill up the orientation matrix of the hand for the first node.
 - (d) Fill up the position of the hand for the first node.
 - (e) Enter a move time to the first node.

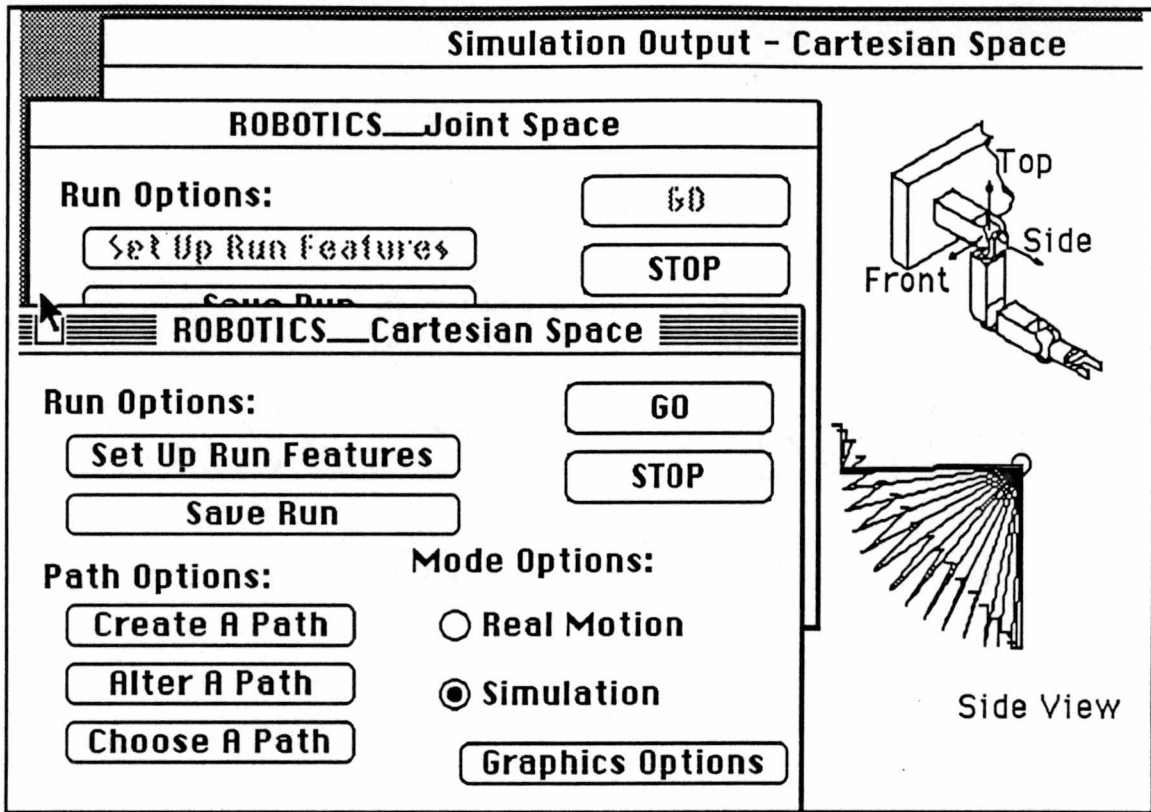


Figure 7-19 : Multiple dialogs and windows

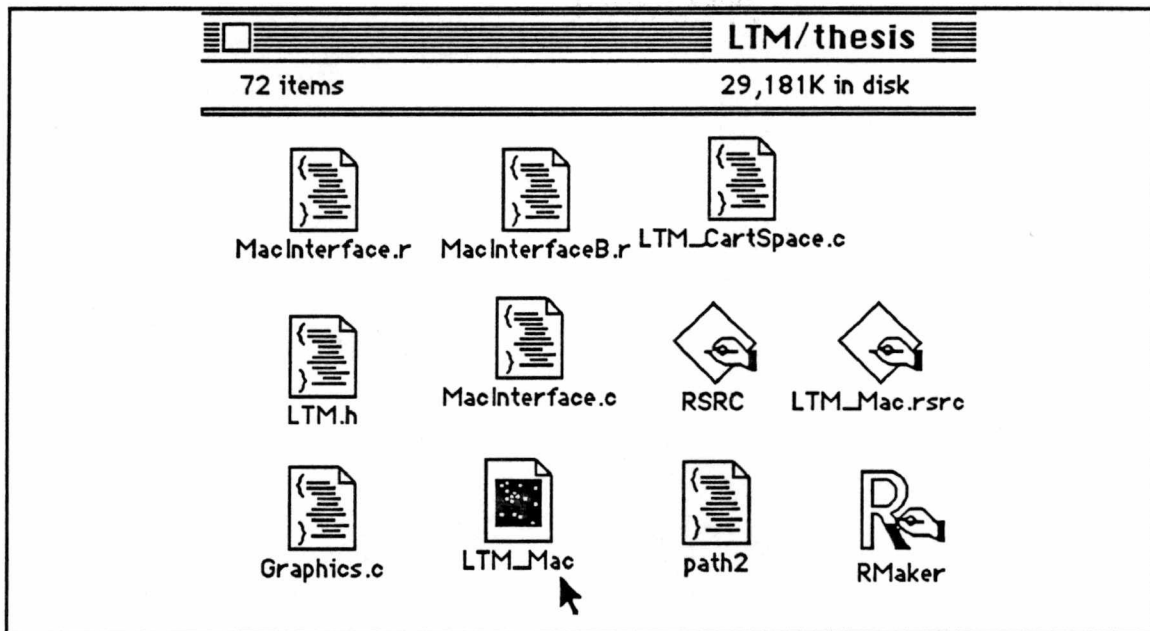


Figure 7-20 : The project icon

- (f) Enter an acceleration factor for the first node.
 - (g) Click on **Node OK**. The box next to **Node**, will change to the next highest node if there is more than one node. Otherwise it stays the same. It always indicates the node you are working on and goes no higher than the last node.
 - (h) Fill up the remaining nodes if there are any.
 - (i) One may choose **Always Same Move Time** by clicking on the square next to it and not have to enter a move time for each node.
 - (j) One may choose **Always Same Acc.Factor** by clicking on the square next to it and not have to enter an acceleration factor for each node.
 - (k) Click on **OK**.
 - (l) Click on **Choose Path** and enter the name from step (a).
- (7) Click on **Set Up Run Features**. The dialog shown in Fig. 7.3.7. will appear.
- (a) Enter the performance criterion switch number in **Perf_Criterion**.
 - (b) Enter the time increment **delta_t**.
 - (c) Pay no attention to **Parameters 1** and **Parameter 2**.
 - (d) Click on the box next to **Graphics** if you want a stick figure of the simulation.
 - (e) If **Graphics** is active, enter the number of manipulator configurations you want skipped between images in the appropriate box.
 - (f) Click on the box next to **Numbers** if you want files of the run created as described in the **Numbers** section above.
 - (g) If you want to start the arm from the anthropomorphic home position, click on the circle next to **Home Position**.
 - (h) If you want any other starting position, click on the circle next to **Specify Degrees** and enter the starting joint angles in degrees for each joint. Reference the angles to Fig. 7-13.
 - (i) Click on the button **OK**.
- (8) Click on the circle next to **Simulation**. If the circle next to **Real Motion** is black no simulation will occur, the program will set the necessary flags for the VME to use the same positioning routines as are used in simulations to actually move the manipulator, these positioning routines have, however, been compiled on the VME using OS-9 instead of LightSpeedC.
- (9) Click on the **GO** button. If **Simulation** was chosen a window will appear. Then stick figures of the manipulator will appear in the window. They represent the top, front, and side view of the arm. A variety of things may happen now:

(a) If the manipulator was successful in completing the path, a beep will occur.

(b) If a joint limit occurred, a dialog will appear telling you which joint and which limit was reached. Click in the **Continue** button to continue.

(c) If a velocity limit was reached, a dialog will appear telling you so. Click in the **Continue** button to continue.

(d) A system error occurred. Push the programmer's button on the side of the Macintosh to restart the computer.

(10) If the stick figure images are all overlapping each other, click on the button **Graphics Options**. Manipulate the figures according to the **Graphics Options** section above.

(11) You may alter the path you created by choosing **Alter Path**. (Be sure to leave that path file by clicking in the box in the upper left hand corner). All of the items in the **Edit** menu work. That is, you can cut and paste, etc.

(14) You may save the run by clicking on **Save Run**. You will be queried for a run name. The files described in the **Set Up Run Features** dialog (i.e. theAngles, thePos, etc.) will be saved and their names will be appended to the name you chose as a run name.

(15) You may choose **Quit Robotics** from the **File** menu of the robotics application to leave the robotics program. Then you may choose **Open** from the **File** menu of the LightSpeedC application and open your path file and edit it. You may look at your other path files (i.e. theAngles, thePos, etc.).

(15) If you go back into the robotics program and choose some path for the manipulator to follow, you must set up the run by choosing **Set Up Run Features**. You will find that the last angles you specified for the run are automatically entered for you.

(16) If you want to start changing other parts of the program besides the functions on the **GetGradientH.c** module, be sure to make copies first.

CHAPTER 8

CONCLUSIONS AND RECOMMENDATIONS

8.1. Conclusions

A computationally efficient real-time robotic control scheme was developed for the seven-degree-of-freedom NASA Laboratory Telerobotic Manipulator. This scheme determines the joint velocities required to follow a specified end-effector trajectory while optimizing a given performance criterion using the gradient projection method.

Real-time resolution of redundancy of the LTM was accomplished through an improvement over the Dubey-Euler-Babcock implementation of the gradient projection method [8]. It reduces the number of calculations for the inverse kinematics of the LTM by 43% by calculating the self-motion of the manipulator in a more efficient manner.

Trajectory may be planned in Cartesian or joint space using this software. Starting, stopping, and transitions between two path segments are all done in a smooth fashion by ensuring continuity of position, velocity, and acceleration. The complete trajectory does not have to be precalculated and only requires that the current position and the next position be known, which allows for immediate changes in trajectory based on sensory feedback.

A user friendly interface was developed to control the LTM. The interface software creates result files of trajectory information which are available to the user for inspection while running the program. Performance criteria and trajectory related information can be added or modified easily. A graphical representation capability showing three orthogonal views of the LTM was developed which permits the operator to investigate the feasibility of control schemes before implementation. A set of instructions for the interface is provided to start using and to get the most out of the robotic control software.

The software was written in the C programming language for speed and ease of portability. The modular approach to programming was utilized to permit different types of user interfaces to be developed independent of the robotic control software. Additionally, the concept of a common data buffer which contains global variables was utilized to facilitate inclusion of the program into the overall system software.

The robotic control scheme has been shown to be both feasible and effective through real-time graphical simulations demonstrating obstacle avoidance, joint limit avoidance, and singularity avoidance. An earlier version of the software has been

successfully ported from the Macintosh environment in which it was developed to the OS-9 environment on the VME. Integration of the final version into the overall system software is currently in progress.

8.2. Recommendations

Multiple-purpose performance criteria should be developed which take into account a multiplicity of desired LTM performance. Such criteria should include the joint limits, internal singularities, obstacle information, and improved dynamic performance of the LTM.

The overall speed of the control software can be improved by the use of multiple processors. A single microprocessor may be devoted to the calculation of the performance criteria. Another microprocessor may be devoted to trajectory planning, inverse kinematics, and error checking calculations.

Trajectory planning and tracking to avoid obstacles can be developed using sensory feedback so that the trajectory is automatically adjusted when encountering an obstacle.

Currently the joint velocity limits in the control scheme are assumed to be constant, however, these limits actually change with the configuration of the LTM and the load and, therefore, should be based on actuator torques .

The user interface can be improved by taking greater advantage of computer graphics capabilities. For instance, currently the orientation of the end-effector is given by the user through specifying the elements of a rotation matrix referenced to the inertial coordinate system. Instead, a graphical representation of the manipulator could be moved around on the screen. When the end-effector is in the desired orientation, forward kinematics could be used to calculate the orientation matrix.

LIST OF REFERENCES

REFERENCES

1. Herndon, J.N., Babcock, S.M., Butler, P.L., Coctello, H.M., Glassell, R.L., Kress, R.L., Kuban, D.P., Rowe, J.C., Williams, D.M. "Telerobotic Manipulator Developments for Ground-Based Space Research." Proceedings of the NASA Conference on Space Telerobotics, January 31-February 2, 1989, Pasadena, California.
2. Whitney, D.E. "Resolved Motion Rate Control of Manipulators and Human Prostheses." *IEEE Trans. Man-Machine Systems* (1969-a)
3. Denavit, J. and Hartenberg, R.S. 1955 "A Kinematic Notation for Lower Pair Mechanisms Bases On Matrices." *J. Applied Mechanics* 22
4. Noble, B., Daniel, J.W. 1966 Applied Linear Algebra Second Edition, Prentice-Hall, Inc. Englewood Cliffs, N.J. 338-339
5. Noble, Ben. 1966 Applied Linear Algebra Preliminary Edition, Prentice-Hall, Inc. Englewood Cliffs, N.J. C-04123 . 257
6. Klein, C.A., Huang, C.H. "Review of Pseudo-inverse Control for use with Kinematically Redundant Manipulators". *IEEE Trans. Systems, Man, Cybern.*, SMC-13, No. 13, 245-50 (1983)
7. Dubey, R.V., Euler, J.A., Babcock, S.M., and Glassell, R.L. 1988 "Real Time Implementation of a Kinematic Optimization Scheme for Seven-Degree-of-Freedom Redundant Robots with Spherical Wrists". *The American Control Conference, Atlanta, Georgia June 15-17, 1988.*
8. Dubey, R.V., Euler, J.A., and Babcock, S.M. 1988. "An Efficient Gradient Projection Optimization Scheme for a Seven-Degree-Of-Freedom Redundant Robot With a Spherical Wrist." *IEEE International Conference on Robotics and Automation April 25-29 1988.*
9. Pieper, D.L. The Kinematics of Manipulators Under Computer Control. *Doctoral Dissertation, Stanford University (1968)*
10. Renaud, M. 1980. Contribution a la modelisation et la commande dynamic des robots manipulators. Ph.D. Thesis, Universite Paul Sabatier de Toulouse.
11. Dubey, R.V., Euler, J.A., Magness, R.B., Babcock, S.M., and Herndon, J.N. "Robotic Control of the Seven-Degree-Of-Freedom NASA Laboratory Telerobotic Manipulator". Proceedings of the NASA Conference on Space Telerobotics. January 31- February 2, 1989. Pasadena, California.
12. Euler, J.A., Dubey, R.V., and Babcock, S.M. "Self Motion Determination Based on Joint Velocity Bounds for Redundant Robots." *Second International Symposium Robotics and Manufacturing Research , Education and Applications, Albuquerque, New Mexico Nov. 16-18 1988*
13. Paul R.P. Robot Manipulators: Mathematics, Programming, and Control Cambridge: MIT Press

14. Euler, J.A., and Dubey, R.V. "A Comparison of Two Real-Time Control Schemes for Redundant Manipulators with Bounded Joint Velocities" *IEEE International Conference on Robotics and Automation*, Scottsdale, Arizona May 1989.

APPENDIXES

Appendix A

Inverse Kinematics for a 3-Degree-Of-Freedom Manipulator

Figure A-1 shows a three-degree-of-freedom manipulator with coordinate frames attached to the links, and a table of the Danevit-Hartenberg parameters. From these parameters we can write the homogeneous transformation matrices which relate coordinate frame i to coordinate frame $i-1$. These matrices have traditionally been called the A matrices and the particular relationship is denoted using subscripts and superscripts in the following manner: A_i^{i-1} . These matrices are

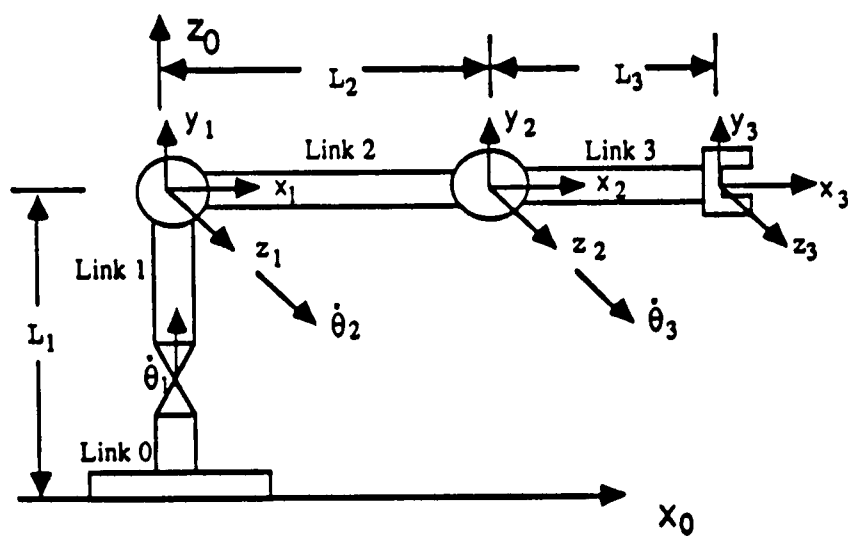
$$(A.1) \quad A_1^0 = \begin{bmatrix} c_1 & 0 & s_1 & 0 \\ s_1 & 0 & -c_1 & 0 \\ 0 & 1 & 0 & L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2^1 = \begin{bmatrix} c_2 & -s_2 & 0 & L_2 c_2 \\ s_2 & c_2 & 0 & L_2 s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A.2)$$

$$A_3^2 = \begin{bmatrix} c_3 & -s_3 & 0 & L_3 c_3 \\ s_3 & c_3 & 0 & L_3 s_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A.3)$$

where the c_i and the s_i denote $\cos(\theta_i)$ and $\sin(\theta_i)$ respectively. The 3×3 submatrix in the upper left hand corner is the rotation matrix which gives the direction cosines of the axes of coordinate system i with respect to the axes of coordinate system $i-1$. The first three elements of the last column is the position of the origin of coordinate system i with respect to coordinate system $i-1$.

The homogeneous transformation matrices which relate coordinate frames of the manipulator links to the base or world coordinate frame have traditionally been termed the T



Denavit-Hartenberg Parameters				
Link _i	α_i	a_i	d_i	θ_i
1	90	0	L_1	θ_1
2	0	L_2	0	θ_2
3	0	L_3	0	θ_3

Figure A-1 : A 3-degree-of-freedom manipulator

matrices and any particular relationship is denoted using subscripts and superscripts in the following manner: T_i^0 . These matrices are

$$T_2^0 = A_1^0 A_2^1 \quad (A.4)$$

$$T_2^0 = \begin{bmatrix} c_1 c_2 & -c_1 s_2 & s_1 & L_2 c_1 c_2 \\ s_1 c_2 & -s_1 s_2 & -c_1 & L_2 s_1 c_2 \\ s_2 & c_2 & 0 & L_2 s_2 + L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A.5)$$

$$T_3^0 = A_1^0 A_2^1 A_3^2 \quad (A.6)$$

$$T_3^0 = \begin{bmatrix} c_1 c_{23} & -c_1 s_{23} & s_1 & L_2 c_1 c_2 + L_3 c_1 c_{23} \\ s_1 c_{23} & -s_1 s_{23} & -c_1 & L_2 s_1 c_2 + L_3 s_1 c_{23} \\ s_{23} & c_{23} & 0 & L_2 s_2 + L_3 s_{23} + L_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (A.7)$$

where the c_{ij} and the s_{ij} represent $\cos(\theta_i + \theta_j)$ and $\sin(\theta_i + \theta_j)$ respectively. If the joint angles are known, the position and orientation of the end-effector can be known with respect to the base (inertial) coordinate system. This is known as the forward kinematics problem.

The equation which relates the linear and angular velocities of the end-effector to the joint velocities is

$$\dot{\mathbf{x}} = \mathbf{J} \dot{\boldsymbol{\theta}}, \quad (A.8)$$

where $\dot{\mathbf{x}}$ is the linear and angular velocity vector of the end-effector, $\dot{\boldsymbol{\theta}}$ is the joint velocity vector, and $\mathbf{J}$ is known as the manipulator Jacobian. The columns of the manipulator Jacobian give the contribution that each joint makes to the motion of the end-effector (for a unit angle): column 1 for joint 1, column 2 for joint 2, et cetera. The general formula for finding the Jacobian is

$$\begin{bmatrix} J_{Lj} \\ J_{Aj} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{j-1} \times \mathbf{I}_{j-1,e} \\ \mathbf{b}_{j-1} \end{bmatrix} \quad (A.9)$$

where I_{Lj} and I_{Aj} refer to the contributions to the linear and angular velocities of the end-effector due to the j th joint, b_{j-1} is the joint axis direction for Θ_j , and $r_{j-1,e}$ is the position vector from the origin of that axis to the origin of the axes fixed to the end-effector. Now

$$b_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad b_1 = \begin{bmatrix} s_1 \\ -c_1 \\ 0 \end{bmatrix}, \quad b_2 = \begin{bmatrix} s_1 \\ -c_1 \\ 0 \end{bmatrix}, \quad (A.10)$$

$$I_0 = \begin{bmatrix} L_2 c_1 c_2 + L_3 c_1 c_{23} \\ L_2 s_1 c_2 + L_3 s_1 c_{23} \\ L_1 + L_2 s_2 + L_3 s_{23} \end{bmatrix}, \quad I_1 = \begin{bmatrix} L_2 c_1 c_2 + L_3 c_1 c_{23} \\ L_2 s_1 c_2 + L_3 s_1 c_{23} \\ L_2 s_2 + L_3 s_{23} \end{bmatrix}, \quad I_2 = \begin{bmatrix} L_3 c_1 c_{23} \\ L_3 s_1 c_{23} \\ L_3 s_{23} \end{bmatrix}, \quad (A.11)$$

Using equations (A.9), (A.10), and (A.11) we obtain

$$J_s = \begin{bmatrix} -s_1(L_2 c_2 + L_3 c_{23}) & -c_1(L_2 s_2 + L_3 s_{23}) & -L_3 c_1 s_{23} \\ c_1(L_2 c_2 + L_3 c_{23}) & s_1(L_2 s_2 + L_3 s_{23}) & L_3 s_1 s_{23} \\ 0 & L_2 c_2 + L_3 c_{23} & L_3 c_{23} \\ 0 & s_1 & s_1 \\ 0 & -c_1 & -c_1 \\ 1 & 0 & 0 \end{bmatrix}. \quad (A.12)$$

Note that J_s is a rectangular matrix and therefore cannot be inverted. Formula (A.9) was used to develop the Jacobian in order to demonstrate the general case. With three more degrees of freedom the Jacobian would become square.

We would like to solve $\dot{\underline{x}} = J \dot{\underline{\theta}}$, which is the inverse kinematics problem. In general in order to move an end-effector in an arbitrary direction with an arbitrary angular velocity six degrees of freedom are required. Because this manipulator has only three degrees of freedom, only three elements of $\dot{\underline{x}}$ can be specified, the others are then automatically determined.

We may choose $\dot{\underline{x}}$ so that it has a full set of velocity components and then choose a subset of $\dot{\underline{x}}$:

(1) Let

$$\dot{\mathbf{x}} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \quad \dot{\mathbf{q}} = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix}, \text{ and } L_1 = L_2 = L_3 = 1. \quad (\text{A.13})$$

(2) If we specify ω_z

$$\dot{q}_1 = \omega_z \quad (\text{A.14})$$

(3) Multiply row 1 by $-c_1$ and row 2 by $-s_1$, and then add rows 1 and 2.

$$c_1(s_1c_2 + s_1c_{23})\dot{q}_1 + c_1^2(s_{23} + s_2)\dot{q}_2 + c_1^2s_{23}\dot{q}_3 = -c_1v_x \quad (\text{A.15})$$

$$-s_1(c_1c_2 + c_1c_{23})\dot{q}_1 + s_1^2(s_{23} + s_2)\dot{q}_2 + s_1^2s_{23}\dot{q}_3 = -s_1v_y \quad (\text{A.16})$$

$$(s_{23} + s_2)\dot{q}_2 + s_{23}\dot{q}_3 = -(c_1v_x + s_1v_y) \quad (\text{A.17})$$

(4) Eliminating $\dot{q}_3$ from row three and (A.17) we have

$$\dot{q}_2 = \frac{c_{23}(s_1v_y + c_1v_x) + s_{23}v_z}{s_3} \quad (\text{A.18})$$

(5) Using the 3rd row we solve for $\dot{q}_3$

$$(c_{23} + c_2)\dot{q}_2 + c_{23}\dot{q}_3 = v_z \quad (\text{A.19})$$

$$\dot{q}_3 = \frac{v_z - (c_{23} + c_2)\dot{q}_2}{c_{23}} \quad (\text{A.20})$$

(5) We cannot specify ω_z , v_x , v_y , and v_z . Using rows 1 and 2 again we multiply row 1 by $-s_1$ and row 2 by c_1 , and add, which eliminates $\dot{\theta}_2$ and $\dot{\theta}_3$ giving

$$(c_{23} + c_2)\dot{\theta}_1 = c_1 v_y - s_1 v_x \quad (\text{A.21})$$

(6) We now write

$$\dot{\theta}_1 = \omega_z \quad (\text{A.22})$$

$$v_y = \frac{(c_{23} + c_2)\dot{\theta}_1 + s_1 v_x}{c_1} \quad (\text{A.23})$$

$$\dot{\theta}_2 = \frac{c_{23}(s_1 v_y + c_1 v_x) + s_{23} v_z}{s_3} \quad (\text{A.24})$$

$$\dot{\theta}_3 = \frac{v_z - (c_{23} + c_2)\dot{\theta}_2}{c_{23}} \quad (\text{A.25})$$

In the example just given an angular rotation and two linear velocities of the manipulator were arbitrary. If we wish to have all three linear velocities arbitrary we may obtain a square Jacobian using

$$[J_{Lj}] = [\mathbf{b}_{j-1} \times \mathbf{r}_{j-1}, \mathbf{e}] \quad (\text{A.26})$$

where the orientation of the manipulator at any particular position is not arbitrary. This Jacobian is given by

$$J = \begin{bmatrix} -s_1(L_2 c_2 + L_3 c_{23}) & -c_1(L_2 s_2 + L_3 s_{23}) & -L_3 c_1 s_{23} \\ c_1(L_2 c_2 + L_3 c_{23}) & s_1(L_2 s_2 + L_3 s_{23}) & L_3 s_1 s_{23} \\ 0 & L_2 c_2 + L_3 c_{23} & L_3 c \end{bmatrix} \quad (\text{A.27})$$

the rows of which are the same as the first three rows of J_s . Using rows 1, 2, and 3 of the Jacobian J we may obtain the following three equations

$$\dot{\theta}_1 = \frac{s_1 v_x - c_1 v_y}{(c_{23} + c_2)} \quad (\text{A.28})$$

$$\dot{\theta}_2 = \frac{c_{23}(s_1 v_y + c_1 v_x) + s_{23} v_z}{s_3} \quad (\text{A.29})$$

$$\dot{\theta}_3 = \frac{v_z - (c_{23} + c_2)\dot{\theta}_2}{c_{23}} \quad (\text{A.30})$$

Appendix B

The A Matrices of the LTM

Note: These are with reference to Fig. 4-1.

$$A_1^0(\Theta_1) = \begin{bmatrix} c_1 & 0 & -s_1 & 0 \\ s_1 & 0 & c_1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_2^1(\Theta_2) = \begin{bmatrix} c_2 & 0 & s_2 & a_2 c_2 \\ s_2 & 0 & -c_2 & a_2 s_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3^2(\Theta_3) = \begin{bmatrix} c_3 & 0 & s_3 & 0 \\ s_3 & 0 & -c_3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_4^3(\Theta_4) = \begin{bmatrix} c_4 & 0 & -s_4 & a_4 c_4 \\ s_4 & 0 & c_4 & a_4 s_4 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_5^4(\Theta_5) = \begin{bmatrix} c_5 & 0 & s_5 & 0 \\ s_5 & 0 & -c_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_6^5(\Theta_6) = \begin{bmatrix} c_6 & 0 & s_6 & 0 \\ s_6 & 0 & -c_6 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_7^6(\Theta_7) = \begin{bmatrix} c_7 & -s_7 & 0 & 0 \\ s_7 & c_7 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix C

The T Matrices of the LTM

The product of the A matrices starting from the base and working out to the end-effector.
Note that s_i refers to $\sin(\Theta_i)$ and c_i refers to $\cos(\Theta_i)$.

$$T_2^0 = \begin{bmatrix} c_1 c_2 & -s_1 & c_1 s_2 & a_2 c_1 c_2 \\ s_1 c_2 & c_1 & s_1 s_2 & a_2 s_1 c_2 \\ -s_2 & 0 & c_2 & -a_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^0 = \begin{bmatrix} c_1 c_2 c_3 - s_1 s_3 & c_1 s_2 & c_1 c_2 s_3 + s_1 c_3 & a_2 c_1 c_2 \\ s_1 c_2 c_3 + c_1 s_3 & s_1 s_2 & s_1 c_2 s_3 - c_1 c_3 & a_2 s_1 c_2 \\ -s_2 c_3 & c_2 & -s_2 s_3 & -a_2 s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^0 = \begin{bmatrix} n_{x4} & t_{x4} & b_{x4} & p_{x4} \\ n_{y4} & t_{y4} & b_{y4} & p_{y4} \\ n_{z4} & t_{z4} & b_{z4} & p_{z4} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} n_{x4} &= c_1 c_2 c_3 c_4 - s_1 s_3 c_4 + c_1 s_2 s_4 \\ n_{y4} &= s_1 c_2 c_3 c_4 + c_1 s_3 c_4 + s_1 s_2 s_4 \\ n_{z4} &= -s_2 c_3 c_4 + c_2 s_4 \\ t_{x4} &= -c_1 c_2 s_3 - s_1 c_3 \\ t_{y4} &= -s_1 c_2 s_3 + c_1 c_3 \\ t_{z4} &= s_2 s_3 \\ b_{x4} &= -c_1 c_2 c_3 s_4 + s_1 s_3 s_4 + c_1 s_2 c_4 \\ b_{y4} &= -s_1 c_2 c_3 s_4 - c_1 s_3 s_4 + s_1 s_2 c_4 \\ b_{z4} &= s_2 c_3 s_4 + c_2 c_4 \\ p_{x4} &= a_4 (c_1 c_2 c_3 c_4 - s_1 s_3 c_4 + c_1 s_2 s_4) + a_2 c_1 c_2 \\ p_{y4} &= a_4 (s_1 c_2 c_3 c_4 + c_1 s_3 c_4 + s_1 s_2 s_4) + a_2 s_1 c_2 \\ p_{z4} &= -a_4 s_2 c_3 c_4 + a_4 c_2 s_4 - a_2 s_2 \end{aligned}$$

$$T_5^0 = \begin{bmatrix} n_{x5} & t_{x5} & b_{x5} & p_{x5} \\ n_{y5} & t_{y5} & b_{y5} & p_{y5} \\ n_{z5} & t_{z5} & b_{z5} & p_{z5} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} n_{x5} &= C_5(C_1C_2C_3C_4 - S_1S_3C_4 + C_1S_2S_4) - S_5(C_1C_2S_3 + S_1C_3) \\ n_{y5} &= C_5(S_1C_2C_3C_4 + C_1S_3C_4 + S_1S_2S_4) - S_5(S_1C_2S_3 + C_1C_3) \\ n_{z5} &= -S_2C_3C_4C_5 + C_2S_4C_5 + S_2S_3S_5 \\ t_{x5} &= -C_1C_2C_3S_4 + S_1S_3S_4 + C_1S_2C_4 \\ t_{y5} &= -S_1C_2C_3S_4 - C_1S_3S_4 + S_1S_2C_4 \\ t_{z5} &= S_2C_3S_4 + C_2C_4 \\ b_{x5} &= S_5(C_1C_2C_3C_4 - S_1S_3C_4 + C_1S_2S_4) + C_5(C_1C_2S_3 + S_1C_3) \\ b_{y5} &= S_5(S_1C_2C_3C_4 + C_1S_3C_4 + S_1S_2S_4) + C_5(S_1C_2S_3 - C_1C_3) \\ b_{z5} &= -S_2C_3C_4S_5 + C_2S_4S_5 - S_2S_3C_5 \\ p_{x5} &= a_4(C_1C_2C_3C_4 - S_1S_3C_4 + C_1S_2S_4) + a_2C_1C_2 \\ p_{y5} &= a_4(S_1C_2C_3C_4 + C_1S_3C_4 + S_1S_2S_4) + a_2S_1C_2 \\ p_{z5} &= -a_4S_2C_3C_4 + a_4C_2S_4 - a_2S_2 \end{aligned}$$

$$T_6^0 = \begin{bmatrix} n_{x6} & t_{x6} & b_{x6} & p_{x6} \\ n_{y6} & t_{y6} & b_{y6} & p_{y6} \\ n_{z6} & t_{z6} & b_{z6} & p_{z6} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} n_{x6} &= C_5 C_6 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) - S_5 C_6 (C_1 C_2 S_3 + S_1 C_3) \\ &\quad - S_4 S_6 (C_1 C_2 C_3 - S_1 S_3) + C_1 S_2 C_4 S_6 \\ n_{y6} &= C_5 C_6 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) - S_5 C_6 (S_1 C_2 S_3 - C_1 C_3) \\ &\quad - S_4 S_6 (S_1 C_2 C_3 + C_1 S_3) + S_1 S_2 C_4 S_6 \\ n_{z6} &= C_6 (-S_2 C_3 C_4 C_5 + C_2 S_4 C_5 + S_2 S_3 S_5) + S_6 (S_2 C_3 S_4 + C_2 C_4) \\ t_{x6} &= S_5 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) + C_5 (C_1 C_2 S_3 + S_1 C_3) \\ t_{y6} &= S_5 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) + C_5 (S_1 C_2 S_3 - C_1 C_3) \\ t_{z6} &= -S_2 C_3 C_4 S_5 + C_2 S_4 S_5 - S_2 S_3 C_5 \\ b_{x6} &= C_5 S_6 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) - S_5 S_6 (C_1 C_2 S_3 + S_1 C_3) \\ &\quad + S_4 C_6 (C_1 C_2 C_3 - S_1 S_3) - C_1 S_2 C_4 C_6 \\ b_{y6} &= C_5 S_6 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) - S_5 S_6 (S_1 C_2 S_3 - C_1 C_3) \\ &\quad + S_4 C_6 (S_1 C_2 C_3 + C_1 S_3) - S_1 S_2 C_4 C_6 \\ b_{z6} &= S_6 (-S_2 C_3 C_4 C_5 + C_2 S_4 C_5 + S_2 S_3 S_5) - C_6 (S_2 C_3 S_4 + C_2 C_4) \\ p_{x6} &= a_4 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) + a_2 C_1 C_2 \\ p_{y6} &= a_4 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) + a_2 S_1 C_2 \\ p_{z6} &= -a_4 S_2 C_3 C_4 + a_4 C_2 S_4 - a_2 S_2 \end{aligned}$$

$$T_7^0 = \begin{bmatrix} n_{x7} & t_{x7} & b_{x7} & p_{x7} \\ n_{y7} & t_{y7} & b_{y7} & p_{y7} \\ n_{z7} & t_{z7} & b_{z7} & p_{z7} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

$$\begin{aligned} n_{x7} &= C_5 C_6 C_7 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) - S_5 C_6 C_7 (C_1 C_2 S_3 + S_1 C_3) \\ &\quad - S_6 C_7 (C_1 C_2 C_3 S_4 - S_1 S_3 S_4 - C_1 S_2 C_4) + S_5 S_7 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) \\ &\quad + C_1 C_2 S_3 C_5 S_7 + S_1 C_3 C_5 S_7 \\ n_{y7} &= C_5 C_6 C_7 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) - S_5 C_6 C_7 (S_1 C_2 S_3 - C_1 C_3) \\ &\quad - S_6 C_7 (S_1 C_2 C_3 S_4 + C_1 S_3 S_4 - S_1 S_2 C_4) + S_5 S_7 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) \\ &\quad + S_1 C_2 S_3 C_5 S_7 - C_1 C_3 C_5 S_7 \\ n_{z7} &= C_6 C_7 (-S_2 C_3 C_4 C_5 + C_2 S_4 C_5 + S_2 S_3 S_5) + S_6 C_7 (S_2 C_3 S_4 + C_2 C_4) \\ &\quad S_5 S_7 (-S_2 C_3 C_4 + C_2 S_4) - S_2 S_3 C_5 S_7 \\ t_{x7} &= C_5 C_6 S_7 (-C_1 C_2 C_3 C_4 + S_1 S_3 C_4 - C_1 S_2 S_4) + S_5 C_6 S_7 (C_1 C_2 S_3 + S_1 C_3) \\ &\quad S_6 S_7 (C_1 C_2 C_3 S_4 - S_1 S_3 S_4 - C_1 S_2 C_4) + S_5 C_7 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) \\ &\quad C_5 C_7 (C_1 C_2 S_3 + S_1 C_3) \\ t_{y7} &= C_5 C_6 S_7 (-S_1 C_2 C_3 C_4 - C_1 S_3 C_4 - S_1 S_2 S_4) + S_5 C_6 S_7 (S_1 C_2 S_3 - C_1 C_3) \\ &\quad S_6 S_7 (S_1 C_2 C_3 S_4 + C_1 S_3 S_4 - S_1 S_2 C_4) + S_5 C_7 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) \\ &\quad C_5 C_7 (S_1 C_2 S_3 - C_1 C_3) \\ t_{z7} &= C_6 S_7 (S_2 C_3 C_4 C_5 - C_2 S_4 C_5 - S_2 S_3 S_5) - S_6 S_7 (S_2 C_3 S_4 + C_2 C_4) \\ &\quad S_5 C_7 (-S_2 C_3 C_4 + C_2 S_4) - S_2 S_3 C_5 C_7 \\ b_{x7} &= C_5 S_6 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) - S_5 S_6 (C_1 C_2 S_3 + S_1 C_3) \\ &\quad S_4 C_6 (C_1 C_2 C_3 - S_1 S_3) - C_1 S_2 C_4 C_6 \\ b_{y7} &= C_5 S_6 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) - S_5 S_6 (S_1 C_2 S_3 - C_1 C_3) \\ &\quad S_4 C_6 (S_1 C_2 C_3 + C_1 S_3) - S_1 S_2 C_4 C_6 \\ b_{z7} &= C_5 S_6 (-S_2 C_3 C_4 + C_2 S_4) + S_2 S_3 S_5 S_6 - S_2 C_3 S_4 C_6 - C_2 C_4 C_6 \\ p_{x7} &= a_4 (C_1 C_2 C_3 C_4 - S_1 S_3 C_4 + C_1 S_2 S_4) + a_2 C_1 C_2 \\ p_{y7} &= a_4 (S_1 C_2 C_3 C_4 + C_1 S_3 C_4 + S_1 S_2 S_4) + a_2 S_1 C_2 \\ p_{z7} &= a_4 (-S_2 C_3 C_4 + C_2 S_4) - a_2 S_2 \end{aligned}$$

Appendix D

THE (wrist) JACOBIAN (Referred to base frame)

Because the manipulator has angular joints the formula used is

$$\begin{bmatrix} \mathbf{J}_{Lj} \\ \mathbf{J}_{Aj} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_{j-1} \times \mathbf{r}_{j-1,w} \\ \mathbf{b}_{j-1} \end{bmatrix} \quad (\text{D.1})$$

where $\mathbf{J}_{Lj}$ and $\mathbf{J}_{Aj}$ refer to the contributions to the linear and angular velocities, respectively, of the wrist due to the j th joint, $\mathbf{b}_{j-1}$ is the joint axis direction for Θ_j , and $\mathbf{r}_{j-1,w}$ is the position vector from the origin of that axis to the origin of the axes fixed to the wrist. Then:

$$\begin{bmatrix} \mathbf{J}_{L1} \\ \mathbf{J}_{A1} \end{bmatrix} = \begin{bmatrix} -a_4(s_1c_2c_3c_4 + c_1s_3c_4 + s_1s_2s_4) - a_2s_1c_2 \\ a_4(c_1c_2c_3c_4 - s_1s_3c_4 + c_1s_2s_4) + a_2c_1c_2 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (\text{D.2})$$

$$\begin{bmatrix} \mathbf{J}_{L2} \\ \mathbf{J}_{A2} \end{bmatrix} = \begin{bmatrix} -a_4c_1(s_2c_3c_4 - c_2s_4) - a_2c_1s_2 \\ -a_4s_1(s_2c_3c_4 - c_2s_4) - a_2s_1s_2 \\ -a_4(c_2c_3c_4 + s_2s_4) - a_2c_2 \\ -s_1 \\ c_1 \\ 0 \end{bmatrix} \quad (\text{D.3})$$

$$\begin{bmatrix} \mathbf{J}_{L3} \\ \mathbf{J}_{A3} \end{bmatrix} = \begin{bmatrix} -a_4c_4(c_1c_2s_3 + s_1c_3) \\ a_4c_4(c_1c_3 - s_1c_2s_3) \\ a_4s_2s_3c_4 \\ c_1s_2 \\ s_1s_2 \\ c_2 \end{bmatrix} \quad (\text{D.4})$$

$$\begin{bmatrix} J_{L4} \\ J_{A4} \end{bmatrix} = \begin{bmatrix} a_4(s_1 s_3 s_4 + c_1 s_2 c_4 - c_1 c_2 c_3 s_4) \\ a_4(s_1 s_2 c_4 - c_1 s_3 s_4 - s_1 c_2 c_3 s_4) \\ a_4(c_2 c_4 + s_2 c_3 s_4) \\ c_1 c_2 s_3 + s_1 c_3 \\ s_1 c_2 s_3 - c_1 c_3 \\ -s_2 s_3 \end{bmatrix} \quad (D.5)$$

$$\begin{bmatrix} J_{L5} \\ J_{A5} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -c_1 c_2 c_3 s_4 + s_1 s_3 s_4 + c_1 s_2 c_4 \\ -s_1 c_2 c_3 s_4 - c_1 s_3 s_4 + s_1 s_2 c_4 \\ s_2 c_3 s_4 + c_2 c_4 \end{bmatrix} \quad (D.6)$$

$$\begin{bmatrix} J_{L6} \\ J_{A6} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ s_5(c_1 c_2 c_3 c_4 - s_1 s_3 c_4 + c_1 s_2 s_4) + c_5(c_1 c_2 s_3 + s_1 c_3) \\ s_5(s_1 c_2 c_3 c_4 + c_1 s_3 c_4 + s_1 s_2 s_4) + c_5(s_1 c_2 s_3 - c_1 c_3) \\ -s_2 c_3 c_4 s_5 + c_2 s_4 s_5 - s_2 s_3 c_5 \end{bmatrix} \quad (D.7)$$

$$\begin{bmatrix} J_{L7} \\ J_{A7} \end{bmatrix} = \begin{bmatrix} J_{L17} & J_{L27} & J_{L37} & J_{A47} & J_{A57} & J_{A67} \end{bmatrix}^T \quad (D.8)$$

where.

$$\begin{aligned} J_{L17} &= 0 \\ J_{L27} &= 0 \\ J_{L37} &= 0 \\ J_{A47} &= c_5 s_6(c_1 c_2 c_3 c_4 - s_1 s_3 c_4 + c_1 s_2 s_4) - s_5 s_6(c_1 c_2 s_3 + s_1 c_3) \\ &\quad + s_4 c_6(c_1 c_2 c_3 - s_1 s_3) - c_1 s_2 c_4 c_6 \\ J_{A57} &= c_5 s_6(s_1 c_2 c_3 c_4 + c_1 s_3 c_4 + s_1 s_2 s_4) - s_5 s_6(s_1 c_2 s_3 - c_1 c_3) \\ &\quad + s_4 c_6(s_1 c_2 c_3 + c_1 s_3) - s_1 s_2 c_4 c_6 \\ J_{A67} &= s_6(-s_2 c_3 c_4 c_5 + c_2 s_4 c_5 + s_2 s_3 s_5) - c_6(s_2 c_3 s_4 + c_2 c_4) \end{aligned}$$

Appendix E

THE MIDFRAME (wrist) JACOBIAN (Referred to frame 3)

Let the velocity and angular velocity of the wrist referred to the base frame be designated by

$$\dot{\mathbf{x}}_w = \begin{bmatrix} \mathbf{v}_w \\ \mathbf{w}_w \end{bmatrix}, \quad (\text{E.1})$$

the Jacobian of the wrist referred to the base frame be designated by $\mathbf{J}_w$, and let $\dot{\mathbf{q}}$ be the 7×1 joint velocity vector referred to the base frame, then

$$\dot{\mathbf{x}}_w = \mathbf{J}_w \dot{\mathbf{q}}. \quad (\text{E.2})$$

Let superscript "n" preceeding a vector or array of vectors designates that that vector or array of vectors is referred to the nth coordinate frame, and let $\mathbf{R}_3^0$ designate the rotation matrix that refers a three element vector from the 3rd coordinate frame to the base coordinate frame, then

$${}^3\dot{\mathbf{x}}_w = {}^3\mathbf{J}_w \dot{\mathbf{q}}. \quad (\text{E.3})$$

$$\dot{\mathbf{x}}_w = \begin{bmatrix} \mathbf{R}_3^0 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_3^0 \end{bmatrix} {}^3\dot{\mathbf{x}}_w \quad (\text{E.4})$$

$$\mathbf{J}_w \dot{\mathbf{q}} = \begin{bmatrix} \mathbf{R}_3^0 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_3^0 \end{bmatrix} {}^3\mathbf{J}_w \dot{\mathbf{q}} \quad (\text{E.5})$$

and therefore

$$\mathbf{J}_w = \begin{bmatrix} \mathbf{R}_3^0 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_3^0 \end{bmatrix} {}^3\mathbf{J}_w \quad (\text{E.7})$$

where $\mathbf{R}_3^0$ is the rotation matrix defined as

$$\mathbf{R}_3^0 = \begin{bmatrix} c_1 c_2 c_3 - s_1 s_3 & c_1 s_2 & c_1 c_2 s_3 + s_1 c_3 \\ s_1 c_2 c_3 + c_1 s_3 & s_1 s_2 & s_1 c_2 s_3 - c_1 c_3 \\ -s_2 c_3 & c_2 & -s_2 s_3 \end{bmatrix} \quad (\text{E.8})$$

and

$$[\mathbf{R}_3^0]^{-1} = [\mathbf{R}_3^0]^T = [\mathbf{R}_0^3] = \begin{bmatrix} c_1 c_2 c_3 - s_1 s_3 & s_1 c_2 c_3 + c_1 s_3 & -s_2 c_3 \\ c_1 s_2 & s_1 s_2 & c_2 \\ c_1 c_2 s_3 + s_1 c_3 & s_1 c_2 s_3 - c_1 c_3 & -s_2 s_3 \end{bmatrix} \quad (\text{E.9})$$

Then

$${}^3\mathbf{I}_w = \begin{bmatrix} \mathbf{R}_3^0 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_3^0 \end{bmatrix}^{-1} \mathbf{I}_w = \begin{bmatrix} \mathbf{R}_3^0 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_3^0 \end{bmatrix}^T \mathbf{I}_w \quad (\text{E.10})$$

$${}^3\mathbf{I}_w = \begin{bmatrix} \mathbf{R}_0^3 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_0^3 \end{bmatrix} \mathbf{I}_w \quad (\text{E.11})$$

The column vectors of the midframe (wrist) Jacobian are:

$${}^3\mathbf{I}_w = \begin{bmatrix} a_4 s_2 s_3 s_4 + a_2 c_2 s_3 & a_4 c_3 s_4 & 0 & -a_4 s_4 & 0 & 0 & 0 \\ -a_4 s_2 s_3 c_4 & -a_4 c_3 c_4 - a_2 & 0 & a_4 c_4 & 0 & 0 & 0 \\ -a_4 (s_2 c_3 s_4 + c_2 c_4) - a_2 c_2 c_3 & a_4 s_3 s_4 & -a_4 c_4 & 0 & 0 & 0 & 0 \\ -s_2 c_3 & s_3 & 0 & 0 & -s_4 & c_4 s_5 & c_4 c_5 s_6 + s_4 c_6 \\ c_2 & 0 & 1 & 0 & c_4 & s_4 s_5 & s_4 c_5 s_6 - c_4 c_6 \\ -s_2 s_3 & -c_3 & 0 & 1 & 0 & c_5 & -s_5 s_6 \end{bmatrix} \quad -(\text{E.12})$$

Appendix F

User Interface Module (MacInterface.c)

```
#include "QuickDraw.h"
#include "MacTypes.h"
#include "WindowMgr.h"
#include "TextEdit.h"
#include "ControlMgr.h"
#include "EventMgr.h"
#include "DeskMgr.h"
#include "MenuMgr.h"
#include "IntlPkg.h"
#include "ToolboxUtil.h"
#include "pascal.h"
#include "DialogMgr.h"
#include "FileMgr.h"
#include "MacInterface.h"
#include "global_data.h"
#include "LTM.h"
#include "stdio.h"
#include "strings.h"
#include "StdFilePkg.h"
#include "PrintMgr.h"
#include <unix.h>
#include <math.h>
```

```
/* declaration of external globals */
    struct global_data    ltm,*LTM;
/* end of globals */
```

```
/* structure definitions */
struct          long_halves
    {          int high;
              int low;
    };
struct          Path
    {
        char          numNodes[25];
        char          Node[MAXnumNodes*15][25];
    };
/* End of structure definitions */
```

```
/* declaration of globals */
    float          delta_t; /* the time increment */
    int             when,leastnorm,SetGraphics,SetNumbers;
```

```

int
static int
static int
static int
for initial run */
static int
conditions for initial run */
static int
static int
static NODE
structures) */
CONFIG
int
extern struct ViewData
JOINT_ANGLES
float
int
static float
1.57079632,0,1.57079632,0,0,1.57079632,0);
float
FILE
FILE
FILE
FILE
FILE
static struct views
char CartComments[17][25],JointComments[17][25];

Nodes;
doPathEdit;
path_num;
graphicsCond,CpathCond,CsetupCond; /* necessary conditions
JpathCond,JsetupCond,CartGO,JointGO; /* necessary
theCPathChanged = 0,theJPathChanged = 0;
ShowTitles = 0,ShowHome = 0;
CartesianPath[9]; /* demonstration path (array of
JointPath[9];
view_case, drawcount = 0;
ViewArm,ViewArm0;
dTMax,MaxSpeed;
scale; /* expand graph */
iteration = 0;
HomeConfig[8] = {0,-
ThetaInit[8];
*thePath,*thePerfCrit;
*theAngles,*theCricket,*thePos;
*theJointPath,*theJointPPos;
*theJointPAngles,*theJointPCricket;
*OutputW;
view;

```

/* end of globals */

```

EventRecord theEvent;
GrafPtr OutputPort;
Rect CntrlRect,destRect,viewRect;
TEHandle hTE;
DialogRecord DlgRecord;
DialogPtr ControlDlgPtr;
ProcPtr do_radio,do_upaction,do_downaction;
MenuHandle myMenus[6];
float kL,kR;
int MaxpathData;
int refNum;
int NumOpnWdw = 0;
Point theOrigin;
static Point putWhere = {106,104};
static Point getWhere = {90,82};
static SFReply theReply;

```

```

static int positioned_flag;
static Point dialog_position;
static int dialog_flags;
static Rect DesktopRect;

```

```

static int      robotic_dialog;
DialogPtr      CartesianSpaceDlog,JointSpaceDlog;

/*****/

/*****
These things added for the simulation windows and path window
*****/
struct WindowData{
    TEHandle      editRec;
    ControlHandle  VScrollBar;
    ControlHandle  HScrollBar;
    Boolean        dirty;
    Byte           padding;
    int            volNum;
    int            refNum;
    CharsHandle    text;
};

struct WINDOW{
    int            ID;
    WindowPtr      Ptr;
    WindowRecord    Record;
    WindowPeek      Peek;
    PicHandle      Pict;
    struct WindowData Data;
} CartW,JointW,PathW;

OSErr            openCode,readCode,writeCode;

/*****/

pascal void      up_action(), down_action();

/** functions defined on other modules **/
    GetDesiredPosition ();
    GetJointConfig();
    DrawArm();
    PictArm();
    Titles();
/**end of functions defined on other modules**/

main (){
    LTM = &lrm;
    InitProcess ();
    defaultSettings();
    EventLoop (); }
EndFunction

void      InitRoboticDialog(){
    positioned_flag = 0;

```

```

        dialog_flags = 0;
        CartesianSpaceDlog = 0;
        JointSpaceDlog = 0;}
EndFunction

```

```

InitProcess (){
    InitMgrs ();
    Stdio_MacInit(TRUE);
    SetUpMenus();
    FillCartCommentsMat();
    FillJointCommentsMat();
    InitRoboticDialog();}
EndFunction

```

```

InitMgrs (){
    InitGraf (&thePort);
    InitFonts ();
    FlushEvents (everyEvent,0);
    InitWindows ();
    InitCursor ();
    InitMenus();
    TEInit();
    InitDialogs(0L);}
EndFunction

```

```

SetUpMenus(){
    int    i;
        myMenus[FileMenu] = GetMenu(FileID);
        myMenus[EditMenu] = GetMenu(EditID);
        myMenus[TeleRobMenu] = GetMenu(TeleRobID);
        myMenus[RoboticsMenu] = GetMenu(RoboticsID);
        for ( i = FileMenu; i<=RoboticsMenu; i++)
            InsertMenu(myMenus[i],0);
        DrawMenuBar();}
EndFunction

```

```

defaultSettings(){
    LTM->flags.simulate = 1;          /* NOT real motion */
    saveJointRun = 0;                /* no run to initially save */
    saveCartRun = 0;                 /* no run to initially save */
    leastnorm = 0;                   /* to be optimized */
    SetGraphics = 1;                 /* graphics enabled */
    SetNumbers = 1;                  /* result files enabled */
    doPathEdit = 0;                  /* editing disabled */
    view_case = 3;                   /* Top view AND Side view AND Front view */
/* necessary conditions to be SET for initial run */
    graphicsCond = 0;
    CpathCond = 0;
    CsetupCond = 0;
    CartGO = 0;
    JointGO = 0;

```

```

    JpathCond = 0;
    JsetupCond = 0;
    DesktopRect.top = 0;
    DesktopRect.left = 0;
    DesktopRect.right = 512;
    DesktopRect.bottom = 350;}

```

EndFunction

AllocateTEHandle(theWindow)

```

struct WINDOW *theWindow;{
    destRect = (theWindow->Ptr)->portRect;
    viewRect = (theWindow->Ptr)->portRect;
    InsetRect(&destRect,4,0);
    destRect.right -= 16;
    viewRect.right -= 16;
    /* viewRect.bottom -= 16; */
    theWindow->Data.editRec = TENew(&destRect,&viewRect);}

```

EndFunction

EventLoop (){

```

Boolean    valid,validDialog1,validDialog2;
DialogPtr  theDialog;
int        itemHit;
Rect       modeRect, armRect;
int        itemType,charCode;
Handle     item;
Rect       itemBox;
void       doRoboticCartesianDlog();
void       doRoboticJointDlog();
while (1){
    SystemTask();
    if(!CartGO) || (!JointGO)
        IsOKtoGO();
    if(doPathEdit){
        TEIdle(PathW.Data.editRec);
        GetCorrectCursor();}
    valid = GetNextEvent(everyEvent, &theEvent);
    validDialog1 = IsDialogEvent(&theEvent);
    if(valid)
        for(;;){
            if (validDialog1){
                validDialog2 = DialogSelect(&theEvent,&theDialog,&itemHit);
                if (validDialog2){
                    if(theDialog == CartesianSpaceDlog)
                        doRoboticCartesianDlog(itemHit,theDialog);
                    else if(theDialog == JointSpaceDlog)
                        doRoboticJointDlog(itemHit,theDialog);}
                else break;}
            else if (!validDialog1){
                switch(theEvent.what){
                    case nullEvent:

```



```

        break;
    case mouseDown:
        DoMouseDownEvent();
        break;
    case mouseUp:
    case keyDown:
        if(doPathEdit){
            charCode = BitAnd(theEvent.message,charCodeMask);
            TEKey(charCode,PathW.Data.editRec);
            AdjustScrollBar();
            PathW.Data.dirty = 1;}
        break;
    case keyUp:
    case autoKey:
        break;
    case updateEvt:
        DoUpdate(theEvent.message);
        break;
    case diskEvt:
        break;
    case activateEvt:
        DoActivate(theEvent.message);
        break;
    case networkEvt:
    case driverEvt:
    case app1Evt:
    case app2Evt:
    case app3Evt:
    case app4Evt:
    default:
        break;})
    break;}})
EndFunction

```

```

frameItem(DlgWindow,item,radius)
DialogPtr    DlgWindow;
int          item,radius;{
int          itemType;
Handle       itemHandle;
Rect         itemBox,frame;
    GetDItem(DlgWindow,item,&itemType,&itemHandle,&itemBox);
    SetPort(DlgWindow);
    InsetRect(&itemBox,-5,-4);
    PenSize(3,3);
    FrameRoundRect(&itemBox,radius,radius);}
EndFunction

```

```

framePanel(window,firstItem,lastItem)
DialogPtr    window;
int          firstItem,lastItem;{
int          itemType;

```

```

Handle      item1,item2;
Rect        Box1,Box2,frame;
    GetDItem(window,firstItem,&itemType,&item1,&Box1);
    GetDItem(window,lastItem,&itemType,&item2,&Box2);
    SetRect(&frame,Box1.left,Box1.top,Box2.right,Box2.bottom);
    InsetRect(&frame,0,-4);
    SetPort(window);
    PenSize(2,2);
    FrameRoundRect(&frame,8,8);}
EndFunction

DoMouseDownEvent(){
static Rect  theBounds = {topBound,leftBound,bottomBound,rightBound};
WindowPtr   whichWindow;
CharsPtr    theTextPtr;
long        theLength;
void        remove_robotic_dialog();
    switch ( FindWindow( theEvent.where, &whichWindow ) ){
        case inDesk:
        case inMenuBar:
            DoMenuEvent(theEvent.where,whichWindow);
            HiliteMenu(0);
            break;
        case inSysWindow:
            SystemClick(&theEvent,whichWindow);
            break;
        case inContent:
            doContent(whichWindow,theEvent.where);
            break;
        case inDrag:
            DragWindow(whichWindow,theEvent.where,&theBounds);
            break;
        case inGrow:
            doGrow(whichWindow,theEvent.where);
            break;
        case inGoAway:
            if (TrackGoAway (whichWindow, theEvent.where))
            if(CartesianSpaceDlog == whichWindow)
                remove_robotic_dialog(&CartesianSpaceDlog);
            else if(JointSpaceDlog == whichWindow)
                remove_robotic_dialog(&JointSpaceDlog);
            else if(CartW.Ptr == whichWindow)
                removeWindow(&CartW);
            else if(JointW.Ptr == whichWindow)
                removeWindow(&JointW);
            else if(PathW.Ptr == whichWindow){
                TEDeactivate(PathW.Data.editRec);
                if(PathW.Data.dirty){
                    PathW.Data.dirty = 0;
                    PathW.Data.text = TEGetText(PathW.Data.editRec);
                    MoveHHI(PathW.Data.editRec);
                    HLock(PathW.Data.editRec);
                    theLength = (long) ((*PathW.Data.editRec).teLength);

```

```

        HUnlock(PathW.Data.editRec);
        MoveHHi(PathW.Data.text);
        HLock(PathW.Data.text);
        theTextPtr = *PathW.Data.text;
        SetFPos(PathW.Data.refNum,fsFromStart,0L);
        writeCode =
FSWrite(PathW.Data.refNum,&theLength,theTextPtr);
        if(writeCode == noErr);
        else
            fileError(writeCode,"pFile Write ");
        HUnlock(PathW.Data.text);}
        removeWindow(&PathW);
        TEDispose(PathW.Data.editRec);
        FlushVol(NIL,theReply.vRefNum);
        FSClose(PathW.Data.refNum);
        doPathEdit = 0;}

        break;
    default:
        break;}}
EndFunction

```

DoMenuEvent(where,theWindow)

```

WindowPtr theWindow;
Point where; {
int itemType;
Handle item;
Rect itemBox;
int menuID,menuItem;
long LongInt, theLength;
CharsPtr theTextPtr;
void create_robotic_dialog();
LongInt = MenuSelect(where);
menuID = HiWord(LongInt);
menuItem = LoWord(LongInt);
if(!menuID) return;
switch(menuID) {
    case FileID:
        switch(menuItem){
            case saveFile:
                if(theWindow == PathW.Ptr)
                    if(doPathEdit && PathW.Data.dirty){
                        PathW.Data.dirty = 0;
                        PathW.Data.text = TEGetText(PathW.Data.editRec);
                        theLength = (long) (**PathW.Data.editRec).teLength;
                        MoveHHi(PathW.Data.text);
                        HLock(PathW.Data.text);
                        theTextPtr = *PathW.Data.text;
                        SetFPos(PathW.Data.refNum,fsFromStart,0L);
                        writeCode =
FSWrite(PathW.Data.refNum,&theLength,theTextPtr);
                        if(writeCode == noErr);

```

```

        else
            fileError(writeCode,"pFile Write ");
            HUnlock(PathW.Data.text);}
        break;
    case pageSetup:
        break;
    case printFile:
        break;
    case quitFile:
        finish();
        break;}
    break;
case EditID:
    switch(menuItem) {
        case undoCommand:
            if(doPathEdit){
                TEDelete(PathW.Data.editRec);
                TEPaste(PathW.Data.editRec);
                AdjustScrollBar();
                PathW.Data.dirty = 1;}
            break;
        case cutCommand:
            if(doPathEdit){
                TECut(PathW.Data.editRec);
                AdjustScrollBar();
                PathW.Data.dirty = 1;}
            break;
        case copyCommand:
            if(doPathEdit)
                TECopy(PathW.Data.editRec);
            break;
        case pasteCommand:
            if(doPathEdit) {
                TEPaste(PathW.Data.editRec);
                AdjustScrollBar();
                PathW.Data.dirty = 1;}
            break;
        case clearCommand:
            if(doPathEdit) {
                TEDelete(PathW.Data.editRec);
                AdjustScrollBar();
                PathW.Data.dirty = 1;}
            break;
        default:
            break; }
    break;
case RoboticsID:
    switch(menuItem) {
        case cartesianSpace:
            create_robotic_dialog(robotCSpaceID,&CartesianSpaceDlog);
            if(!CartGO){
GetDItem(CartesianSpaceDlog,GO,&itemType,&item,&itemBox);
            HiliteControl(item,255);}/* disable Cartesian GO button */

```

```

        break;
    case jointSpace:
        create_robotic_dialog(robotJSpaceID,&JointSpaceDlog);
        if(!JointGO){
            GetDItem(JointSpaceDlog,GO,&itemType,&item,&itemBox);
            HiliteControl(item,255);/* disable Joint GO button */
            break;
        default:
            break;}
    break;
default:
    break; }}
EndFunction

```

```

DoUpdate(message)
long      message; {
GrafPtr   save_graf;
WindowPtr update_window,DialogWindow, theWindow;
update_window = (WindowPtr) message;
if(update_window == CartW.Ptr){
    GetPort(&save_graf);
    SetPort(update_window);
    BeginUpdate(update_window);
    DrawPicture(CartW.Pict,&CartW.Ptr->portRect);
    EndUpdate(update_window);
    SetPort(save_graf); }
else if(update_window == JointW.Ptr){
    GetPort(&save_graf);
    SetPort(update_window);
    BeginUpdate(update_window);
    DrawPicture(JointW.Pict,&JointW.Ptr->portRect);
    EndUpdate(update_window);
    SetPort(save_graf); }
else if(update_window == PathW.Ptr){
    GetPort(&save_graf);
    SetPort(update_window);
    BeginUpdate(update_window);
    EraseRect(&PathW.Ptr->portRect);
    TEUpdate(&PathW.Ptr->portRect,PathW.Data.editRec);
    DrawControls(PathW.Ptr);
    EndUpdate(update_window);
    SetPort(save_graf); }}
EndFunction

```

```

DoActivate(theMessage)
long      theMessage;{
CursHandle theCursor;
WindowPtr theWindow = (WindowPtr) theMessage;

```

```

        if(theWindow == PathW.Ptr)
            if(doPathEdit){
                theCursor = GetCursor(iBeamCursor);
                SetCursor(*theCursor);
                ShowCursor();}
        if((theWindow == CartW.Ptr) || (theWindow == JointW.Ptr)){
            SetCursor(&arrow);
            ShowCursor();}}

EndFunction

doContent(theWindow,Where)
WindowPtr    theWindow;
Point        Where; {
Boolean      extend;
    if(theWindow != FrontWindow()) {
        if(PathW.Ptr == FrontWindow()){
            TEDeactivate(PathW.Data.editRec);
            doPathEdit = 0;
            HideCursor();
            SetCursor(&arrow);
            SelectWindow(theWindow);
            ShowCursor();}
        else
            SelectWindow(theWindow);
        if(theWindow == PathW.Ptr) {
            TEActivate(PathW.Data.editRec);
            doPathEdit = 1;}}
    else{
        if(theWindow == PathW.Ptr ){
            GlobalToLocal(&Where);
            if(!(DoPathScroll(theWindow,Where))){
                if(doPathEdit == 1){
                    if((Where.v >= viewRect.top) && (Where.v <=
viewRect.bottom))
                        if((Where.h >= viewRect.left) && (Where.h <=
viewRect.right)){
                            extend = (Boolean) BitAnd((long)
theEvent.modifiers,(long) shiftKey);

                            TEClick(Where,extend,PathW.Data.editRec);}}}}}}
EndFunction

```

```

/* NOTE THIS FUNCTION IS NOT COMPLETE */
/* It has to handle grow icons left over when the window is increased
   in size. Perhaps this should be done in the DoUpdate function.
   Perhaps InvalRect and ValRect should be used */
doGrow(theWindow,startPt)
WindowPtr    theWindow;
Point        startPt;{
static Rect  sizeLimit = {100,100,500,500};
long         theSize;
    theSize = GrowWindow(theWindow,startPt,&sizeLimit);
    SizeWindow(theWindow,LoWord(theSize),HiWord(theSize),TRUE);}

```

EndFunction

/* NOTE THIS FUNCTION IS NOT COMPLETE */

finish()

```
{
    _closeall();
    exit(0);
}
```

WriteCartPath(fileName,data,numNodes)

```
int      numNodes;
char      *fileName;
struct Path  data; {
int      i,j,k, nextNode=0,lastNode = -14;
FILE      *fp;
    fp = fopen(fileName,"w");
    if (fp == NULL) ;
    fprintf(fp, "%s\t", data.numNodes);
    fprintf(fp, "\t\t\t\t%s\n", &CartComments[0][0]);
    for( i=1; i <= numNodes; ++i) {
        fprintf(fp, "\n");
        fprintf(fp, "%d\t", i);
        fprintf(fp, "\t\t\t\t%s\n", &CartComments[1][0]);
        nextNode += 15;
        lastNode += 15;
        for(j = lastNode,k=2; j <= nextNode; ++j, ++k) {
            fprintf(fp, "%s\t", &data.Node[j][0]);
            fprintf(fp, "\t\t\t\t%s\n", &CartComments[k][0]);}
    fclose(fp);}
```

EndFunction

getDataForPath(theDlog, itemNum, Data, j)

```
DialogPtr    theDlog;
int          itemNum, j;
struct Path *Data; {
Handle        item;
Rect          itemBox;
Str255        theText;
int          itemType;
    GetDItem(theDlog,itemNum,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strncpy(&Data->Node[j][0],theText,25); }
```

EndFunction

FillCartCommentsMat() {

```
    strcpy(&CartComments[0][0],"Number_of_Nodes");
    strcpy(&CartComments[1][0],"Node_number");
    strcpy(&CartComments[2][0],"x1");
    strcpy(&CartComments[3][0],"x2");
```

```

        strcpy(&CartComments[4][0],"x3");
        strcpy(&CartComments[5][0],"y1");
        strcpy(&CartComments[6][0],"y2");
        strcpy(&CartComments[7][0],"y3");
        strcpy(&CartComments[8][0],"z1");
        strcpy(&CartComments[9][0],"z2");
        strcpy(&CartComments[10][0],"z3");
        strcpy(&CartComments[11][0],"p1");
        strcpy(&CartComments[12][0],"p2");
        strcpy(&CartComments[13][0],"p3");
        strcpy(&CartComments[14][0],"move_time");
        strcpy(&CartComments[15][0],"acceleration_factor");
        strcpy(&CartComments[16][0],"LastNodeFlag");}

/*EndFunction */

OpenResultFiles() {
    theAngles = fopen("theAngles","w");
    if(theAngles == NULL){
        exit(0); }
    theCricket = fopen("theCricket","w");
    if(theCricket == NULL){
        exit(0); }
    thePos = fopen("thePos","w");
    if(thePos == NULL){
        exit(0); }}

EndFunction

CloseResultFiles() {
    fclose(theAngles);
    fclose(theCricket);
    fclose(thePos);}

EndFunction

CartPathOpen(name)
char    *name; {
FILE    *fp;
int     NumNodes,i,junk;
char    Junk[80];
float   temp;
fp = fopen(name,"r");
if(fp == 0){
    return(0);}
fscanf(fp,"%d",&NumNodes);
fprintf(thePath,"%d\\t\\t\\t\\t",NumNodes);
fscanf(fp,"%s",Junk);
fprintf(thePath,"%s\\n",Junk);
for(i = 1;i<=NumNodes;++i) {
    fscanf(fp,"%d",&junk);                /* Discard the # of the node */
    fprintf(thePath,"%d\\t\\t\\t\\t",junk);
    fscanf(fp,"%s",Junk);                /* Discard node number comment
*/

```



```

        fprintf(thePath,"%s\n",Junk);
orientation matrix */
        fscanf (fp,"%f",&CartesianPath[i].s1);          /* Write into buffer x1 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].s1);
        fscanf(fp,"%s",Junk);                             /* Discard x1 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].s2);          /* Write into buffer x2 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].s2);
        fscanf(fp,"%s",Junk);                             /* Discard x2 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].s3);          /* Write into buffer x3 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].s3);
        fscanf(fp,"%s",Junk);                             /* Discard x3 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].n1);          /* Write into buffer y1 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].n1);
        fscanf(fp,"%s",Junk);                             /* Discard y1 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].n2);          /* Write into buffer y2 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].n2);
        fscanf(fp,"%s",Junk);                             /* Discard y2 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].n3);          /* Write into buffer y3 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].n3);
        fscanf(fp,"%s",Junk);                             /* Discard y3 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].a1);          /* Write into buffer z1 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].a1);
        fscanf(fp,"%s",Junk);                             /* Discard z1 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].a2);          /* Write into buffer z2 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].a2);
        fscanf(fp,"%s",Junk);                             /* Discard z2 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].a3);          /* Write into buffer z3 of
orientation matrix */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].a3);
        fscanf(fp,"%s",Junk);                             /* Discard z3 comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].pos1);        /* Write into buffer the X
position */
        fprintf(thePath,"%f\t\t\t",CartesianPath[i].pos1);
        fscanf(fp,"%s",Junk);                             /* Discard X_component
comment */
        fprintf(thePath,"%s\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].pos2);        /* Write into buffer the Y
position */

```

```

        fprintf(thePath,"%f\\t\\t",CartesianPath[i].pos2);
        fscanf(fp,"%s",Junk);                /* Discard Y_component
comment */
        fprintf(thePath,"%s\\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].pos3);        /* Write into buffer the Z
position */
        fprintf(thePath,"%f\\t\\t",CartesianPath[i].pos3);
        fscanf(fp,"%s",Junk);                /* Discard Z_component
comment */
        fprintf(thePath,"%s\\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].move_time);    /* Write into buffer the move
time */
        fprintf(thePath,"%f\\t\\t",CartesianPath[i].move_time);
        fscanf(fp,"%s",Junk);                /* Discard move_time
comment */
        fprintf(thePath,"%s\\n",Junk);
        fscanf (fp,"%f",&CartesianPath[i].accFac);        /* Write into buffer acceleration
factor */
        fprintf(thePath,"%f\\t\\t",CartesianPath[i].accFac);
        fscanf(fp,"%s",Junk);                /* Discard
acceleration_factor comment */
        fprintf(thePath,"%s\\n",Junk);
        fscanf (fp,"%d",&CartesianPath[i].end);        /* Write into buffer if last
node */
        fprintf(thePath,"%d\\t\\t\\t\\t",CartesianPath[i].end);
        fscanf(fp,"%s",Junk);                /* Discard LastNodeFlag
comment */
        fprintf(thePath,"%s\\n",Junk);}
    fclose(fp);
    fclose(thePath);
    return(1);}
/*EndFunction*/

```

```

PascalStrCopy(ps1,ps2)
register    *ps1, *ps2; {
    register int length;
    length = *ps2++ = *ps1++;
    while (--length>=0) *ps2++ = *ps1++; }
EndFunction

```

```

FileError(message,fileName)
Str255 message, fileName; {
    ParamText(message, fileName, "\p", "\p");
    Alert( ErrorAlert, NIL); }
/* End of Function */

```

```

Query(theQuery)
Str255 theQuery; {
    ParamText(theQuery, "\p", "\p", "\p");
    Alert(QueryAlert, NIL); }

```

/* End of Function */

```
int  UnsaveQuery(fileName, diskName)
Str255  fileName, diskName; {
    ParamText(fileName, diskName, "\p", "\p");
    return(Alert(UnsaveID, NIL)); }
/* End of Function */
```

```
Adviser(theAdvice,inRefTo)
Str255  theAdvice, inRefTo; {
    ParamText(theAdvice, inRefTo, "\p", "\p");
    Alert(AdviceAlert, NIL); }
/* End of Function */
```

```
ErrorMessage(theKind,theJoint,theLimit)
int  theKind;
Str255  *theJoint,*theLimit;{
    switch (theKind){
        case joint_limit:
            ParamText(theJoint,theLimit,"\p","\p");
            Alert(JointLmtID, NIL);
            break;
        case velocity_limit:
            Alert(VelErrID, NIL);
            break;
        default:
            break;}}
```

EndFunction

```
fileError(rcode, theKind)
Str255  theKind;
OSErr  rcode; {
Str255  theMessage;
PascalStrCopy("\p", theMessage);
switch(rcode) {
    case dskFulErr:
        PascalStrCopy("\p Error: Disk full",theMessage);
        break;
    case fLckdErr:
        PascalStrCopy("\p Error: File locked",theMessage);
        break;
    case ioErr:
        PascalStrCopy("\p Error: I/O error",theMessage);
        break;
    case paramErr:
        PascalStrCopy("\p Error: Negative count",theMessage);
        break;
    case rfNumErr:
        PascalStrCopy("\p Error: Bad reference number",theMessage);
        break;
    case vLckdErr:
        PascalStrCopy("\p Error: Software volume lock",theMessage);
        break;
```

```

    case wPrErr:
        PascalStrCopy("\p Error: Hardware volume lock",theMessage);
        break;
    case wrPermErr:
        PascalStrCopy("\p Error: Read/write permission doesn't allow
writing",theMessage);
        break;
    case extFSErr:
        PascalStrCopy("\p Error: External file system",theMessage);
        break;
    case fnOpnErr:
        PascalStrCopy("\p Error: File not open",theMessage);
        break;
    case bdNamErr:
        PascalStrCopy("\p Error: Bad file name",theMessage);
        break;
    case fnfErr:
        PascalStrCopy("\p Error: File not found",theMessage);
        break;
    case nsvErr:
        PascalStrCopy("\p Error: No such volume",theMessage);
        break;
    case tmfoErr:
        PascalStrCopy("\p Error: Too many files open",theMessage);
        break;
    case dirFulErr:
        PascalStrCopy("\p Error: File directory full",theMessage);
        break;
    case dupFNErr:
        PascalStrCopy("\p Error: Duplicate file name and version",theMessage);
        break;
    default:
        PascalStrCopy("\p Error: System Error",theMessage);
        break; }
    Adviser(theKind,theMessage); }
/* End of Function */

```

```

CloseTheWindow(theWindow)
WindowPtr theWindow; {
    HideWindow(theWindow);
    CloseWindow(theWindow);
    if(NumOpnWdw>=0)
        --NumOpnWdw; }
EndFunction

VelocityLimit(){
int theKind = velocity_limit;
    ErrorMessage(theKind,"\p","\p");}
EndFunction

```

```

JointLimit (casenum,theta)
int *casenum;

```

```

JOINT_ANGLES  theta;{
char          joint[10];
char          theLimit[10];
int           LimitFlag = 0,theKind = joint_limit;
  if (theta.one > (.5235) ){
    LimitFlag = 1;
    stpcpy(joint,"1");
    CtoPstr(joint);
    stpcpy(theLimit,"120");
    CtoPstr(theLimit);}
  else if (theta.one < (-2.356) ) {
    LimitFlag = 1;
    stpcpy(joint,"1");
    CtoPstr(joint);
    stpcpy(theLimit,"-45");
    CtoPstr(theLimit);}
  if (theta.two > (3.1416)){
    LimitFlag = 1;
    stpcpy(joint,"2");
    CtoPstr(joint);
    stpcpy(theLimit,"180");
    CtoPstr(theLimit);}
  else if (theta.two < (-3.1416) ) {
    LimitFlag = 1;
    stpcpy(joint,"2");
    CtoPstr(joint);
    stpcpy(theLimit,"-180");
    CtoPstr(theLimit);}
  if (theta.three > (2.356)){
    LimitFlag = 1;
    stpcpy(joint,"3");
    CtoPstr(joint);
    stpcpy(theLimit,"45");
    CtoPstr(theLimit);}
  else if (theta.three < (-.5235) ) {
    LimitFlag = 1;
    stpcpy(joint,"3");
    CtoPstr(joint);
    stpcpy(theLimit,"-120");
    CtoPstr(theLimit);}
  if (theta.four > (3.1416)){
    LimitFlag = 1;
    stpcpy(joint,"4");
    CtoPstr(joint);
    stpcpy(theLimit,"180");
    CtoPstr(theLimit);}
  else if (theta.four < (-3.1416) ) {
    LimitFlag = 1;
    stpcpy(joint,"4");
    CtoPstr(joint);
    stpcpy(theLimit,"-180");
    CtoPstr(theLimit);}
  if (theta.five > (2.3562)){
    LimitFlag = 1;

```

```

        stpcpy(joint,"5");
        CtoPstr(joint);
        stpcpy(theLimit,"135");
        CtoPstr(theLimit);}
else if (theta.five < (-.5235) ) {
    LimitFlag = 1;
    stpcpy(joint,"5");
    CtoPstr(joint);
    stpcpy(theLimit,"-30");
    CtoPstr(theLimit);}
if (theta.six > (3.1415)){
    LimitFlag = 1;
    stpcpy(joint,"6");
    CtoPstr(joint);
    stpcpy(theLimit,"90");
    CtoPstr(theLimit);}
else if (theta.six < (0) ) {
    LimitFlag = 1;
    stpcpy(joint,"6");
    CtoPstr(joint);
    stpcpy(theLimit,"-90");
    CtoPstr(theLimit);}
if (theta.seven > (3.1416)){
    LimitFlag = 1;
    stpcpy(joint,"7");
    CtoPstr(joint);
    stpcpy(theLimit,"180");
    CtoPstr(theLimit);}
else if (theta.seven < (-3.1416) ) {
    LimitFlag = 1;
    stpcpy(joint,"7");
    CtoPstr(joint);
    stpcpy(theLimit,"-180");
    CtoPstr(theLimit);}
if (LimitFlag) {
    ErrorMessage(theKind,joint,theLimit);
    LTM->flags.robotics = 0; /* tell arm to stop */
    LTM->flags.jointLimit = 1;
    *casenum = 0;
    _closeall();}
EndFunction

```

```

/*
JointLimit2 (casenum,n,theta)
int          *casenum,*n;
JOINT_ANGLES theta;{
    IsJointWithinLimits("1",-45L,120L,theta.one,n,casenum);
    IsJointWithinLimits("2",-180L,180L,theta.two,n,casenum);
    IsJointWithinLimits("3",-45L,120L,theta.three,n,casenum);
    IsJointWithinLimits("4",-180L,180L,theta.four,n,casenum);
    IsJointWithinLimits("5",-45L,120L,theta.five,n,casenum);
    IsJointWithinLimits("6",-180L,180L,theta.six,n,casenum);
    IsJointWithinLimits("7",-45L,120L,theta.seven,n,casenum);

```

EndFunction

```
IsJointWithinLimits(theJoint,lowLimit,highLimit,theValue,n,casenum)
int *casenum,*n;
char *theJoint;
long lowLimit,highLimit;
float theValue; {
char joint[10];
Str255 theLimit;
int LimitFlag = 0,theKind = joint_limit;
theValue *= 180/PI;
if (theValue > highLimit){
    LimitFlag = 1;
    strcpy(joint,theJoint);
    CtoPstr(joint);
    NumToString(highLimit,theLimit);
else if (theValue < lowLimit ) {
    LimitFlag = 1;
    strcpy(joint,theJoint);
    CtoPstr(joint);
    NumToString(lowLimit,theLimit);
if (LimitFlag) {
    ErrorMessage(theKind,joint,theLimit);
    LTM->flags.robotics = 0;
    LTM->flags.jointLimit = 1;
    *n = 0;
    *casenum = 0;
    _closeall();}}

```

EndFunction

*/

```

/*****
COMMON FUNCTIONS OF MACINTOSH AND NONMACINTOSH INTERFACES.
*****/

```

```
InitPosition () {
    LTM->l.s.shld.data.pitch_pos = ThetaInit[1] ;
    LTM->l.s.shld.data.yaw_pos = ThetaInit[2] ;
    LTM->l.s.elbow.data.pitch_pos = ThetaInit[3] ;
    LTM->l.s.elbow.data.yaw_pos = ThetaInit[4] ;
    LTM->l.s.wrist.data.pitch_pos = ThetaInit[5] ;
    LTM->l.s.wrist.data.yaw_pos = ThetaInit[6] ;
    LTM->l.s.wrist.roll_data.pos = ThetaInit[7] ;}

```

EndFunction

```

/***** start of InitPath *****/
InitPath (){
if (LTM->flags.robotics){
    /* initializes the angle commands to the current angle values */
    THETA_1 = LTM->l.s.shld.data.pitch_pos;
    THETA_2 = LTM->l.s.shld.data.yaw_pos;
    THETA_3 = LTM->l.s.elbow.data.pitch_pos;

```

```

    THETA_4 = LTM->l.s.elbow.data.yaw_pos;
    THETA_5 = LTM->l.s.wrist.data.pitch_pos;
    THETA_6 = LTM->l.s.wrist.data.yaw_pos;
    THETA_7 = LTM->l.s.wrist.roll_data.pos;}}
/* end of initpath */

/***** info on LoadNextNode function *****/
The LoadNextNode function will load the next node in a predetermined path. By
node I mean a major position and orientation of the hand as opposed to the finite
positional steps between each major position/orientation.
*****/
LoadNextNode ()
{
static int    node = 1;

    if(LTM->flags.jointLimit) {
        node = 1;
        LTM->flags.jointLimit = 0;}
    LTM->l.s.hand.pos.i    = CartesianPath[node].pos1;
    LTM->l.s.hand.pos.j    = CartesianPath[node].pos2;
    LTM->l.s.hand.pos.k    = CartesianPath[node].pos3;
    LTM->l.s.hand.sliding.i = CartesianPath[node].s1;
    LTM->l.s.hand.sliding.j = CartesianPath[node].s2;
    LTM->l.s.hand.sliding.k = CartesianPath[node].s3;
    LTM->l.s.hand.normal.i  = CartesianPath[node].n1;
    LTM->l.s.hand.normal.j  = CartesianPath[node].n2;
    LTM->l.s.hand.normal.k  = CartesianPath[node].n3;
    LTM->l.s.hand.approach.i = CartesianPath[node].a1;
    LTM->l.s.hand.approach.j = CartesianPath[node].a2;
    LTM->l.s.hand.approach.k = CartesianPath[node].a3;
    LTM->l.s.hand.move_time  = CartesianPath[node].move_time;
    LTM->l.s.hand.accFactor  = CartesianPath[node].accFac;

    if (CartesianPath[node].end)
    {
        LTM->l.s.hand.LastNode = 1;
        node = 1;
    }
    else
    {
        LTM->l.s.hand.LastNode = 0;
        ++node;
    }

}

/***** end of LoadNextNode *****/

/*****/

void doRoboticCartesianDlog(theItem,whichDlog)
int      theItem;
DialogPtr whichDlog;{
int      itemHit;
static int      Home = 1;
int      DoModalLoop = 1,AlertOn = 1;
float     factor, angle;

```



```

long          count;
int           itemType;
Handle       item;
Rect         itemBox,picFrame;
DialogPtr    theDlog,theAlert;
DialogRecord theDRecord;
Str255       theText,kText, nodeText;
SFTYPEList   theType;
double       theNum;
long         numNodes,THEnum,node = 1;
COUNTER      itemNum,i, j = 0,k;
FLAG         sameTime=0,sameAcc=0;
static FLAG  setGraphics,Least_Norm;
static FLAG  setNumbers;
struct Path  PathData;
static char  theCPathName[20];
static char  SetupData[13][10];
static int   InitSetup = 1,InitPathName = 1;
char         pathBuff[8000],fileName[25];
long         posOff = 0;
char         *returnValue, tempString[25];
float        GetEditItemFloat();

switch(theItem){
    case GO:
        if (LTM->flags.simulate){
            if( SetGraphics){
                createWindow(CartesianWID,&CartW);
                picFrame = CartW.Ptr->portRect;
                CartW.Pict = OpenPicture(&picFrame);
                ShowPen();
                if (SetNumbers){
                    GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
                    HiliteControl(item,0); /* enable save run */
                    saveCartRun = 1;
                    OpenResultFiles();})
                LTM->flags.robotics = 1;
            InitPath ();
            LoadNextNode();
            if (LTM->flags.robotics){
                /*      LTM->flags.robotics_go = 1;  */
                for (;;){
                    GetDesiredPosition ();
                    if (LTM->flags.load_next_pos){
                        LoadNextNode();
                        drawcount = 0;
                        LTM->flags.load_next_pos = 0;}
                /*      LTM->flags.robotics_go = 1;  */
                if (LTM->flags.simulate && SetGraphics) {
                    if (! iteration) DrawArm (ViewArm0,view,when);
                    else DrawArm (ViewArm,view,when); }
                ++iteration;
                if (!LTM->flags.robotics) {
                    _closeall();
                    SysBeep(4);

```

```

        break;}}
    if (LTM->flags.simulate){
        iteration = 0;
        if(ShowTitles)
            Titles (view);
        if(ShowHome && view_case==3)
            PictArm ();
        if (SetGraphics) {
            HidePen();
            ClosePicture();}}}}
else {
    /* AN EVENT MUST BE CREATED FOR actually moving the
       manipulator. The following must be used
       InitPath ();
       LoadNextNode();
       GetDesiredPosition ();
       LTM->flags.robotics_go = 1;
       LTM->flags.load_next_pos = 0;    */
    break;
}
case STOP:
break;
case setUpRun:
    theDlog = GetNewDialog(SetupCartID,&theDRecord, -1L);
    frameItem(theDlog,OK,12);
    if(InitSetup){
        InitSetup = 0;}
    else {
        for(i = perfCrit;i<=startAngle7;++i){
            GetDItem(theDlog,i,&itemType,&item,&itemBox);
            SetIText(item,CtoPstr(SetupData[i-2]));}
        GetDItem(theDlog,Graphics,&itemType,&item,&itemBox);
        if(SetGraphics){
            SetCtlValue(item,1);
            setGraphics = 1; }
        else {
            SetCtlValue(item,0);
            setGraphics = 0;}
        GetDItem(theDlog,Numbers,&itemType,&item,&itemBox);
        if(SetNumbers) {
            SetCtlValue(item,1);
            setNumbers = 1;}
        else {
            SetCtlValue(item,0);
            setNumbers = 0;}
        GetDItem(theDlog,leastNorm,&itemType,&item,&itemBox);
        if(leastnorm) {
            SetCtlValue(item,1);
            Least_Norm = 1;}
        else {
            SetCtlValue(item,0);
            Least_Norm = 0;}
        GetDItem(theDlog,Optimized,&itemType,&item,&itemBox);
        if(!leastnorm) SetCtlValue(item,1);

```

```

else
    SetCtlValue(item,0);
if(Home) {
    GetDItem(theDlog,HomePos,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    GetDItem(theDlog,SpecifyDegrees,&itemType,&item,&itemBox);
    SetCtlValue(item,0);}
else {
    GetDItem(theDlog,HomePos,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,SpecifyDegrees,&itemType,&item,&itemBox);
    SetCtlValue(item,1);}
while (DoModalLoop) {
    ModalDialog( FALSE,&itemHit);
    switch (itemHit) {
        case OK:
            GetDItem(theDlog,perfCrit,&itemType,&item,&itemBox);
            GetIText(item,theText);
            PtoCstr((char *)theText);
            strcpy(SetupData[1],theText);
            LTM->l.s.hand.perf_crit = atoi((char *)theText);
            thePerfCrit = fopen("thePerfCrit","w");
            fprintf (thePerfCrit,"%d ",LTM->l.s.hand.perf_crit);
            fclose(thePerfCrit);
            GetDItem(theDlog,deltaT,&itemType,&item,&itemBox);
            GetIText(item,theText);
            PtoCstr((char *)theText);
            strcpy(SetupData[2],theText);
            delta_t = (float) atof(theText);
            GetdTMax();
            GetDItem(theDlog,Param3,&itemType,&item,&itemBox);
            GetIText(item,theText);
            PtoCstr((char *)theText);
            strcpy(SetupData[3],theText);
            GetDItem(theDlog,Param4,&itemType,&item,&itemBox);
            GetIText(item,theText);
            PtoCstr((char *)theText);
            strcpy(SetupData[4],theText);
            GetDItem(theDlog,When,&itemType,&item,&itemBox);
            GetIText(item,theText);
            PtoCstr((char *)theText);
            strcpy(SetupData[5],theText);
            when = atoi(theText);
            if(setGraphics) {
                SetGraphics = 1;
                if(LTM->flags.simulate){
                    GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
                    HiliteControl(item,0);}
                else {
                    SetGraphics = 0;
                    GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
                    HiliteControl(item,255);}
                if(setNumbers)

```

```

        SetNumbers = 1;
    else
        SetNumbers = 0;
    if(Least_Norm)
        leastnorm = 1;
    else
        leastnorm = 0;
    if(Home){
        for(i=1;i<=7;++i) ThetaInit[i] = HomeConfig[i];
        for(i=6;i<=12;++i) strcpy(SetupData[i],"0");
    }
    else {
        factor = PI/180;
        for( itemNum=startAngle1; itemNum<=startAngle7;
++itemNum) {

        GetDItem(theDlog,itemNum,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(SetupData[itemNum-2],theText);
        ThetaInit[itemNum-7] = HomeConfig[itemNum-7] +
factor*(atof(theText)); }}

    InitPosition();
    CsetupCond = 1;
    saveCartRun = 0;
    GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
    HiliteControl(item,255);/* disable save run */
    CloseDialog(theDlog);
    DoModalLoop = 0;
    break;
case cancel:
    GetDItem(theDlog,perfCrit,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[1],theText);
    GetDItem(theDlog,deltaT,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[2],theText);
    GetDItem(theDlog,Param3,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[3],theText);
    GetDItem(theDlog,Param4,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[4],theText);
    GetDItem(theDlog,When,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[5],theText);
    if(Home){
        for(k=6;k<=12;++k)
            strcpy(SetupData[k],"0");
    }
    else {

```

```

        factor = PI/180;
        for( itemNum=startAngle1; itemNum<=startAngle7;
++itemNum) {

    GetDlgItem(theDlog,itemNum,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(SetupData[itemNum-2],theText);}}
    CloseDialog(theDlog);
    DoModalLoop = 0;
    break;
case Graphics:
    GetDlgItem(theDlog,Graphics,&itemType,&item,&itemBox);
    if(!setGraphics) {
        SetCtlValue(item,1);
        setGraphics = 1;}
    else {
        SetCtlValue(item,0);
        setGraphics = 0;}
    break;
case Numbers:
    GetDlgItem(theDlog,Numbers,&itemType,&item,&itemBox);
    if(!setNumbers) {
        SetCtlValue(item,1);
        setNumbers = 1;
    }
    else {
        SetCtlValue(item,0);
        setNumbers = 0;}
    break;
case leastNorm:
    GetDlgItem(theDlog,leastNorm,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    GetDlgItem(theDlog,Optimized,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    Least_Norm = 1;
    break;
case Optimized:
    GetDlgItem(theDlog,leastNorm,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDlgItem(theDlog,Optimized,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    Least_Norm = 0;
    break;
case HomePos:
    GetDlgItem(theDlog,HomePos,&itemType,&item,&itemBox);
    SetCtlValue(item,1);

    GetDlgItem(theDlog,SpecifyDegrees,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        Home = 1;
        break;
case SpecifyDegrees:

```

```

        GetDItem(theDlog,SpecifyDegrees,&itemType,&item,&itemBox);
        SetCtlValue(item,1);
        GetDItem(theDlog,HomePos,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        Home = 0;
        break;
    default:
        break; })
    break;
case saveRun:
    theDlog = GetNewDialog(choosePathID,&theDRecord, -1L);
    strcpy(tempString,"Name the Run"); /* insert tempString into modal
dialog */

    CtoPstr(tempString);
    GetDItem(theDlog,4,&itemType,&item,&itemBox);
    SetIText(item,&tempString);
    frameItem(theDlog,OK,16);
    if(InitPathName) InitPathName = 0;
    else {
        GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
        SetIText(item,CtoPstr(theCPathName)); /* theCPathName is the
previously entered path name */
        while (DoModalLoop) {
            ModalDialog( FALSE,&itemHit);
            switch (itemHit) {
                case OK:
                    GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
                    GetIText(item,theText);
                    PtoCstr((char *)theText);
                    strcpy(theCPathName,theText);
                    if(!strlen(theText)) {
                        NoteAlert(noRunNmID,FALSE);
                        break; }
                    SaveRun(theText,whichDlog);
                    CloseDialog(theDlog);
                    DoModalLoop = 0;
                    saveCartRun = 0;
                    GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
                    HiliteControl(item,255);/* disable save run */
                    break;
                case cancel:
                    CloseDialog(theDlog);
                    DoModalLoop = 0;
                    break;
                default:
                    break; })
            break;
case createPath:
    theDlog = GetNewDialog(createCPathID,&theDRecord, -1L);
    frameItem(theDlog,OK,16);
    while (DoModalLoop) {
        ModalDialog( FALSE,&itemHit);

```

```

switch (itemHit) {
    case OK:
        GetDItem(theDlog,pathName,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        if(!strlen(theText)) {
            NoteAlert(noPathNmID,FALSE);
            break; }
        WriteCartPath(theText, PathData, (int) numNodes);
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
    case cancel:
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
    case AlwaysSameTime:
        GetDItem(theDlog,AlwaysSameTime,&itemType,&item,&itemBox);
        if(!sameTime) {
            SetCtlValue(item,1);
            sameTime = 1; }
        else {
            SetCtlValue(item,0);
            sameTime = 0; }
        break;
    case AlwaysSameAcc:
        GetDItem(theDlog,AlwaysSameAcc,&itemType,&item,&itemBox);
        if(!sameAcc) {
            SetCtlValue(item,1);
            sameAcc = 1; }
        else {
            SetCtlValue(item,0);
            sameAcc = 0; }
        break;
    case nodeOK:
        if(node == 1) {
            GetDItem(theDlog,HowManyNodes,&itemType,&item,&itemBox);
            GetIText(item,theText);
            StringToNum(theText,&numNodes) ;
            PtoCstr((char *)theText);
            for( i=0; theText[i]!='\0'; ++i)
                PathData.numNodes[i] = theText[i];
            PathData.numNodes[i] = '\0'; }
            for( itemNum=X1; itemNum<=P3; ++itemNum) {
                ++j;
                getDataForPath(theDlog, itemNum, &PathData,j) ; }
            if (sameTime) {
                ++j;
                getDataForPath(theDlog, theAlwaysTime, &PathData,j) ;
            }
        }
    else {

```

```

        ++j;
        getDataForPath(theDlog, theMoveTime,
&PathData,j) ;}

        if (sameAcc) {
            ++j;
            getDataForPath(theDlog, theAlwysAcc,
&PathData,j) ; }

        else {
            ++j;
            getDataForPath(theDlog, theAcc, &PathData,j) ; }
        if (node == numNodes) {
            ++j;
            strcpy(&PathData.Node[j][0],"1");
            break;}
        else {
            ++j;
            strcpy(&PathData.Node[j][0],"0");
            ++node;
            NumToString(node, nodeText);

GetDItem(theDlog,theCNode,&itemType,&item,&itemBox);
            SetIText(item,nodeText);
            break; }
        case    nodeCancel:
            for( itemNum=X1; itemNum<=theAcc; ++itemNum) {

GetDItem(theDlog,itemNum,&itemType,&item,&itemBox);
                SetIText(item,""); }
            break;
        default:
            break; }}
        break;
    case alterPath:
        count = 1;
        theType[0] = "TEXT";
        SFGGetFile(getWhere, "\p", 0L, 1, theType, 0L, &theReply);
        if(theReply.good) {
            PathW.ID = PathWID;
            createWindow(PathW.ID,&PathW);
            SetWTitle(PathW.Ptr,theReply.fName);
            AllocateTEHandle(&PathW);
            TEActivate(PathW.Data.editRec);
            openCode =
FSOpen(theReply.fName,theReply.vRefNum,&(PathW.Data.refNum));
            if(openCode == noErr) {
                SetFPos(PathW.Data.refNum,fsFromStart,posOff);
                FSRead(PathW.Data.refNum,&count,&pathBuff[0]);
                Nodes = (pathBuff[0] - '0');
                count = Nodes*300;
                FSRead(PathW.Data.refNum,&count,&pathBuff[1]);}
            TSEsetText(pathBuff,count,PathW.Data.editRec);
            for(i=0;i<=((int) count);++i)
                pathBuff[i] = ";
            doPathEdit = 1;

```



```

        AdjustScrollBar();
        if(!strcmp(PtoCstr((char *)theReply.fName),theCPathName)){

GetDItem(CartesianSpaceDlog,GO,&itemType,&item,&itemBox);
        HiliteControl(item,255);
        CartGO = 0;

GetDItem(CartesianSpaceDlog,choosePath,&itemType,&item,&itemBox);
        HiliteControl(item,10);
        theCPathChanged = 1;}}/* disable Cartesian GO button */
    else
        break;
    break;
case choosePath:
    theDlog = GetNewDialog(choosePathID,&theDRecord, -1L);
    frameItem(theDlog,OK,16);
    if(InitPathName) InitPathName = 0;
    else {
        GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
        SetIText(item,CtoPstr(theCPathName));
        while (DoModalLoop) {
            ModalDialog( FALSE,&itemHit);
            switch (itemHit) {
                case OK:
                    GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
                    GetIText(item,theText);
                    PtoCstr((char *)theText);
                    strcpy(theCPathName,theText);
                    if(!strlen(theText)) {
                        NoteAlert(noPathNmID,FALSE);
                        break; }
                    else
                        thePath = fopen("thePath","w");
                    if(CartPathOpen(theText))
                        CpathCond = 1;
                    if(theCPathChanged){
                        theCPathChanged = 0;

GetDItem(CartesianSpaceDlog,GO,&itemType,&item,&itemBox);
                        HiliteControl(item,0);

GetDItem(CartesianSpaceDlog,choosePath,&itemType,&item,&itemBox);
                        HiliteControl(item,0);}
                    CloseDialog(theDlog);
                    DoModalLoop = 0;
                    break;
                case cancel:
                    CloseDialog(theDlog);
                    DoModalLoop = 0;
                    break;
                default:
                    break; }}
        break;
case realMode:

```

```

        GetDItem(whichDlog,simulationMode,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        GetDItem(whichDlog,realMode,&itemType,&item,&itemBox);
        SetCtlValue(item,1);
        LTM->flags.simulate = 0;
        GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
        HiliteControl(item,255); /* disable graphics options */
        break;
    case simulationMode:
        GetDItem(whichDlog,simulationMode,&itemType,&item,&itemBox);
        SetCtlValue(item,1);
        GetDItem(whichDlog,realMode,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        LTM->flags.simulate = 1;
        if(SetGraphics){
            GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
            HiliteControl(item,0);/* enable graphics options */
            break;
        case graphicsOptions:
            doGraphicsOption();
            break;
        default:
            break;}}

```

EndFunction

```

void doRoboticJointDlog(theItem,whichDlog)
int theItem;
DialogPtr whichDlog;{
int itemHit;
static int Home = 1;
int DoModalLoop = 1,AlertOn = 1;
float factor, angle;
long count;
int itemType;
Handle item;
Rect itemBox,picFrame;
DialogPtr theDlog,theAlert;
DialogRecord theDRecord;
Str255 theText,kText, nodeText;
SFTYPEList theType;
double theNum;
long numNodes,THEnum,node = 1;
COUNTER itemNum,i, j = 0;
FLAG sameTime=0,sameAcc=0;
static FLAG setGraphics,Least_Norm;
static FLAG setNumbers;
struct Path PathData;
static char SetupData[3][10];
static char theJPathName[20];
static int InitSetup=1,InitPathName=1;
char pathBuff[4000],fileName[25];
char *returnValue,tempString[25];
long posOff = 0;

```

```

switch(theItem){
case GO:
    if (LTM->flags.simulate && (SetGraphics || SetNumbers)) {
        GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
        HiliteControl(item,0); /* enable save run */
        saveJointRun = 1;}
    if(LTM->flags.simulate && SetGraphics){
        createWindow(JointWID,&JointW);
        picFrame = JointW.Ptr->portRect;
        JointW.Pict = OpenPicture(&picFrame);
        ShowPen();}
    InitPosition ();
    LTM->flags.robotics = 1;
    if (SetNumbers)
        OpenJointFiles();
    InitPath ();
    LoadNextJointNode();
    if (LTM->flags.robotics){
/*      LTM->flags.robotics_go = 1; */
        for (;;){
            GetJointConfig();
            if (LTM->flags.load_next_pos){
                LoadNextJointNode();
                drawcount = 0;
                LTM->flags.load_next_pos = 0;}
/*      LTM->flags.robotics_go = 1; */
            if (LTM->flags.simulate && SetGraphics) {
                if (!iteration) DrawArm (ViewArm0,view,when);
                else DrawArm (ViewArm,view,when); }
            ++iteration;
            if (!LTM->flags.robotics) {
                _closeall();
                SysBeep(4);
                if (LTM->flags.simulate && SetGraphics) {
                    HidePen();}
                break;}}
            CloseJointFiles();
        if (LTM->flags.simulate){
            iteration = 0;
            if(ShowTitles)
                Titles (view);
            if(ShowHome && view_case==3)
                PictArm ();
            ClosePicture();}}
    break;
case STOP:
    break;
case setUpRun:
    theDlog = GetNewDialog(SetupJointID,&theDRecord, -1L);
    frameItem(theDlog,OK,12);
    if(InitSetup){
        InitSetup = 0;}
    else {
        for(i = PerCentMaxSpd;i<=(When-2);++i){

```

```

        GetDItem(theDlog,i,&itemType,&item,&itemBox);
        SetIText(item,CtoPstr(SetupData[i-3]));}
GetDItem(theDlog,(Graphics-9),&itemType,&item,&itemBox);
if(SetGraphics){
    SetCtlValue(item,1);
    setGraphics = 1; }
else {
    SetCtlValue(item,0);
    setGraphics = 0;}
GetDItem(theDlog,(Numbers-9),&itemType,&item,&itemBox);
if(SetNumbers) {
    SetCtlValue(item,1);
    setNumbers = 1;}
else {
    SetCtlValue(item,0);
    setNumbers = 0;}
while (DoModalLoop) {
ModalDialog( FALSE,&itemHit);
switch (itemHit) {
    case    OK:

```

```

GetDItem(theDlog,PerCentMaxSpd,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[0],theText);
    MaxJointSpeeds((float) atof(theText));
    GetDItem(theDlog,deltaT,&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[1],theText);
    delta_t = (float) atof(theText);
    GetDItem(theDlog,(When-2),&itemType,&item,&itemBox);
    GetIText(item,theText);
    PtoCstr((char *)theText);
    strcpy(SetupData[2],theText);
    when = atoi(theText);
    if(setGraphics) {
        SetGraphics = 1;
        if(LTM->flags.simulate){

```

```

GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
        HiliteControl(item,0);})/* enable graphics options

```

```

*/

```

```

    else {
        SetGraphics = 0;

```

```

GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
        HiliteControl(item,255);})/* disable graphics options */
if(setNumbers)
    SetNumbers = 1;
else
    SetNumbers = 0;
InitPosition();
JsetupCond = 1;

```

```

        saveJointRun = 0;
        GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
        HiliteControl(item,255);/* disable save run */
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
    case cancel:

        GetDItem(theDlog,PerCentMaxSpd,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(SetupData[0],theText);
        GetDItem(theDlog,deltaT,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(SetupData[1],theText);
        GetDItem(theDlog,(When-2),&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(SetupData[2],theText);
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
    case (Graphics-9):
        GetDItem(theDlog,(Graphics-
9),&itemType,&item,&itemBox);
        if(!setGraphics) {
            SetCtlValue(item,1);
            setGraphics = 1;}
        else {
            SetCtlValue(item,0);
            setGraphics = 0; }
        break;
    case (Numbers-9):
        GetDItem(theDlog,(Numbers-
9),&itemType,&item,&itemBox);
        if(!setNumbers) {
            SetCtlValue(item,1);
            setNumbers = 1; }
        else {
            SetCtlValue(item,0);
            setNumbers = 0; }
        break;
    default:
        break; }}
    break;
case saveRun:
    theDlog = GetNewDialog(choosePathID,&theDRecord, -1L);
    strcpy(tempString,"Name the Run");
    CtoPstr(tempString);
    GetDItem(theDlog,4,&itemType,&item,&itemBox);
    SetIText(item,&tempString);
    frameItem(theDlog,OK,16);
    if(InitPathName) InitPathName = 0;

```

```

else {
    GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
    SetIText(item,CtoPstr(theJPathName));}
while (DoModalLoop) {
ModalDialog( FALSE,&itemHit);
switch (itemHit) {
    case OK:
        GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
        strcpy(theJPathName,theText);
        if(!strlen(theText)) {
            NoteAlert(noRunNmID,FALSE);
            break; }
        SaveRun(theText,whichDlog);
        saveJointRun = 0;
        CloseDialog(theDlog);
        DoModalLoop = 0;
        GetDItem(whichDlog,saveRun,&itemType,&item,&itemBox);
        HiliteControl(item,255);/* disable save run */
        break;
    case cancel:
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
    default:
        break; })
break;
case createPath:
    theDlog = GetNewDialog(createJPathID,&theDRecord, -1L);
    frameItem(theDlog,OK,16);
    while (DoModalLoop) {
        ModalDialog( FALSE,&itemHit);
        switch (itemHit) {
            case OK:
                GetDItem(theDlog,pathName,&itemType,&item,&itemBox);
                GetIText(item,theText);
                PtoCstr((char *)theText);
                if(!strlen(theText)) {
                    NoteAlert(noPathNmID,FALSE);
                    break; }
                WriteJointPath(theText, PathData, (int) numNodes);
                CloseDialog(theDlog);
                DoModalLoop = 0;
                break;
            case cancel:
                CloseDialog(theDlog);
                DoModalLoop = 0;
                break;
            case jointnodeOK:
                if(node == 1) {
                    GetDItem(theDlog,HowManyNodes,&itemType,&item,&itemBox);
                    GetIText(item,theText);

```

```

        StringToNum(theText,&numNodes) ;
        PtoCstr((char *)theText);
        for( i=0; theText[i]!='\0'; ++i)
            PathData.numNodes[i] = theText[i];
        PathData.numNodes[i] = '\0'; }
    for( itemNum=jointAngle1; itemNum<=jointAngle7;
++itemNum) {

        ++j;
        getDataForPath(theDlog, itemNum, &PathData,j) ; }
    ++j;
    getDataForPath(theDlog, jointAccFac, &PathData,j) ;
    if (node == numNodes) {
        ++j;
        strcpy(&PathData.Node[j][0],"1");
        break;}
    else {
        ++j;
        strcpy(&PathData.Node[j][0],"0");
        ++node;
        NumToString(node, nodeText);

        GetDItem(theDlog,CurrentNode,&itemType,&item,&itemBox);
        SetIText(item,nodeText);
        break; }

    default:
        break; })
    break;
case alterPath:
    count = 1;
    theType[0] = "TEXT";
    SFGetFile(getWhere, "\p", 0L, 1, theType, 0L, &theReply);
    if(theReply.good) {
        PathW.ID = PathWID;
        createWindow(PathW.ID,&PathW);
        SetWTitle(PathW.Ptr,theReply.fName);
        AllocateTEHandle(&PathW);
        TEActivate(PathW.Data.editRec);
        openCode =
FSOpen(theReply.fName,theReply.vRefNum,&(PathW.Data.refNum));
        if(openCode == noErr) {
            SetFPos(PathW.Data.refNum,fsFromStart,posOff);
            FSRead(PathW.Data.refNum,&count,&pathBuff[0]);
            count = (pathBuff[0] - '0')*265;
            FSRead(PathW.Data.refNum,&count,&pathBuff[1]);}
        TSEsetText(pathBuff,count,PathW.Data.editRec);
        for(i=0;i<=((int) count);++i)
            pathBuff[i] = " ";
        doPathEdit = 1;
        AdjustScrollBar();
        if(!strcmp(PtoCstr((char *)theReply.fName),theJPathName)){
            GetDItem(JointSpaceDlog,GO,&itemType,&item,&itemBox);
            HiliteControl(item,255);
            JointGO = 0;

```

```

GetDItem(JointSpaceDlog,choosePath,&itemType,&item,&itemBox);
        HiliteControl(item,10); /* disable Joint GO button */
        theJPathChanged = 1;}}

    else
        break;
break;
case choosePath:
    theDlog = GetNewDialog(choosePathID,&theDRecord, -1L);
    frameItem(theDlog,OK,16);
    if(InitPathName) InitPathName = 0;
    else {
        GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
        SetIText(item,CtoPstr(theJPathName));}
    while (DoModalLoop) {
        ModalDialog( FALSE,&itemHit);
        switch (itemHit) {
            case OK:
                GetDItem(theDlog,PathName,&itemType,&item,&itemBox);
                GetIText(item,theText);
                PtoCstr((char *)theText);
                strcpy(theJPathName,theText);
                if(!strlen(theText)) {
                    NoteAlert(noPathNmID,FALSE);
                    break; }
                if(JointPathOpen(theText))
                    JpathCond = 1;
                if(theJPathChanged){
                    theJPathChanged = 0;

GetDItem(JointSpaceDlog,GO,&itemType,&item,&itemBox);
        HiliteControl(item,0);

GetDItem(JointSpaceDlog,choosePath,&itemType,&item,&itemBox);
        HiliteControl(item,0);}
        CloseDialog(theDlog);
        DoModalLoop = 0;
        break;
        case cancel:
            CloseDialog(theDlog);
            DoModalLoop = 0;
            break;
        default:
            break; }}
break;
case realMode:
    GetDItem(whichDlog,simulationMode,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(whichDlog,realMode,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    LTM->flags.simulate = 0;
    GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
    HiliteControl(item,255);/* disable graphics options */
break;

```



```

        case simulationMode:
            GetDItem(whichDlog,simulationMode,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
            GetDItem(whichDlog,realMode,&itemType,&item,&itemBox);
            SetCtlValue(item,0);
            LTM->flags.simulate = 1;
            if(SetGraphics){

GetDItem(whichDlog,graphicsOptions,&itemType,&item,&itemBox);
        HiliteControl(item,0);/* enable graphics options */
        break;
        case graphicsOptions:
            doGraphicsOption();
            break;
        default:
            break;}}
EndFunction

```

```

FillJointCommentsMat() {
    strcpy(&JointComments[0][0],"Number_of_Nodes");
    strcpy(&JointComments[1][0],"Node_number");
    strcpy(&JointComments[2][0],"Theta1");
    strcpy(&JointComments[3][0],"Theta2");
    strcpy(&JointComments[4][0],"Theta3");
    strcpy(&JointComments[5][0],"Theta4");
    strcpy(&JointComments[6][0],"Theta5");
    strcpy(&JointComments[7][0],"Theta6");
    strcpy(&JointComments[8][0],"Theta7");
    strcpy(&JointComments[9][0],"acceleration_factor");
    strcpy(&JointComments[10][0],"LastNodeFlag");}
EndFunction

```

```

WriteJointPath(fileName,data,numNodes)
    int        numNodes;
    char        *fileName;
    struct Path  data; {
    int        i,j,k, nextNode=0,lastNode = -8;
    FILE        *fp;
        fp = fopen(fileName,"w");
        if (fp == NULL) ;
        fprintf(fp, "%s\t", data.numNodes);
        fprintf(fp, "\t\t\t\t%s\n", &JointComments[0][0]);
        for( i=1; i <= numNodes; ++i) {
            fprintf(fp, "\n");
            fprintf(fp, "%d\t", i);
            fprintf(fp, "\t\t\t\t%s\n", &JointComments[1][0]);
            nextNode += 9;
            lastNode += 9;
            for(j = lastNode,k=2; j <= nextNode; ++j, ++k) {
                fprintf(fp, "%s\t", &data.Node[j][0]);
                fprintf(fp, "\t\t\t\t%s\n", &JointComments[k][0]);}}
        fclose(fp);}

```

EndFunction

```
void create_robotic_dialog(dlgID,dlgPtr)
int      dlgID;
DialogPtr *dlgPtr; {
Rect      r;
int      i,j;
    if(*dlgPtr == 0){
        *dlgPtr = GetNewDialog(dlgID,0L,-1L);
        doHilite(*dlgPtr);
/*      if(positioned_flag == FALSE){
            r = *dlgPtr->portRect;
            dialog_position.h = ((DesktopRect.right-DesktopRect.left)-
                                (r.right-r.left) - DesktopRect.left);
            dialog_position.v = TOP_OF_SCREEN + 20;}
            MoveWindow(*dlgPtr,dialog_position.h,dialog_position.v,FALSE);
            ShowWindow(*dlgPtr); */)
    else {
        SelectWindow(*dlgPtr);
        doHilite(*dlgPtr);}}
```

EndFunction

```
void remove_robotic_dialog(dlgPtr)
DialogPtr *dlgPtr;{
GrafPtr    aPort;
    GetPort(&aPort);
    SetPort(*dlgPtr);
    dialog_flags = 0;
    dialog_position.v = dialog_position.h = 0;
    LocalToGlobal(&dialog_position);
    DisposDialog(*dlgPtr);
    *dlgPtr = 0;
    SetPort(aPort);}
```

EndFunction

```
JointPathOpen(name)
char      *name; {
FILE      *fp;
int      NumNodes,i,junk;
char      Junk[80];
float     temp;
    fp = fopen(name,"r");
    if(fp == 0)
        return(0);
    fscanf(fp,"%d",&NumNodes);
    fprintf(theJointPath,"%d\\t\\t\\t\\t",NumNodes);
    fscanf(fp,"%s",Junk);
    fprintf(theJointPath,"%s\\n",Junk);
    for(i = 1;i<=NumNodes;++i) {
        fscanf(fp,"%d",&junk);
```

/* Discard the # of the node */

```

        fprintf(theJointPath,"%d\n",junk);
        fscanf(fp,"%s",Junk);          /* Discard "Data for NODE" comment */
        fprintf(theJointPath,"%s",Junk);
        fscanf (fp,"%f",&JointPath[i].J1);          /* Write into buffer Theta 1 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J1);
        fscanf(fp,"%s",Junk);          /* Discard Theta 1 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J2);          /* Write into buffer Theta 2 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J2);
        fscanf(fp,"%s",Junk);          /* Discard Theta 2 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J3);          /* Write into buffer Theta 3 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J3);
        fscanf(fp,"%s",Junk);          /* Discard Theta 3 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J4);          /* Write into buffer Theta 4 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J4);
        fscanf(fp,"%s",Junk);          /* Discard Theta 4 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J5);          /* Write into buffer Theta 5 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J5);
        fscanf(fp,"%s",Junk);          /* Discard Theta 5 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J6);          /* Write into buffer Theta 6 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J6);
        fscanf(fp,"%s",Junk);          /* Discard Theta 6 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].J7);          /* Write into buffer Theta 7 */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].J7);
        fscanf(fp,"%s",Junk);          /* Discard Theta 7 comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%f",&JointPath[i].accFac); /* Write into buffer acceleration factor */
        fprintf(theJointPath,"%f\t\t\t",JointPath[i].accFac);
        fscanf(fp,"%s",Junk);          /* Discard acceleration_factor comment */
        fprintf(theJointPath,"%s\n",Junk);
        fscanf (fp,"%d",&JointPath[i].end);          /* Write into buffer if end of path */
        fprintf(theJointPath,"%d\t\t\t\t\t",JointPath[i].end);
        fscanf(fp,"%s",Junk);          /* Discard LastNodeFlag comment */
        fprintf(theJointPath,"%s\n",Junk);}

fclose(fp);
fclose(theJointPath);
return(1);}

EndFunction

```

```

GetdTMax() {
    dTMax.one = (MaxSpd12/LGearRatio)*delta_t;
    dTMax.two = (MaxSpd12/LGearRatio)*delta_t;
    dTMax.three = (MaxSpd12/LGearRatio)*delta_t;
    dTMax.four = (MaxSpd12/LGearRatio)*delta_t;
    dTMax.five = (MaxSpd12/SGearRatio)*delta_t;
    dTMax.six = (MaxSpd12/SGearRatio)*delta_t;
    dTMax.seven = (MaxSpdRoll/RGearRatio)*delta_t; }

```

EndFunction

```
int SaveRun(RunName,theDlog)
char      RunName[25];
DialogPtr theDlog;{
char      flag;
char      tempString[25];
FILE      *fp;
int       itemHit;
int       DoModalLoop = 1;
int       itemType;
Handle    item;
Rect      itemBox;
DialogPtr theAlert;
DialogRecord theDRecord;
Str255    theText;
    strcpy(tempString,RunName);
    if(theDlog == CartesianSpaceDlog){
        strcat(tempString,".Path");
        fp = fopen(tempString,"r");
        if (fp) {
            theAlert = GetNewDialog(nameInUseID,&theDRecord, -1L);
            frameItem(theAlert,cancel,16);
            while (DoModalLoop) {
                ModalDialog( FALSE,&itemHit);
                switch (itemHit) {
                    case contAnyway:
                        CloseDialog(theAlert);
                        DoModalLoop = 0;
                        remove(tempString);
                        strcpy(tempString,RunName);
                        strcat(tempString,".Angles");
                        remove(tempString);
                        strcpy(tempString,RunName);
                        strcat(tempString,".Cricket");
                        remove(tempString);
                        strcpy(tempString,RunName);
                        strcat(tempString,".PerfCrit");
                        remove(tempString);
                        strcpy(tempString,RunName);
                        strcat(tempString,".Pos");
                        remove(tempString);
                        break;
                    case cancel:
                        CloseDialog(theAlert);
                        DoModalLoop = 0;
                        return(0);}}
            strcpy(tempString,RunName);
            strcat(tempString,".Path");
            rename("thePath",tempString);
            strcpy(tempString,RunName);
            strcat(tempString,".Angles");
            rename("theAngles",tempString);
            strcpy(tempString,RunName);
```

```

    strcat(tempString, ".Cricket");
    rename("theCricket", tempString);
    strcpy(tempString, RunName);
    strcat(tempString, ".PerfCrit");
    rename("thePerfCrit", tempString);
    strcpy(tempString, RunName);
    strcat(tempString, ".Pos");
    rename("thePos", tempString);
    return(1);}
else if(theDlog == JointSpaceDlog){
    strcat(tempString, ".JPath");
    fp = fopen(tempString, "r");
    if (fp) {
        theAlert = GetNewDialog(nameInUseID, &theDRecord, -1L);
        frameItem(theAlert, cancel, 16);
        while (DoModalLoop) {
            ModalDialog( FALSE, &itemHit);
            switch (itemHit) {
                case contAnyway:
                    CloseDialog(theAlert);
                    DoModalLoop = 0;
                    remove(tempString);
                    strcpy(tempString, RunName);
                    strcat(tempString, ".JPAngles");
                    remove(tempString);
                    strcpy(tempString, RunName);
                    strcat(tempString, ".JPCricket");
                    remove(tempString);
                    strcpy(tempString, RunName);
                    strcat(tempString, ".JPPos");
                    remove(tempString);
                    break;
                case cancel:
                    CloseDialog(theAlert);
                    DoModalLoop = 0;
                    return(0);}}}
    strcpy(tempString, RunName);
    strcat(tempString, ".JPath");
    rename("theJointPath", tempString);
    strcpy(tempString, RunName);
    strcat(tempString, ".JPAngles");
    rename("theJointPAngles", tempString);
    strcpy(tempString, RunName);
    strcat(tempString, ".JPCricket");
    rename("theJointPCricket", tempString);
    strcpy(tempString, RunName);
    strcat(tempString, ".JPPos");
    rename("theJointPPos", tempString);
    return(1);}]

```

EndFunction

These have to do with moving in Joint Space.

*****/

```
MaxJointSpeeds(factor)
float factor; {
    factor /= 100;
    MaxSpeed.one = factor*(MaxSpd12/LGearRatio);
    MaxSpeed.two = factor*(MaxSpd12/LGearRatio);
    MaxSpeed.three = factor*(MaxSpd12/LGearRatio);
    MaxSpeed.four = factor*(MaxSpd12/LGearRatio);
    MaxSpeed.five = factor*(MaxSpd12/SGearRatio);
    MaxSpeed.six = factor*(MaxSpd12/SGearRatio);
    MaxSpeed.seven = factor*(MaxSpdRoll/RGearRatio); }
EndFunction
```

```
LoadNextJointNode(){
static int node = 1;
static float factor;
    if(node==1) factor = PI/180;
    if(LTM->flags.jointLimit) {
        node = 1;
        LTM->flags.jointLimit = 0;}
    LTM->l.s.joint.J1 = (JointPath[node].J1-90)*factor;
    LTM->l.s.joint.J2 = JointPath[node].J2*factor;
    LTM->l.s.joint.J3 = (JointPath[node].J3+90)*factor;
    LTM->l.s.joint.J4 = JointPath[node].J4*factor;
    LTM->l.s.joint.J5 = JointPath[node].J5*factor;
    LTM->l.s.joint.J6 = (JointPath[node].J6+90)*factor;
    LTM->l.s.joint.J7 = JointPath[node].J7*factor;
    LTM->l.s.hand.accFactor = JointPath[node].accFac;
    if (JointPath[node].end){
        LTM->l.s.hand.LastNode = 1;
        node = 1;}
    else{
        LTM->l.s.hand.LastNode = 0;
        ++node;}}
EndFunction
```

```
OpenJointFiles() {
    theJointPath = fopen("theJointPath","w");
    if(theJointPath == NULL){
        exit(0); }
    theJointPAngles = fopen("theJointPAngles","w");
    if(theJointPAngles == NULL){
        exit(0); }
    theJointPCricket = fopen("theJointPCricket","w");
    if(theJointPCricket == NULL){
        exit(0); }
    theJointPPos = fopen("theJointPPos","w");
    if(theJointPPos == NULL){
```

```

        exit(0); }}
EndFunction

```

```

CloseJointFiles() {
    fclose(theJointPath);
    fclose(theJointPAngles);
    fclose(theJointPCricket);
    fclose(theJointPPos);}
EndFunction

```

```

/*****
These functions have to do with the Simulation output window.
*****/

```

```

createWindow(theID,theWindow)
int         theID;
struct WINDOW *theWindow;{
    if(theWindow->Ptr == 0){
        theWindow->Ptr = GetNewWindow(theID,&theWindow->Record,-1L);
        ClipRect(&((theWindow->Ptr)->portRect));
        if(theWindow->Ptr == PathW.Ptr){
            PathW.Data.VScrollBar = GetNewControl(vScrollBar,theWindow-
>Ptr);

            PathW.Data.dirty = 0;}
        SetPort(theWindow->Ptr);}
    else {
        SelectWindow(theWindow->Ptr);
        ClipRect(&((theWindow->Ptr)->portRect)); /*
        SetPort(theWindow->Ptr);
        RectRgn((theWindow->Ptr)->visRgn,&(theWindow->Ptr)->portRect);
        EraseRect(&((theWindow->Ptr)->portRect));} }
EndFunction

```

```

removeWindow(theWindow)
struct WINDOW *theWindow;{
GrafPtr     aPort;
    GetPort(&aPort);
    SetPort(theWindow->Ptr);
    CloseWindow(theWindow->Ptr);
    theWindow->Ptr = 0;
    SetPort(aPort);}
EndFunction

```

```

int GetEditItemInt(theDlog,theEditItem)
DialogPtr    theDlog;
int          theEditItem;{
int          itemType;
Handle       item;
Rect         itemBox;
Str255       theText;

```

```

        GetDlgItem(theDlog,theEditItem,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
    return(atoi(theText));}

EndFunction

float GetEditItemFloat(theDlog,theEditItem)
DialogPtr    theDlog;
int          theEditItem;{
int          itemType;
Handle       item;
Rect         itemBox;
Str255       theText;
        GetDlgItem(theDlog,theEditItem,&itemType,&item,&itemBox);
        GetIText(item,theText);
        PtoCstr((char *)theText);
    return((float)  atof(theText));}

EndFunction

GiveEditItemNum(theDlog,theEditItem,theNum)
DialogPtr    theDlog;
int          theEditItem;
Str255       theNum;{
int          itemType;
Handle       item;
Rect         itemBox;
        GetDlgItem(theDlog,theEditItem,&itemType,&item,&itemBox);
        SetIText(item,theNum);}

EndFunction

doGraphicsOption() {
DialogPtr    theDlog;
DialogRecord theDRecord;
int          itemType,ViewCase;
int          DoModalLoop = 1;
Handle       item;
int          itemHit;
Rect         itemBox;
static  Str255
        TopViewParam[7],SideViewParam[7],FrontViewParam[7],AllViewsParam[7];
long         tempNum;
COUNTER      i;
static int   initGraphics = 1,initFrontView = 1,initTopView = 1;
static int   initSideView = 1,showTitlesFlag = 0,showHomeFlag = 0;
        theDlog = GetNewDialog(GraphicsID,&theDRecord, -1L);
        frameItem(theDlog,OK,16);
        if(initGraphics){
            initGraphics = 0;
            GetDlgItem(theDlog,AllViews,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
        ViewCase = 3;
            GiveEditItemNum(theDlog,vertTop,topY);
            GiveEditItemNum(theDlog,horTop,topX);
            GiveEditItemNum(theDlog,vertFront,frontY);

```



```

GiveEditItemNum(theDlog,horFront,frontX);
GiveEditItemNum(theDlog,vertSide,sideY);
GiveEditItemNum(theDlog,horSide,sideX);
GiveEditItemNum(theDlog,sizeImage,"p1.0");
view.top.y = GetEditItemInt(theDlog,vertTop);
view.top.x = GetEditItemInt(theDlog,horTop);
view.front.y = GetEditItemInt(theDlog,vertFront);
view.front.x = GetEditItemInt(theDlog,horFront);
view.side.y = GetEditItemInt(theDlog,vertSide);
view.side.x = GetEditItemInt(theDlog,horSide);
scale = GetEditItemFloat(theDlog,sizeImage);
sprintf(AllViewsParam[0],"p%.2f ",scale);
scale *= Scale3View;

view_case = 3;
NumToString((long) view.top.y,&AllViewsParam[1]);
NumToString((long) view.top.x,&AllViewsParam[2]);
NumToString((long) view.front.y,&AllViewsParam[3]);
NumToString((long) view.front.x,&AllViewsParam[4]);
NumToString((long) view.side.y,&AllViewsParam[5]);
NumToString((long) view.side.x,&AllViewsParam[6]);
else {
    if(showTitlesFlag){
        GetDItem(theDlog,showTitles,&itemType,&item,&itemBox);
        SetCtlValue(item,1);}
    if(showHomeFlag){
        GetDItem(theDlog,showHome,&itemType,&item,&itemBox);
        SetCtlValue(item,1);}
    switch(view_case){
        case 0:
            GetDItem(theDlog,TopView,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
            ViewCase = 0;
            for(i = sizeImage;i<= horSide;++i)
                GiveEditItemNum(theDlog,i,TopViewParam[i-9]);
            break;
        case 1:
            GetDItem(theDlog,FrontView,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
            ViewCase = 1;
            for(i = sizeImage;i<= horSide;++i)
                GiveEditItemNum(theDlog,i,FrontViewParam[i-9]);
            break;
        case 2:
            GetDItem(theDlog,SideView,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
            for(i = sizeImage;i<= horSide;++i)
                GiveEditItemNum(theDlog,i,SideViewParam[i-9]);
            ViewCase = 2;
            break;
        case 3:
            GetDItem(theDlog,AllViews,&itemType,&item,&itemBox);
            SetCtlValue(item,1);
            ViewCase = 3;
            for(i = sizeImage;i<= horSide;++i)

```

```

        GiveEditItemNum(theDlog,i,AllViewsParam[i-9]);
        break;}}
while (DoModalLoop) {
ModalDialog( FALSE,&itemHit);
switch (itemHit) {
    case OK:
        graphicsCond = 1;
        if(showTitlesFlag)
            ShowTitles = 1;
        else
            ShowTitles = 0;
        if(showHomeFlag)
            ShowHome = 1;
        else
            ShowHome = 0;
        switch(ViewCase){
            case 0:
                view.top.y = GetEditItemInt(theDlog,vertTop);
                view.top.x = GetEditItemInt(theDlog,horTop);
                scale = GetEditItemFloat(theDlog,sizeImage);
                sprintf(TopViewParam[0],"p%.2f ",scale);
                scale *= ScaleTop;
                view_case = 0;
                NumToString((long) view.top.y,&TopViewParam[1]);
                NumToString((long) view.top.x,&TopViewParam[2]);
                break;
            case 1:
                view.front.y = GetEditItemInt(theDlog,vertFront);
                view.front.x = GetEditItemInt(theDlog,horFront);
                scale = GetEditItemFloat(theDlog,sizeImage);
                sprintf(FrontViewParam[0],"p%.2f ",scale);
                scale *= ScaleFront;
                view_case = 1;
                NumToString((long) view.front.y,&FrontViewParam[3]);
                NumToString((long) view.front.x,&FrontViewParam[4]);
                break;
            case 2:
                view.side.y = GetEditItemInt(theDlog,vertSide);
                view.side.x = GetEditItemInt(theDlog,horSide);
                scale = GetEditItemFloat(theDlog,sizeImage);
                sprintf(SideViewParam[0],"p%.2f ",scale);
                scale *= ScaleSide;
                NumToString((long) view.side.y,&SideViewParam[5]);
                NumToString((long) view.side.x,&SideViewParam[6]);
                view_case = 2;
                break;
            case 3:
                view.top.y = GetEditItemInt(theDlog,vertTop);
                view.top.x = GetEditItemInt(theDlog,horTop);
                view.front.y = GetEditItemInt(theDlog,vertFront);
                view.front.x = GetEditItemInt(theDlog,horFront);
                view.side.y = GetEditItemInt(theDlog,vertSide);
                view.side.x = GetEditItemInt(theDlog,horSide);
                scale = GetEditItemFloat(theDlog,sizeImage);

```

```

        sprintf(AllViewsParam[0], "\p%.2f ", scale);
        scale *= Scale3View;
        view_case = 3;
        NumToString((long) view.top.y, &AllViewsParam[1]);
        NumToString((long) view.top.x, &AllViewsParam[2]);
        NumToString((long) view.front.y, &AllViewsParam[3]);
        NumToString((long) view.front.x, &AllViewsParam[4]);
        NumToString((long) view.side.y, &AllViewsParam[5]);
        NumToString((long) view.side.x, &AllViewsParam[6]);
        break;
    default:
        break;
    }
    CloseDialog(theDlog);
    DoModalLoop = 0;
    break;
case cancel:
    CloseDialog(theDlog);
    DoModalLoop = 0;
    break;
case TopView:
    GetDItem(theDlog, TopView, &itemType, &item, &itemBox);
    SetCtlValue(item, 1);
    GetDItem(theDlog, FrontView, &itemType, &item, &itemBox);
    SetCtlValue(item, 0);
    GetDItem(theDlog, SideView, &itemType, &item, &itemBox);
    SetCtlValue(item, 0);
    GetDItem(theDlog, AllViews, &itemType, &item, &itemBox);
    SetCtlValue(item, 0);
    if(initTopView){
        initTopView = 0;
        GiveEditItemNum(theDlog, vertTop, TopY);
        GiveEditItemNum(theDlog, horTop, TopX);
        GiveEditItemNum(theDlog, vertFront, "\p0");
        GiveEditItemNum(theDlog, horFront, "\p0");
        GiveEditItemNum(theDlog, vertSide, "\p0");
        GiveEditItemNum(theDlog, horSide, "\p0");
        GiveEditItemNum(theDlog, sizeImage, "\p1.0");
        tempNum = GetEditItemFloat(theDlog, sizeImage);
        sprintf(TopViewParam[0], "\p%.2f ", tempNum);
        tempNum = GetEditItemInt(theDlog, vertTop);
        NumToString(tempNum, &TopViewParam[1]);
        tempNum = GetEditItemInt(theDlog, horTop);
        NumToString(tempNum, &TopViewParam[2]);
        NumToString((long) 0, &TopViewParam[3]);
        NumToString((long) 0, &TopViewParam[4]);
        NumToString((long) 0, &TopViewParam[5]);
        NumToString((long) 0, &TopViewParam[6]);
    }
    else
        for(i = sizeImage; i <= horSide; ++i)
            GiveEditItemNum(theDlog, i, TopViewParam[i-9]);
    ViewCase = 0;
    break;
case FrontView:
    GetDItem(theDlog, TopView, &itemType, &item, &itemBox);

```

```

SetCtlValue(item,0);
GetDItem(theDlog,FrontView,&itemType,&item,&itemBox);
SetCtlValue(item,1);
GetDItem(theDlog,SideView,&itemType,&item,&itemBox);
SetCtlValue(item,0);
GetDItem(theDlog,AllViews,&itemType,&item,&itemBox);
SetCtlValue(item,0);
if(initFrontView){
    initFrontView = 0;
    GiveEditItemNum(theDlog,vertTop,"p0");
    GiveEditItemNum(theDlog,horTop,"p0");
    GiveEditItemNum(theDlog,vertFront,FrontY);
    GiveEditItemNum(theDlog,horFront,FrontX);
    GiveEditItemNum(theDlog,vertSide,"p0");
    GiveEditItemNum(theDlog,horSide,"p0");
    GiveEditItemNum(theDlog,sizeImage,"p1.0");
    tempNum = GetEditItemFloat(theDlog,sizeImage);
    sprintf(FrontViewParam[0],"p%.2f ",tempNum);
    NumToString((long) 0,&FrontViewParam[1]);
    NumToString((long) 0,&FrontViewParam[2]);
    tempNum = GetEditItemInt(theDlog,vertFront);
    NumToString(tempNum,&FrontViewParam[3]);
    tempNum = GetEditItemInt(theDlog,horFront);
    NumToString(tempNum,&FrontViewParam[4]);
    NumToString((long) 0,&FrontViewParam[5]);
    NumToString((long) 0,&FrontViewParam[6]);
}
else
    for(i = sizeImage;i<= horSide;++i)
        GiveEditItemNum(theDlog,i,FrontViewParam[i-9]);
ViewCase = 1;
break;
case SideView:
    GetDItem(theDlog,TopView,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,FrontView,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,SideView,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    GetDItem(theDlog,AllViews,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,AllViews,&itemType,&item,&itemBox);
    if(initSideView){
        initSideView = 0;
        GiveEditItemNum(theDlog,vertTop,"p0");
        GiveEditItemNum(theDlog,horTop,"p0");
        GiveEditItemNum(theDlog,vertFront,"p0");
        GiveEditItemNum(theDlog,horFront,"p0");
        GiveEditItemNum(theDlog,vertSide,SideY);
        GiveEditItemNum(theDlog,horSide,SideX);
        GiveEditItemNum(theDlog,sizeImage,"p1.0");
        tempNum = GetEditItemFloat(theDlog,sizeImage);
        sprintf(SideViewParam[0],"p%.2f ",tempNum);
        NumToString((long) 0,&SideViewParam[1]);
        NumToString((long) 0,&SideViewParam[2]);
    }

```

```

        NumToString((long) 0,&SideViewParam[3]);
        NumToString((long) 0,&SideViewParam[4]);
        tempNum = GetEditItemInt(theDlog,vertSide);
        NumToString(tempNum,&SideViewParam[5]);
        tempNum = GetEditItemInt(theDlog,horSide);
        NumToString(tempNum,&SideViewParam[6]);
    else
        for(i = sizeImage;i<= horSide;++i)
            GiveEditItemNum(theDlog,i,SideViewParam[i-9]);
    ViewCase = 2;
    break;
case AllViews:
    GetDItem(theDlog,TopView,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,FrontView,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,SideView,&itemType,&item,&itemBox);
    SetCtlValue(item,0);
    GetDItem(theDlog,All Views,&itemType,&item,&itemBox);
    SetCtlValue(item,1);
    for(i = sizeImage;i<= horSide;++i)
        GiveEditItemNum(theDlog,i,All ViewsParam[i-9]);
    ViewCase = 3;
    break;
case showTitles:
    GetDItem(theDlog,showTitles,&itemType,&item,&itemBox);
    if(showTitlesFlag){
        showTitlesFlag = 0;
        SetCtlValue(item,0);}
    else {
        showTitlesFlag = 1;
        SetCtlValue(item,1);}
    break;
case showHome:
    GetDItem(theDlog,showHome,&itemType,&item,&itemBox);
    if(showHomeFlag){
        showHomeFlag = 0;
        SetCtlValue(item,0);}
    else {
        showHomeFlag = 1;
        SetCtlValue(item,1);}
    break;}}

```

EndFunction

```

doHilite(theDialog)
DialogPtr theDialog;{
int itemType;
Handle item;
Rect itemBox;
if(saveCartRun) {
    GetDItem(CartesianSpaceDlog,saveRun,&itemType,&item,&itemBox);
    HiliteControl(item,0);/* enable save run */
else if(saveJointRun) {
    GetDItem(JointSpaceDlog,saveRun,&itemType,&item,&itemBox);

```

```

        HiliteControl(item,0);)/* enable save run */
    else {
        GetDItem(theDialog,saveRun,&itemType,&item,&itemBox);
        HiliteControl(item,255);)/* disable save run */
    if(LTM->flags.simulate) {
        GetDItem(theDialog,simulationMode,&itemType,&item,&itemBox);
        SetCtlValue(item,1);
        GetDItem(theDialog,realMode,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        if(SetGraphics){
            GetDItem(theDialog,graphicsOptions,&itemType,&item,&itemBox);
            HiliteControl(item,0);)/* enable graphics options */
        }
    else {
        GetDItem(theDialog,simulationMode,&itemType,&item,&itemBox);
        SetCtlValue(item,0);
        GetDItem(theDialog,realMode,&itemType,&item,&itemBox);
        SetCtlValue(item,1);
        GetDItem(theDialog,graphicsOptions,&itemType,&item,&itemBox);
        HiliteControl(item,255);)/* disable graphics options */
    }
EndFunction

IsOKtoGO(){
    int      itemType;
    Handle    item;
    Rect      itemBox;
    if(graphicsCond){
        if(CartesianSpaceDlog) {
            if(CsetupCond && CpathCond && (!theCPathChanged)){
                GetDItem(CartesianSpaceDlog,GO,&itemType,&item,&itemBox);
                HiliteControl(item,0);
                CartGO = 1;})/* enable Cartesian GO button */
            if(JointSpaceDlog) {
                if(JsetupCond && JpathCond && (!theJPathChanged)){
                    GetDItem(JointSpaceDlog,GO,&itemType,&item,&itemBox);
                    HiliteControl(item,0);
                    JointGO = 1;})})/* enable Joint GO button */
            }
        }
    }
EndFunction

GetCorrectCursor() {
    Point      mouseLoc;
    CursHandle theCursor;
    GetMouse(&mouseLoc);
    if((mouseLoc.v <= PathW.Ptr->portRect.top) || (mouseLoc.v >= (PathW.Ptr-
>portRect.bottom)))
        SetCursor(&arrow);
    else if((mouseLoc.h <= PathW.Ptr->portRect.left) || (mouseLoc.h >= (PathW.Ptr-
>portRect.right - 16)))
        SetCursor(&arrow);
    else {
        theCursor = GetCursor(iBeamCursor);
        SetCursor(*theCursor);
    }
    ShowCursor();
}
EndFunction

```

```

GetCorrectCursor2() {
Point      mouseLoc;
CursHandle theCursor;
    GetMouse(&mouseLoc);
    if((mouseLoc.v <= PathW.Ptr->portRect.top) || (mouseLoc.v >= (PathW.Ptr-
>portRect.bottom - 16)))
        SetCursor(&arrow);
    else if((mouseLoc.h <= PathW.Ptr->portRect.left) || (mouseLoc.h >= (PathW.Ptr-
>portRect.right - 16)))
        SetCursor(&arrow);
    else {
        theCursor = GetCursor(iBeamCursor);
        SetCursor(*theCursor);}
    ShowCursor();}
EndFunction

```

```

int DoPathScroll(theWindow,Where)
WindowPtr      theWindow;
Point          Where;{
ControlHandle   whichControl;
int            controlPart;
    controlPart = FindControl(Where,theWindow,&whichControl);
    if(!controlPart)
        return(0);
    else
        DoScroll(&controlPart,Where);}
EndFunction

```

```

pascal void ScrollText(theControl,thePart)
ControlHandle theControl;
int           thePart;{
int           lineHeight,top,bottom;
static int    delta,oldValue;
    switch(thePart){
        case inUpButton:
            delta = -1;
            break;
        case inDownButton:
            delta = 1;
            break;
        case inPageUp:
            lineHeight = (**PathW.Data.editRec).lineHeight;
            top = (**PathW.Data.editRec).viewRect.top;
            bottom = (**PathW.Data.editRec).viewRect.bottom;
            delta = (top - bottom)/lineHeight + 1;
            break;
        case inPageDown:
            lineHeight = (**PathW.Data.editRec).lineHeight;
            top = (**PathW.Data.editRec).viewRect.top;
            bottom = (**PathW.Data.editRec).viewRect.bottom;
            delta = (bottom - top)/lineHeight - 1;

```

```

                                break;
                        default:
                                break;}
        if(thePart){
            oldValue = GetCtlValue(theControl);
            SetCtlValue(theControl,oldValue + delta);
            AdjustText();}
EndFunction

AdjustText(){
int    oldScroll,newScroll;
int    viewRectTop,destRectTop,lineHeight;
        HLock(PathW.Data.editRec);
            viewRectTop = (**PathW.Data.editRec).viewRect.top;
            destRectTop = (**PathW.Data.editRec).destRect.top;
            lineHeight = (**PathW.Data.editRec).lineHeight;
            oldScroll = viewRectTop - destRectTop;
            newScroll = GetCtlValue(PathW.Data.VScrollBar) * lineHeight;
            TEScroll(0,(oldScroll - newScroll),PathW.Data.editRec);
        HUnlock(PathW.Data.editRec);}
EndFunction

DoScroll(thePart,where)
int    *thePart;
Point  where;{
pascal void ScrollText();
        if(*thePart == inThumb){
            *thePart = TrackControl(PathW.Data.VScrollBar,where,NIL);
            AdjustText();}
        else
            *thePart = TrackControl(PathW.Data.VScrollBar,where,ScrollText);}
EndFunction

AdjustScrollBar(){
int    windowHeight; /* height of text rect in lines */
int    maxTop;        /* maximum value for top line in window */
int    top,bottom,lineHeight,numLines;
        top = (**PathW.Data.editRec).viewRect.top;
        bottom = (**PathW.Data.editRec).viewRect.bottom;
        lineHeight = (**PathW.Data.editRec).lineHeight;
        numLines = (**PathW.Data.editRec).nLines;
        windowHeight = (bottom - top)/lineHeight;
        maxTop = numLines - windowHeight;
        if(maxTop <= 0){
            maxTop = 0;
            HiliteControl(PathW.Data.VScrollBar,inactive);}
        else {
            HiliteControl(PathW.Data.VScrollBar,active);}
        SetCtlMax(PathW.Data.VScrollBar,maxTop);}
EndFunction

```


Appendix G
MacInterface.r
(resource files)

LTM_Mac.rsrc
APPLLTMR

Type WIND
 ,129
Simulation Output - Cartesian Space
45 10 332 508
Visible GoAway
4
0

Type WIND
 ,130
Simulation Output - Joint Space
45 10 332 508
Visible GoAway
4
0

Type WIND
 ,131
Path
45 10 332 508
Visible GoAway
4
0

Type WIND
 ,132
Simulation Output - Cartesian Space
80 10 480 600
Visible GoAway
4
0

Type WIND
 ,133
Simulation Output - Joint Space
80 10 480 600
Visible GoAway
4
0

Type WIND
 ,134

Help File
45 10 332 508
Visible GoAway
4
0

Type WIND
 ,135
Reference Stance - MacPlus
45 10 332 508
Visible GoAway
4
0

Type WIND
 ,136
Reference Stance MacII
80 10 480 600
Visible GoAway
4
0

Type CNTL
 ,128
vertical scroll bar
-1 483 287 499
Visible
16
0
0 100 0

 ,129
horizontal scroll bar
272 -1 287 499
Visible
16
0
0 100 0

Type DLOG
 ,258
ROBOTICS__Cartesian Space
100 150 280 450
Visible GoAway
4
1
258

Type DITL
12,258

button enabled
10 210 30 290
GO

button enabled
30 20 46 170
Set Up Run Features

button enabled
52 20 68 170
Save Run

button enabled
94 20 110 130
Create A Path

button enabled
116 20 132 130
Alter A Path

button enabled
138 20 154 130
Choose A Path

radiobutton
94 160 110 270
Real Motion

radiobutton
120 160 136 270
Simulation

button enabled
146 170 162 296
Graphics Options

staticText
10 10 26 130
Run Options:

staticText
76 10 90 130
Path Options:

staticText
70 150 86 270
Mode Options:

Type DLOG

,259

ROBOTICS__Joint Space

120 170 300 470

Visible GoAway

4

2

259

Type DITL

,259

12

button enabled

10 210 30 290

GO

button enabled

30 20 46 170

Set Up Run Features

button enabled

52 20 68 170

Save Run

button enabled

94 20 110 130

Create A Path

button enabled

116 20 132 130

Alter A Path

button enabled

138 20 154 130

Choose A Path

radiobutton

94 160 110 270

Real Motion

radiobutton

120 160 136 270

Simulation

button enabled

146 170 162 296

Graphics Options

staticText

10 10 26 130

Run Options:

staticText

76 10 90 130

Path Options:

staticText

70 150 86 270

Mode Options:

* These are the menu resources

TYPE MENU

,129

File ;; menu name

Quit Robotics

TYPE MENU

,130

Edit

Undo

Cut

Copy

Paste

Clear

Type MENU

,131

TeleRobotics

TYPE MENU

,132

Robotics

Cartesian Space

Joint Space

TYPE MENU

,133

Help

Demonstration

Reference for Joint Angles

Cartesian Space Dialog

Joint Space Dialog

* the following are for the Modal Dialog to CREATE a CARTESIAN path

Type DLOG
 ,260
No Title
30 100 300 480
Visible NoGoAway
1
0
260

Type DITL
 ,260
34

button
10 280 30 360
OK

button
40 280 60 360
Cancel

editText ::path name
88 284 108 376

editText
10 144 30 170
1

checkBox
40 4 60 210
Always same Move Time

staticText
64 4 84 120
the Move Time:

editText
64 124 84 180
4

checkBox
88 4 108 184
Always same Acc Factor

staticText
112 4 132 120
the Acc Factor:

editText
112 124 132 180
.1

staticText

138 10 153 200
ORIENTATION MATRIX

editText ;; x-axis vector
156 5 176 65
0.0000

editText
184 5 204 65
0.0000

editText
212 5 232 65
1.0000

editText ;; y-axis vector
156 73 176 133
0.0000

editText
184 73 204 133
-1.0000

editText
212 73 232 133
0.0000

editText ;; z-axis vector
156 141 176 199
1.0000

editText
184 141 204 199
0.0000

editText
212 141 232 199
0.0000

editText ;; position
156 220 176 280
32.0000

editText
184 220 204 280
-23.0000

editText
212 220 232 280
0.0000

editText ;; move time
237

240 94 260 120

editText ;; acc factor
240 234 260 280

staticText
138 218 153 282
POSITION

staticText
138 300 158 340
NODE:

editText ;; the node indicator
138 344 158 370
1

staticText
240 5 260 90
Move Time:

staticText
240 130 260 230
Acc Factor:

button
170 290 190 374
Node OK

button
200 290 220 374
Node Cancel

staticText
10 4 30 140
Number of Nodes:

staticText
88 190 108 280
PATH NAME:

* Dialog for CHOOSE PATH

Type DLOG
 ,262
No Title
30 130 120 380
Visible NoGoAway
1
0
262

Type DITL
 ,262
4

button
10 10 30 120
OK

button
10 130 30 240
Cancel

editText
50 120 70 240
Demo1

staticText
50 10 70 110
Name of Path:

* Dialog for Alter Old Path

Type DLOG
 ,263
No Title
30 130 120 380
Visible NoGoAway
1
0
263

Type DITL
 ,263
4

button
10 10 30 120
OK

button
10 130 30 240
Cancel

editText
50 120 70 240
Demo

staticText
50 10 70 110
Name of Path:

Type DLOG /* set up (cartesian path) dialog */
 ,264

No Title
30 100 310 480
Visible NoGoAway
1
0
264

Type DITL
 ,264
40

button
6 240 22 370
OK

button
28 240 40 370
Cancel

editText ;; Perf criterion switch number
36 148 52 200
1

editText ;; delta_t Parameter
60 148 76 200
0.05

editText ;; Parameter 3
84 148 100 200
None

editText ;; Parameter 4
108 148 124 200
None

editText ;; image # parameter
170 148 184 168
5

editText ;;Theta1
120 323 135 363

editText ;;Theta2
143 323 158 363

editText ;;Theta3
166 323 181 363

editText ;;Theta4
189 323 204 363

editText ;;Theta5

240

212 323 227 363

editText ;;Theta6
235 323 250 363

editText ;;Theta7
258 323 273 363

checkBox
156 40 168 120
Graphics

checkBox
188 40 200 120
Numbers

radioButton
234 20 246 140
Least Norm

radioButton
250 20 262 140
Optimized

radioButton
74 240 86 368
Home Position

radioButton
95 240 110 365
Specify Degrees

staticText
8 100 20 200
PREFERENCES

staticText
44 270 56 360
STARTING

staticText
56 246 68 366
CONFIGURATION

staticText
120 254 135 309
Theta 1 :

staticText
143 254 158 309
Theta 2 :

staticText
166 254 181 309

Theta 3 :

staticText
189 254 204 309
Theta 4 :

staticText
212 254 227 309
Theta 5 :

staticText
235 254 250 309
Theta 6 :

staticText
258 254 273 309
Theta 7 :

staticText
24 8 40 80
Parameters:

staticText
40 40 52 140
Perf.Criterion:

staticText
64 40 76 140
delta_t:

staticText
64 208 76 238
sec

staticText
88 40 100 140
Parameter 3:

staticText
112 40 124 140
Parameter 4:

staticText
132 8 148 200
Type of output:

staticText
172 48 184 140
Output every

staticText
172 176 184 236
images

staticText
210 8 226 200
Type of solution:

Type DLOG /* Set up run (joint path) dialog */
 ,266
No Title
80 150 292 490
Visible NoGoAway
1
0
266

Type DITL
 ,266
14

button
10 230 30 330
OK

button
36 230 56 330
Cancel

editText ;; % of max speed specification
10 164 24 200
90

editText ;; delta_t Parameter (time step)
32 164 46 200
0.01

editText ;; image # parameter
104 128 120 148
20

checkBox
80 20 96 100
Graphics

checkBox
128 20 144 100
Numbers

staticText
10 8 26 160
% of Max Speed:

staticText
32 8 48 160
Specify delta_t:

staticText
60 8 76 120
Type of output:

staticText
104 30 120 120
Output every

staticText
104 156 120 216
images

staticText
156 8 172 200
Starting Configuration:

staticText
180 20 196 260
The current joint angles are used.

Type ALRT
 ,130
50 150 250 430
130
4444

Type DITL
 ,130
2

button
150 40 170 240
Continue

staticText
75 10 95 270
Please specify a file name for the path

Type ALRT
 ,131
50 50 125 450
131
5555
 ;; Error Alert

Type DITL
 ,131
2

button
50 100 70 300
Continue

staticText
25 10 45 390
^0 ^1

Type ALRT
 ,132 ;; Advice Alert
50 50 125 450
132
4444

Type DITL
 ,132
2

button
50 100 70 300
Continue

staticText
25 10 45 390
^0^1

Type ALRT
 ,133 ;; Query Alert
50 50 150 450
132
5555

Type DITL
 ,133
4

button
70 10 90 110
Yes

button
70 290 90 390
Cancel

button
70 150 90 250
No

staticText
30 10 50 390
^0

Type ALRT ;; Unsave
 ,134
50 50 150 450
134
4444

Type DITL
 ,134
7

button
70 10 90 110
Unsave

button
70 290 90 390
Cancel

staticText
5 10 25 200
Unsave file named:

editText
30 10 50 200
^0

button
70 150 90 250
Drive

button
40 150 60 250
Eject

staticText
70 150 90 250
^1

* the following are for the Modal Dialog to CREATE a JOINT path

Type DLOG
 ,265
No Title
50 150 280 500
Visible NoGoAway
1
0
265

Type DITL
 ,265
28

button
10 240 30 340
OK

button
36 240 56 340

Cancel

editText ;;joint path name
10 134 30 190

editText ;;Number of nodes
40 134 60 160
1

editText ;;Current node on
70 134 90 160
1

editText ;;Theta1
100 74 120 110

editText ;;Theta2
130 74 150 110

editText ;;Theta3
160 74 180 110

editText ;;Theta4
100 184 120 220

editText ;;Theta5
130 184 150 220

editText ;;Theta6
160 184 180 220

editText ;;Theta7
100 294 120 330

editText ;;Acceleration Factor
130 294 150 330
.1

button
160 230 180 340
Node OK

staticText
10 10 30 130
Name Joint Path:

staticText
40 10 60 130
Number of Nodes:

staticText
70 10 90 130
Current Node:

staticText
190 10 206 340
NOTE 1: Express joint angles in degrees.

staticText
210 10 226 346
NOTE 2: For reference stance see Help menu.

staticText
100 10 120 70
Theta 1 :

staticText
130 10 150 70
Theta 2 :

staticText
160 10 180 70
Theta 3 :

staticText
100 120 120 180
Theta 4 :

staticText
130 120 150 180
Theta 5 :

staticText
160 120 180 180
Theta 6 :

staticText
100 230 120 290
Theta 7 :

staticText
128 230 140 290
Acc.

staticText
140 230 152 290
Factor:

* the following is for the Modal Dialog for GRAPHICS OPTIONS

Type DLOG
 ,267
No Title
80 150 310 450
Visible NoGoAway
1
0

267

Type DITL
 ,267
26

button
10 210 30 290
OK

button
36 210 56 290
Cancel

radiobutton enabled
30 10 46 95
Top

radiobutton enabled
50 10 66 95
Front

radiobutton enabled
30 105 46 190
All Views

radiobutton enabled
50 105 66 190
Side

checkBox
84 10 96 110
Shoulder

checkBox
98 10 110 250
Identification of View

checkBox
112 10 126 250
Definition of Top, Front, Side

* The following are edit text items for image's SCREEN POSITION

editText ;;image size
148 56 162 90

editText ;;top vertical
190 120 204 150

editText ;;top horizontal
212 120 226 150

editText 190 180 204 210	::front vertical
editText 212 180 226 210	::front horizontal
editText 190 240 204 270	::side vertical
editText 212 240 226 270	::side horizontal
staticText 10 10 26 160 Specify A View:	
staticText 70 10 82 90 Show:	
staticText 130 10 144 110 Adjust Image:	
staticText 148 10 162 50 Size:	
staticText 170 10 186 106 Location:	
staticText 190 10 204 116 Vertical.....	
staticText 212 10 226 116 Horizontal.....	
staticText 170 110 186 160 Top	
staticText 170 170 186 220 Front	
staticText 170 230 186 280 Side	

Type DLOG
268
Help Cartesian Dialog
100 150 280 450
Visible GoAway
4
1
268

Type DITL
268
13

button enabled
10 190 35 290
All Help Files

button enabled
40 190 65 290
Supplement

button enabled
30 20 46 170
Set Up Run Features

button enabled
52 20 68 170
Save Run

button enabled
94 20 110 130
Create A Path

button enabled
116 20 132 130
Alter A Path

button enabled
138 20 154 130
Choose A Path

radiobutton
94 160 110 270
Real Motion

radiobutton
120 160 136 270
Simulation

button enabled
146 170 162 296
Graphics Options

staticText
10 10 26 130
Run Options:

staticText
76 10 90 130
Path Options:

staticText
70 150 86 270
Mode Options:

Type DLOG
 ,269
Help Joint Dialog
100 150 280 450
Visible GoAway
4
1
269

Type DITL
 ,269
13

button enabled
10 190 35 290
All Help Files

button enabled
40 190 65 290
Supplement

button enabled
30 20 46 170
Set Up Run Features

button enabled
52 20 68 170
Save Run

button enabled
94 20 110 130
Create A Path

button enabled
116 20 132 130
Alter A Path

button enabled
138 20 154 130
Choose A Path

radiobutton
94 160 110 270
Real Motion

radiobutton
120 160 136 270
Simulation

button enabled
146 170 162 296
Graphics Options

staticText
10 10 26 130
Run Options:

staticText
76 10 90 130
Path Options:

staticText
70 150 86 270
Mode Options:

Type DLOG
 ,270
Help Joint Dialog
100 150 210 450
Visible GoAway
4
1
270

Type DITL
 ,270
2

button enabled
20 10 50 290
Help on Redundancy Demonstration

button enabled
60 10 90 290
Joint Angle Reference Defined

Type ALRT
 ,135
50 330 125 500
135
5555

:: Joint Limit Alert

Type DITL
 ,135
3

button
50 10 70 160
Continue

staticText
10 42 20 160
Joint ^0 limit

staticText
25 20 45 160
Limit = ^1 degrees.

Type ALRT
 ,136
50 330 125 500
136
5555

:: Velocity limit Alert

Type DITL
 ,136
2

button
50 10 70 160
Continue

staticText
20 10 40 160
JOINT VELOCITY LIMIT

Type ALRT
 ,137
50 150 250 430
137
4444

Type DITL
 ,137
2

button
150 40 170 240
Continue

staticText
75 10 95 270
Please specify a run name

Type DLOG
 ,138
No Title
80 160 170 440
Visible NoGoAway
1
0
138

Type DITL
 ,138
3

button
60 150 80 270
Continue Anyway

button
60 10 80 130
Cancel

staticText
20 40 40 260
THAT NAME ALREADY IN USE !!

Appendix H

HEADER FILES

```

/* .LTM.h header file */

#define EQUALS      =
#define AND         &&
#define OR          ||
#define NOT         !
#define THETA_1     LTM->l.s.shld.setpnts.pitch_pos
#define THETA_2     LTM->l.s.shld.setpnts.yaw_pos
#define THETA_3     LTM->l.s.elbow.setpnts.pitch_pos
#define THETA_4     LTM->l.s.elbow.setpnts.yaw_pos
#define THETA_5     LTM->l.s.wrist.setpnts.pitch_pos
#define THETA_6     LTM->l.s.wrist.setpnts.yaw_pos
#define THETA_7     LTM->l.s.wrist.setpnts.roll
#define MaxSpd12    496      /* rad/sec */
#define MaxSpdRoll  12.5     /* rad/sec should be 12.5 rad/sec */
#define SGearRatio  150.8
#define LGearRatio  522.2
#define RGearRatio  25
#define Constant_a2 23.00
#define Constant_a4 20.00
#define Constant_d7 12.00
#define EndFunction

#define MacType      0 /* MacPlus = 0, Mac11 = 1 */
#if (MacType)
    #define TopX      "\p175"
    #define TopY      "\p200"
    #define FrontX    "\p175"
    #define FrontY    "\p100"
    #define SideX     "\p275"
    #define SideY     "\p125"
    #define topX      "\p100"
    #define topY      "\p20"
    #define frontX     "\p100"
    #define frontY     "\p250"
    #define sideX      "\p450"
    #define sideY      "\p250"
    #define ScaleTop   10
    #define ScaleSide  10
    #define ScaleFront 10
    #define Scale3View 5
    #define LimitLineX1 475
    #define LimitLineX2 612
#else
    #define TopX      "\p175"
    #define TopY      "\p200"
    #define FrontX    "\p175"

```

```

#define FrontY      "\p100"
#define SideX       "\p250"
#define SideY       "\p125"
#define topX        "\p50"
#define topY        "\p50"
#define frontX      "\p50"
#define frontY      "\p150"
#define sideX       "\p350"
#define sideY       "\p150"
#define ScaleTop    10
#define ScaleSide   10
#define ScaleFront  10
#define Scale3View  5
#define LimitLineX1 375
#define LimitLineX2 512
#endif

```

```

/* macros */

```

```

#define max(a,b)    (( a>b ) ? a : b)
#define min(a,b)    (( a<b ) ? a : b)

```

```

typedef int    COUNTER;
typedef int    SCALAR;
typedef int    FLAG;

typedef struct {
    double pos1;
    double pos2;
    double pos3;
    double s1;
    double s2;
    double s3;
    double n1;
    double n2;
    double n3;
    double a1;
    double a2;
    double a3;
} HOMOGEN_TRANS;

```

```

typedef struct {
    float pos1;
    float pos2;
    float pos3;
    float s1;
    float s2;
    float s3;
    float n1;
    float n2;
    float n3;
    float a1;

```

```

float a2;
float a3;
float move_time;
int end;
float accFac;
} NODE;

```

```

typedef struct {
    float J1;
    float J2;
    float J3;
    float J4;
    float J5;
    float J6;
    float J7;
    float accFac;
    int end;
} CONFIG;

```

```

typedef struct {
    double x;
    double y;
    double z;
    double phi;
    double theta;
    double psi;
} HAND_POS_OR;

```

```

typedef struct {
    double one;
    double two;
    double three;
    double four;
    double five;
    double six;
    double seven;
} JOINT_ANGLES;

```

```

typedef struct {
    double dx;
    double dy;
    double dz;
    double dThetaX;
    double dThetaY;
    double dThetaZ;
} INCREMENTS;

```

```

typedef struct {
    /* e11 stands for element 11 etc. */

```

```

        double e11;
        double e12;
        double e13;
        double e21;
        double e22;
        double e23;
        double e31;
        double e32;
        double e33;
    } MATRIX;

```

```

struct    VECTOR
{
    double x;
    double y;
    double z;
};

```

```

struct    ViewData
{
    struct VECTOR    elbow;
    struct VECTOR    wrist;
    struct VECTOR    hand;
    struct VECTOR    orient;
};

```

```

struct    views
{
    struct VECTOR    top;
    struct VECTOR    front;
    struct VECTOR    side;
};

```

```

/* MacInterface.h    header file*/

```

```

#define MACPLUS
#ifdef MACPLUS
# define CartesianWID    129
# define JointWID        130
# define PathWID         131
# define topBound        20
# define leftBound       0
# define bottomBound     614
# define rightBound      986
#endif
#ifdef MACII
# define PathWID          131
# define CartesianWID     132
# define JointWID         133
# define topBound         60

```

```

# define leftBound      0
# define bottomBound    714
# define rightBound     1086
#endif
# define K_SCROLL       135      /*resource ID k scroll */
# define active         0
# define inactive       255
# define UP             1
# define DOWN           2
# define DRVVR          0x44525652L /* string DRVVR as a long */

# define drvNum         1
# define MAXnodeData    94
# define MAXnumNodes    10
# define MAXwindows     10
# define ScrlBarWidth   15
# define inRadioButton  11

/* Some Pascal constants used in Inside Macintosh */
# define NIL            0L

/* Modal Dialog common indices */
# define OK             1
# define cancel         2

/* resource IDs of menus */
# define FileID        129
# define EditID        130
# define TeleRobID     131
# define RoboticsID    132

/* ALRT IDs */
# define noPathNmID    130
# define ErrorAlert    131
# define AdviceAlert   132
# define QueryAlert    133
# define UnsaveID      134
# define JointLmtID    135
# define VelErrID      136
# define noRunNmID     137
# define nameInUseID   138

/* the kind of manipulator error */
# define joint_limit    1
# define velocity_limit 2

/* Alert DLOG indices */
# define Yes            1
# define No             3
# define UnsaveFile    1
# define whichFile      4
# define changeDrive    5
# define EjectDrive     6

```

```

#define whichDrive      7
#define contAnyway      1

/* DLOG IDs */
#define robotCSpaceID   258
#define robotJSpaceID   259
#define createCPathID   260
#define alterPathID      261
#define choosePathID     262
#define SetUpCartID      264
#define createJPathID     265
#define SetUpJointID      266
#define GraphicsID       267

/* CNTL IDs */
#define vScrollBar      128
#define hScrollBar      129

/* Robotics Dialog indices */
#define GO               1
#define STOP             2
#define setUpRun         3
#define saveRun          4
#define createPath       5
#define alterPath        6
#define choosePath       7
#define realMode         8
#define simulationMode    9
#define graphicsOptions 10

/* CreateCartesianPath DLOG indices */
#define pathName         3      /* also used for joint paths */
#define HowManyNodes     4      /* also used for joint paths */
#define AlwaysSameTime   5
#define theAlwysTime     7
#define AlwaysSameAcc    8
#define theAlwysAcc      10
#define X1               12
#define X2               13
#define X3               14
#define Y1               15
#define Y2               16
#define Y3               17
#define Z1               18
#define Z2               19
#define Z3               20
#define P1               21
#define P2               22
#define P3               23
#define theMoveTime      24
#define theAcc           25
#define theCNode         28
#define nodeOK           31
#define nodeCancel       32

```

```

/* CreateJointPath DLOG indices */
#define CurrentNode      5
#define jointAngle1      6
#define jointAngle2      7
#define jointAngle3      8
#define jointAngle4      9
#define jointAngle5     10
#define jointAngle6     11
#define jointAngle7     12
#define jointAccFac      13
#define jointnodeOK      14

/* AlterPath DLOG indices */
#define deleteNode       5
#define addNode          6
#define changeNode       7
#define theANode         9
#define doChange         10

/* ChoosePath and alternate AlterPath DLOG indices */
#define PathName         3

/* Setup Cart run DLOG indices */
#define perfCrit         3
#define deltaT           4 /* also used for joint setup */
#define Param3           5
#define Param4           6
#define When             7
#define startAngle1      8
#define startAngle2      9
#define startAngle3     10
#define startAngle4     11
#define startAngle5     12
#define startAngle6     13
#define startAngle7     14
#define Graphics         15
#define Numbers          16
#define leastNorm        17
#define Optimized        18
#define HomePos          19
#define SpecifyDegrees   20

/* Setup Joint run DLOG indices */
#define PerCentMaxSpd    3

/* Graphics Option DLOG indices */
#define TopView          3
#define FrontView        4
#define AllViews         5

```



```

#define SideView      6
#define showTitles    7
#define showHome      8
#define sizeImage     9
#define vertTop       10
#define horTop        11
#define vertFront     12
#define horFront      13
#define vertSide      14
#define horSide       15

```

```

/* File menu indices */
#define newFile      1
#define openFile     2
#define closeFile    3
#define saveFile     5
#define saveAsFile   6
#define unSaveFile   7
#define pageSetup    9
#define printFile    10
#define quitFile     12

```

```

/* Edit menu indices */
#define undoCommand  1
#define cutCommand   2
#define copyCommand  3
#define pasteCommand 4
#define clearCommand 5

```

```

/* Menu Bar indices */
#define FileMenu      0
#define EditMenu      1
#define TeleRobMenu   2
#define RoboticsMenu  3
#define ModeMenu      4
#define PathMenu      5
#define SetupMenu     6

```

```

/* Robotics Menu indices */
#define cartesianSpace 1
#define jointSpace    2

```

```

#define fileNew      1
#define fileOpen     2
#define fileClose    4
#define fileSave     5
#define fileSaveAs   6
#define fileRevert   7
#define filePageSetUp 9
#define filePrint    10

```

```
#define fileQuit      13
```

```
#define aaSave        1
```

```
#define aaDiscard     2
```

```
#define aaCancel      3
```

```
#define TOP_OF_SCREEN 20
```

```
/* global_data.h header file */
```

```
struct vector        {  
    double i;  
    double j;  
    double k;  
};
```

```
struct joint_path    {  
    float  J1;  
    float  J2;  
    float  J3;  
    float  J4;  
    float  J5;  
    float  J6;  
    float  J7;  
};
```

```
struct mot_data      {  
    short  pos;  
    short  vel;  
    short  torq;  
    char   temp;  
};
```

```
struct j_cmd         {  
    char   mode;  
    int    brakes;  
};
```

```
struct j_data        {  
    char   mode_status;  
    char   brakes_status;  
    char   gen_status;  
    double pitch_pos;  
    short  pitch_vel;  
    double yaw_pos;  
    short  yaw_vel;  
    struct mot_data      motor1;  
    struct mot_data      motor2;  
    char   logic_brd_temp;  
    char   pwr_brd_temp;  
};
```

```

struct hand_data
{
    int    perf_crit;
    float  move_time;
    float  accFactor;
    int    LastNode;
    struct vector      pos;
    struct vector      sliding;
    struct vector      normal;
    struct vector      approach;
};

```

```

struct wr_data
{
    double pos;
    short  vel;
    char   temp;
};

```

```

struct j_setpnts
{
    double pitch_pos;
    double yaw_pos;
    double   roll;
    double pitch_vel;
    double yaw_vel;
    short  mot1_drive;
    short  mot2_drive;
};

```

```

struct j_info
{
    struct j_cmd      cmd;
    struct j_data      data;
    short  mot1_cur;
    short  mot2_cur;
    struct j_setpnts  setpnts;
    /* gain,lim,bal,etc. arrays*/
};

```

```

struct wr_info
{
    struct j_cmd      cmd;
    struct j_cmd      roll_cmd;
    struct j_data      data;
    struct wr_data      roll_data;
    short  mot1_cur;
    short  mot2_cur;
    short  roll_cur;
    struct j_setpnts  setpnts;
    /* gain,lim,bal,etc. arrays*/
};

```

```

struct grip_data
{
    char  trig;
    char  joy_x;
    char  joy_y;
}

```

```

        char    buttons;
    };

    struct master_data    {
        struct j_info      shld;
        struct j_info      elbow;
        struct wr_info      wrist;
        struct grip_data    grip;
        struct hand_data    hand;
    };

    struct slave_data    {
        struct j_info      shld;
        struct j_info      elbow;
        struct wr_info      wrist;
        struct grip_data    grip;
        struct hand_data    hand;
        struct joint_path    joint;
        short dumbEE_cur;
        short dumbEE_drive;
        /* struct smartEE_data    smartEE; */
    };

    struct flag_data    {
        short    robotics;
        short    load_next_pos;
        short    robotics_go;
        short    simulate;
        int    jointLimit;
        int    VelLimit;
        char    start_proc;
        char    ap1_done;
        char    ap2_done;
        /* others as required */
    };

    struct arm_data    {
        struct master_data    m;
        struct slave_data    s;
    };

    struct global_data    {
        struct flag_data    flags;
        struct arm_data    l;
        struct arm_data    r;
    };

```

Appendix I

Graphics Module (Graphics.c)

```
#include "QuickDraw.h"
#include "stdio.h"
#include "MacTypes.h"
#include "LTM.h"
#include "global_data.h"
#include <unix.h>
#include <math.h>

extern int iteration, drawcount, view_case;
extern FILE *theAngles, *theCricket, *thePos;
extern JOINT_ANGLES *Angle;
extern struct ViewData ViewArm, ViewArm0;
extern float scale, delta_t;
extern struct global_data ltm, *LTM;
float H; /* the performance index */

/***** **The structure Angle is filled up in the function GetNextPosition() *****/
FillResultFiles(theView)
int theView; {
float RtoD;
RtoD = 180/PI;
EvalPerfCrit();
fprintf(theAngles, "%d\t%f\t%f\t%f\t", theView, RtoD*Angle->one,
RtoD*Angle->two, RtoD*Angle->three);
fprintf(theAngles, "%f\t%f\t%f\t", RtoD*Angle->four, RtoD*Angle->five,
RtoD*Angle->six);
fprintf(theAngles, "%f\t%f\n", RtoD*Angle->seven, H);
fprintf(thePos, "%d\n", theView);
fprintf(thePos, "%f %f %f\n", ViewArm.elbow.x, ViewArm.elbow.y,
ViewArm.elbow.z);
fprintf(thePos, "%f %f
%f\n", ViewArm.wrist.x, ViewArm.wrist.y, ViewArm.wrist.z);
fprintf(thePos, "%f %f %f\n", ViewArm.hand.x, ViewArm.hand.y, ViewArm.hand.z);
fprintf(thePos, "%f %f %f\n\n", ViewArm.orient.x, ViewArm.orient.y,
ViewArm.orient.z);}

EndFunction

/*****this draws the top view*****/
top_view (e1,e3,w1,w3,h1,h3,g,top,scale)
double e1,e3,w1,w3,h1,h3; /*elbow, wrist, and hand coordinates for top view*/
struct VECTOR g; /* unit vector in direction of hand x-axis */
struct VECTOR top; /* location of shoulder */
float scale; { /* how much to scale image */

267
```

```

int          int_e1,int_e3,int_w1,int_w3,int_h1,int_h3,int_gx,int_gz;
/**** top view screen conversions ****/
    e1 = top.x + scale*e1;
    e3 = top.y + scale*e3;
    w1 = top.x + scale*w1;
    w3 = top.y + scale*w3;
    h1 = top.x + scale*h1;
    h3 = top.y + scale*h3;
    g.x = top.x + scale*g.x;
    g.z = top.y + scale*g.z;
/****changing into an int****/
    int_e1 = (int) (e1 + .5);
    int_e3 = (int) (e3 + .5);
    int_w1 = (int) (w1 + .5);
    int_w3 = (int) (w3 + .5);
    int_h1 = (int) (h1 + .5);
    int_h3 = (int) (h3 + .5);
    int_gx = (int) (g.x + .5);
    int_gz = (int) (g.z + .5);
/***** drawing the image*****/
    MoveTo((int) top.x,(int) top.y);
    LineTo (int_e1,int_e3);
    LineTo (int_w1,int_w3);
    LineTo (int_h1,int_h3);
    LineTo ( int_gx,int_gz);}
EndFunction

```

```

/*****this draws the front view*****/
front_view (e1,e2,w1,w2,h1,h2,g,front,scale)
double      e1,e2,w1,w2,h1,h2;    /*elbow, wrist, and hand coord for front view*/
struct VECTOR g;                  /* unit vector in direction of hand x-axis */
struct VECTOR front;              /* location of shoulder */
float       scale; {              /* how much to scale image */
int          int_e1,int_e2,int_w1,int_w2,int_h1,int_h2,int_gx,int_gy;

/****front view screen conversions****/
    e1 = front.x + scale*e1;
    e2 = front.y - scale*e2;
    w1 = front.x + scale*w1;
    w2 = front.y - scale*w2;
    h1 = front.x + scale*h1;
    h2 = front.y - scale*h2;
    g.x = front.x + scale*g.x;
    g.y = front.y - scale*g.y;
/****changing to an int****/
    int_e1 = (int) (e1 + .5);
    int_e2 = (int) (e2 + .5);
    int_w1 = (int) (w1 + .5);
    int_w2 = (int) (w2 + .5);
    int_h1 = (int) (h1 + .5);
    int_h2 = (int) (h2 + .5);
    int_gx = (int) (g.x + .5);

```

```

    int_gy = (int) (g.y + .5);
/****drawing the image*****/
    MoveTo((int) front.x,(int) front.y);
    LineTo (int_e1,int_e2);
    LineTo (int_w1,int_w2);
    LineTo (int_h1,int_h2);
    LineTo ( int_gx,int_gy); }
EndFunction

/*****this draws the side view*****/
    side_view (e2,e3,w2,w3,h2,h3,g,side,scale)
double      e2,e3,w2,w3,h2,h3;    /*elbow, wrist, and hand coord for side view*/
struct VECTOR g;                  /* unit vector in direction of hand x-axis */
struct VECTOR side;               /* location of shoulder */
float       scale; {              /* how much to scale image */
int         int_e2,int_e3,int_w2,int_w3,int_h2,int_h3,int_gy,int_gz;
/****side view screen conversions****/
    e2 = side.y - scale*e2;
    e3 = side.x - scale*e3;
    w2 = side.y - scale*w2;
    w3 = side.x - scale*w3;
    h2 = side.y - scale*h2;
    h3 = side.x - scale*h3;
    g.y = side.y - scale*g.y;
    g.z = side.x - scale*g.z;
/****changing to an int*****/
    int_e2 = (int) (e2 + .5);
    int_e3 = (int) (e3 + .5);
    int_w2 = (int) (w2 + .5);
    int_w3 = (int) (w3 + .5);
    int_h2 = (int) (h2 + .5);
    int_h3 = (int) (h3 + .5);
    int_gy = (int) (g.y + .5);
    int_gz = (int) (g.z + .5);
/****drawing the image*****/
    MoveTo((int) side.x,(int) side.y);
    LineTo (int_e3,int_e2);
    LineTo (int_w3,int_w2);
    LineTo (int_h3,int_h2);
    LineTo ( int_gz,int_gy);}
EndFunction

```

```

DrawArm (A,view,when_draw)
struct    ViewData  A;
struct    views     view;
int       when_draw;{
static int  theView;
static float theTime = 0;
float      RtoD;
if ( iteration == 0)
{
    eraseplot ();
    drawcount = when_draw;

```

```

    theTime = 0.0;
}
else    ++drawcount;
if (drawcount == when_draw){
    theView = 1;
    FillResultFiles(theView);
    EvalPerfCrit();
    RtoD = 180/PI;
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle->one,RtoD*Angle->two,RtoD*Angle-
>three);
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle->four,RtoD*Angle->five,RtoD*Angle-
>six);
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle-
>seven,ViewArm.hand.x,ViewArm.hand.y);
    fprintf(theCricket,"%f\t%f\t%f\t",ViewArm.hand.z,H,theTime);
    theView = 0;
    drawcount = 0;
    switch (view_case)
    {
        case 0:
            top_view
(A.elbow.x,A.elbow.z,A.wrist.x,A.wrist.z,A.hand.x,A.hand.z,A.orient,view.top,scale);
            break;
        case 1:
            front_view
(A.elbow.x,A.elbow.y,A.wrist.x,A.wrist.y,A.hand.x,A.hand.y,A.orient,view.front,scale);
            break;
        case 2:
            side_view
(A.elbow.y,A.elbow.z,A.wrist.y,A.wrist.z,A.hand.y,A.hand.z,A.orient,view.side,scale);
            break;
        case 3:

            front_view(A.elbow.x,A.elbow.y,A.wrist.x,A.wrist.y,A.hand.x,A.hand.y,A.orient,view.
front,scale);

            top_view(A.elbow.x,A.elbow.z,A.wrist.x,A.wrist.z,A.hand.x,A.hand.z,A.orient,view.to
p,scale);

            side_view(A.elbow.y,A.elbow.z,A.wrist.y,A.wrist.z,A.hand.y,A.hand.z,A.orient,view.s
ide,scale);
            break;}}
else    {
    RtoD = 180/PI;
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle->one,RtoD*Angle->two,RtoD*Angle-
>three);
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle->four,RtoD*Angle->five,RtoD*Angle-
>six);
    fprintf(theCricket,"%f\t%f\t%f\t",RtoD*Angle-
>seven,ViewArm.hand.x,ViewArm.hand.y);
    fprintf(theCricket,"%f\t%f\t%f\t",ViewArm.hand.z,H,theTime);
    FillResultFiles(theView);}
theTime += delta_t;
EndFunction

```



```

/*****

```

When the option to draw the anthropomorphic stance of arm is selected in the Graphics Option dialog, this routine draws that image.

```

*****/

```

```

PictArm (){

```

```

int      x = 300,y = 20; /* Macintosh screen coordinate off-set. */

```

```

int      S = 2;          /* scaling factor */

```

```

int      x1,y1,x2,y2;

```

```

static float V[2][150] = /* vertices of the arm picture */

```

```

{ {0,0,0,3,3,19,17,17,18,0,20,0,20,21,21,23,22.5,20,11.5,8,
  8,13,17,18,18.5,18,18,17.5,17,15,18,21,21.5,21.5,18,19,20,
  20.5,20.5,19.5,19,18.5,18.5,18,17,0,16.5,17,18,19.5,19.5,
  19.5,20,21,22,22,0,22,22,21,21.5,18,15,15,15.5,16.5,17,18,
  18,18,21.5,21.5,23.5,20,20,20,19,17,19,20,27,28,28.5,28.5,
  28.5,27.5,27,25,28,30.5,31.5,32,32,31,30.5,30,30,30,0,29,
  27.5,29,31,33,33,33.5,34,34.5,36,37,37,40,40,37,0,35,37.5,
  37.5,34,34,33.5,33.5,17,11,11.5,12,18.5,18.5,18,19,20.5,
  28.5,27.5,27.5,28,29,30.5,31,30.5}

```

```

{0,10,26,28,12,3,2,0,5,0,7,0,10,12,13,14,16.5,18,13,15,19,
  22,24,24,23.5,22.5,21.5,21,20,19,17,19,19.5,21.5,19.5,19,
  19.5,20,20.5,21,22,23,25,25.5,25,0,25.5,26,26,25.5,24,23,
  21.5,21,21.5,22,0,23,24,24.5,25,27,
  25,35.5,37,37.5,38,38,37.5,35,33,35,34,36,40,39,39.5,
  38.5,39,38,44,44,43.5,43,42,41,40,39,37,38.5,39.5,40,41,
  41.5,42,42.5,43.5,44.5,0,45,44.5,45.5,45,46,45,44,43.5,
  43,43,43.5,44,45.5,47,45.5,0,46,47,48.5,46.5,45.5,45.5,
  46.5,22,26,25,26,20.5,11.5,13,13,22.5,27,
  27,26,39.5,39,39.5,41,41.5}

```

```

static int L[2][150] = /* lines of the arm picture */

```

```

{ {1,1,1,2,3,3,4,5,5,6,8,10,12,13,13,15,16,17,17,18,19,20,
  21,22,23,24,25,26,27,28,28,29,30,31,32,60,60,61,61,62,
  62,63,64,65,66,67,68,69,70,72,72,73,75,76,77,66,78,74,
  80,81,82,83,84,85,86,87,89,90,91,92,93,94,95,96,97,99,
  99,80,100,101,102,103,104,105,106,107,108,104,109,110,
  111,112,113,110,116,116,117,118,119,121,122,123,123,
  126,127,127,130,131,131,92,34,35,36,37,38,39,40,41,42,
  43,44,23,46,47,48,49,49,51,52,53,54,55,57,58,59,19,
  38,14,86,88,135,136,137,0}

```

```

{2,4,7,3,4,21,5,6,8,7,10,12,13,14,15,16,17,18,31,19,20,
  21,22,23,24,25,26,27,28,29,34,30,34,32,33,61,70,62,69,
  22,63,64,65,66,67,68,69,70,71,73,89,74,76,77,66,78,79,
  80,81,82,83,84,85,86,73,88,90,91,92,93,94,95,96,97,99,
  99,81,100,101,102,103,104,105,106,107,108,109,108,110,
  111,112,113,115,119,117,119,118,121,120,103,123,124,
  125,127,128,129,131,132,133,108,35,36,37,38,39,40,41,
  42,43,44,22,46,47,48,49,51,58,52,53,54,55,57,58,59,60,
  29,54,15,135,134,136,137,138,0}

```

```

int      *pL1,*pL2;

```

```

pL1 = &L[0][0];
pL2 = &L[1][0];
x1 = ((int) S*V[0][*pL1]) + x;
y1 = ((int) S*V[1][*pL1]) + y;
while (*pL1 != 0){
    x1 = ((int) S*V[0][*pL1]) + x;
    y1 = ((int) S*V[1][*pL1]) + y;
    x2 = ((int) S*V[0][*pL2]) + x;
    y2 = ((int) S*V[1][*pL2]) + y;
    MoveTo(x1,y1);
    LineTo (x2,y2);
    ++pL1;
    ++pL2;}
MoveTo (x+40,y+20);
DrawString ("pTop");
MoveTo (x-10,y+65);
DrawString ("pFront");
MoveTo (x+50,y+50);
DrawString ("pSide");}
EndFunction

```

Titles (view)

```

struct views      view;{
int      Left,Top,Right,Bottom;
Rect      Oval;
    switch (view_case){
        case 0:
            Left = ((int) view.top.x) - 4;
            Top = ((int) view.top.y) - 4;
            Right = ((int) view.top.x) + 4;
            Bottom = ((int) view.top.y) + 4;
            SetRect(&Oval,Left,Top,Right,Bottom);
            FrameOval (&Oval);
            MoveTo (Left-20,Top - 20);
            DrawString ("pTop View");
            MoveTo (Left-20,Top - 10);
            DrawString ("p(left arm)");
            break;
        case 1:
            Left = ((int) view.front.x) - 4;
            Top = ((int) view.front.y) - 4;
            Right = ((int) view.front.x) + 4;
            Bottom = ((int) view.front.y) + 4;
            SetRect(&Oval,Left,Top,Right,Bottom);
            FrameOval (&Oval);
            MoveTo (Left-20,Top - 20);
            DrawString ("pFront View");
            MoveTo (Left-40,Top - 10);
            DrawString ("p(anthropomorphic side of left arm)");
            break;
        case 2:
            Left = ((int) view.side.x) - 4;

```

```

    Top = ((int) view.side.y) - 4;
    Right = ((int) view.side.x) + 4;
    Bottom = ((int) view.side.y) + 4;
    SetRect(&Oval,Left,Top,Right,Bottom);
    FrameOval (&Oval);
    MoveTo (Left-20,Top - 20);
    DrawString ("pSide View");
    MoveTo (Left-20,Top - 10);
    DrawString ("\p(anthropomorphic front of left arm)");
    break;
case 3:
    Left = ((int) view.top.x) - 4;
    Top = ((int) view.top.y) - 4;
    Right = ((int) view.top.x) + 4;
    Bottom = ((int) view.top.y) + 4;
    SetRect(&Oval,Left,Top,Right,Bottom);
    FrameOval (&Oval);
    MoveTo (Left-20,Top - 10);
    DrawString ("\pTop View");
    Left = ((int) view.front.x) - 4;
    Top = ((int) view.front.y) - 4;
    Right = ((int) view.front.x) + 4;
    Bottom = ((int) view.front.y) + 4;
    SetRect(&Oval,Left,Top,Right,Bottom);
    FrameOval (&Oval);
    MoveTo (Left-20,Top + 100);
    DrawString ("\pFront View");
    Left = ((int) view.side.x) - 4;
    Top = ((int) view.side.y) - 4;
    Right = ((int) view.side.x) + 4;
    Bottom = ((int) view.side.y) + 4;
    SetRect(&Oval,Left,Top,Right,Bottom);
    FrameOval (&Oval);
    MoveTo (Left-20,Top + 100);
    DrawString ("\pSide View");
    break;}}

```

EndFunction

```

EvalPerfCrit() {
JOINT_ANGLES Cos,Sin;
int CaseNum;
CaseNum = LTM->l.s.hand.perf_crit;
TrigValues(&Cos,&Sin,*Angle);
switch (CaseNum){
    case 1: /* SINGULARITIES
        Joint limit using  $H = \sin(\theta_2)^2 + \cos(\theta_3)^2 + \sin(\theta_4)^2 + \cos(\theta_6)^2$ 
        */
        H = Cos.three*Cos.three + Cos.four*Cos.four;
        break;
    case 2: /* JOINT LIMITS */
        H = (Angle->five - 0.785398)*(Angle->five - 0.785398)

```

```

        +(Angle->three - 0.785398)*(Angle->three - 0.785398);
    break;
case 3: /*JOINT LIMITS */
    H = (Angle->five - 0.785398)*(Angle->five - 0.785398);
    break;}}
EndFunction

```

Appendix J

Joint Space Module (LTM_JointSpace.c)

```
#include "unix.h"
#include "global_data.h"
#include "LTM.h"
#include "MacTypes.h"
#include "QuickDraw.h"
#include <stdio.h>
#include <math.h>
```

```
/* declaration of globals */
extern struct global_data ltm,*LTM;
extern struct ViewData ViewArm,ViewArm0;
extern float delta_t;
extern JOINT_ANGLES *Angle;
extern FILE *thePromptW;
extern JOINT_ANGLES dTMax;
extern JOINT_ANGLES MaxSpeed;
extern float ThetaInit[8];
```

```
/* functions defined on other modules */
VelocityLimit();
JointLimit();
struct ViewData ForwardPosKinematics();
```

```
/*
the cartesian space counterpart to GetJointConfig() is GetDesiredPosition(). The scheme to
smoothly change a coordinate applies to joint space too. Just substitute joint angles for the
generalized coordinate.
*/
```

```
GetJointConfig () {
static COUNTER i;
static int TheLastNode,firstIteration;
static JOINT_ANGLES NULL_STRUCT = {0,0,0,0,0,0};
static JOINT_ANGLES Cos,Sin,theta;
static MATRIX P; /* actually a structure of type MATRIX */
float t; /* total move time between current points */
double temp; /* temporary variable */
static int nt; /* number of steps in move time t */
static int na; /* number of steps in acceleration time */
static int two_na; /* two times na */
static int casenum = 0; /* case for switch statement */
```

```
/****** create structured variables *****/
```

Most of these variable names correspond to those used by Richard P. Paul on page 135 of ROBOT MANIPULATORS TS191.8.P38, for more detailed information consult Chapter 5, Section 5.7.

```
*****/
```

```

static JOINT_ANGLES      JA; /* configuration at which acc begins */
static JOINT_ANGLES      JB; /* configuration that will be approached and
passed */
static JOINT_ANGLES      JC; /* the next configuration */
JOINT_ANGLES            JD; /* another next configuration */
static JOINT_ANGLES      delta_JB; /* JA-JB */
static JOINT_ANGLES      delta_JC; /* JC-JB */
static HOMOGEN_TRANS      current_pos; /* the current pos and orient */
static JOINT_ANGLES      f1,f2;
static JOINT_ANGLES      desiredJointAngles;
/***** end of create structured variables *****/
/***** create pointers *****/
JOINT_ANGLES            *pJA,*pJB,*pJC;
pJA = &JA;
pJB = &JB;
pJC = &JC;
Angle = &theta;
/***** main body of GetJointConfig *****/
switch (casenum) {
    case 0:
        casenum = 1;
        TheLastNode = LTM->l.s.hand.LastNode;
        NextJointNode(&JC); /* Assigns next jointconfiguration to JC */
        GetInitialAngles (&theta);
        TrigValues (&Cos,&Sin,theta);
        ProjectionElements (Cos,Sin,&P);
        ViewArm0 = ForwardPosKinematics (&current_pos,Cos,Sin,P);
        JA = theta;
        JB = JA;
        delta_JB = NULL_STRUCT; /* NULL_STRUCT = structure with all elements = 0
*/
        SubtractVectors2 (pJC,pJB,&delta_JC);
        GetTimeSegment(&t,JB,JC);
        nt = t/delta_t + .5;
        na = nt*(LTM->l.s.hand.accFactor); /* number of steps in na */
        i = -na;
        JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
        JointAcc (JA,na,two_na,f1,f2,&theta,i);
        firstIteration = 1;
        break;
    case 1:
        if ( i >= na+1 AND i < nt-na )
            ConstJointVel(pJB,nt,delta_JC,&theta,i);
        else
            JointAcc (JA,na,two_na,f1,f2,&theta,i);
        break;
    case 2:
        JointAcc (JA,na,two_na,f1,f2,&theta,i);
        if (i == na) {
            LTM->flags.robotics = 0; /* the last position has been reached*/
            casenum = 0; }
        break; }
if(JointVelCheck(&firstIteration,&theta)) {
    VelocityLimit();

```

```

    LTM->flags.robotics = 0; /* tell arm to stop */
    LTM->flags.VelLimit = 1;
    casenum = 0; }
TrigValues (&Cos,&Sin,theta);
ProjectionElements (Cos,Sin,&P);
ViewArm = ForwardPosKinematics(&current_pos,Cos,Sin,P);
JointLimit (&casenum,theta);
if (!LTM->flags.simulate)
    SendCommand (theta);
++i;
if ((i == nt-(na+1)) AND !TheLastNode)
    LTM->flags.load_next_pos = 1;
if (i == nt-na AND TheLastNode) {
    casenum = 2;
    JA = theta;
    JB = JC;
    SubtractVectors2(pJA,pJB,&delta_JB);
    delta_JC = NULL_STRUCT;
    JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
    i = -na; }
else if (i == nt-na) {
    NextJointNode(&JD);
    GetTimeSegment(&t,JC,JD);
    JA = theta;
    JB = JC;
    JC = JD;
    SubtractVectors2(pJA,pJB,&delta_JB);
    SubtractVectors2 (pJC,pJB,&delta_JC);
    nt = t/delta_t + .5;
    na = nt*(LTM->l.s.hand.accFactor); /* number of steps in na */
    TheLastNode = LTM->l.s.hand.LastNode;
    if (TheLastNode) nt = nt - na;
    JointAccSetup (na,nt,&two_na,delta_JC,delta_JB,&f1,&f2);
    i = -na;}}
EndFunction

```

/******calculates constants needed by acceleration function******/

```

JointAccSetup (na,nt,two_na,delta_C,delta_B,f1,f2)
int          na,nt,*two_na;
JOINT_ANGLES delta_B,delta_C,*f1,*f2;{
double
    *two_na      = na + na;
    temp         = ((double)na)/((double) nt);
    f1->one       = (delta_C.one * temp) + delta_B.one;
    f1->two       = (delta_C.two * temp) + delta_B.two;
    f1->three     = (delta_C.three * temp) + delta_B.three;
    f1->four      = (delta_C.four * temp) + delta_B.four;
    f1->five      = (delta_C.five * temp) + delta_B.five;
    f1->six       = (delta_C.six * temp) + delta_B.six;
    f1->seven     = (delta_C.seven * temp) + delta_B.seven;
    f2->one       = delta_B.one + delta_B.one;
    f2->two       = delta_B.two + delta_B.two;
    f2->three     = delta_B.three + delta_B.three;

```

```

f2->four      = delta_B.four + delta_B.four;
f2->five      = delta_B.five + delta_B.five;
f2->six       = delta_B.six + delta_B.six;
f2->seven     = delta_B.seven + delta_B.seven;}
EndFunction

```

```

JointAcc (A,na,two_na,f1,f2,desiredAngles,i)
JOINT_ANGLES A,f1,f2,*desiredAngles;
int          na,two_na,i; {
double       h,h2,h3,two_h2,twoh2_h3;
  h = ( (double)i + (double)na)/((double)two_na);
  h2 = h*h;
  h3 = h2*h;
  two_h2 = h2 + h2;
  twoh2_h3 = two_h2 - h3;
  desiredAngles->one = (f1.one*twoh2_h3 - f2.one)*h + A.one;
  desiredAngles->two = (f1.two*twoh2_h3 - f2.two)*h + A.two;
  desiredAngles->three = (f1.three*twoh2_h3 - f2.three)*h + A.three;
  desiredAngles->four = (f1.four*twoh2_h3 - f2.four)*h + A.four;
  desiredAngles->five = (f1.five*twoh2_h3 - f2.five)*h + A.five;
  desiredAngles->six = (f1.six*twoh2_h3 - f2.six)*h + A.six;
  desiredAngles->seven = (f1.seven*twoh2_h3 - f2.seven)*h + A.seven;}
EndFunction

```

```

ConstJointVel (B,nt,delta_C,desiredAngles,i)
JOINT_ANGLES *B,delta_C,*desiredAngles;
int          nt,i;{
double       h;
  h = ((double)i)/((double)nt);
  desiredAngles->one = (delta_C.one * h) + B->one;
  desiredAngles->two = (delta_C.two * h) + B->two;
  desiredAngles->three = (delta_C.three * h) + B->three;
  desiredAngles->four = (delta_C.four * h) + B->four;
  desiredAngles->five = (delta_C.five * h) + B->five;
  desiredAngles->six = (delta_C.six * h) + B->six;
  desiredAngles->seven = (delta_C.seven * h) + B->seven;}
EndFunction

```

```

NextJointNode(JD)
JOINT_ANGLES *JD;{
  JD->one = LTM->l.s.joint.J1;
  JD->two = LTM->l.s.joint.J2;
  JD->three = LTM->l.s.joint.J3;
  JD->four = LTM->l.s.joint.J4;
  JD->five = LTM->l.s.joint.J5;
  JD->six = LTM->l.s.joint.J6;
  JD->seven = LTM->l.s.joint.J7;}
EndFunction

```

```

/*****

```


This calculates the time between nodes to give GetJointConfig(). It depends on the maximum speed of the slowest joint motor. It also depends on what was specified as maximum speed in the dialog Set Up Run Features (for Joint Space).

```

*****/
GetTimeSegment(T,JC,JD)
float      *T;
JOINT_ANGLES JC,JD;{
JOINT_ANGLES t;
    t.one   = (fabs(JD.one - JC.one))/MaxSpeed.one;
    t.two   = (fabs(JD.two - JC.two))/MaxSpeed.two;
    t.three = (fabs(JD.three - JC.three))/MaxSpeed.three;
    t.four  = (fabs(JD.four - JC.four))/MaxSpeed.four;
    t.five  = (fabs(JD.five - JC.five))/MaxSpeed.five;
    t.six   = (fabs(JD.six - JC.six))/MaxSpeed.six;
    t.seven = (fabs(JD.seven - JC.seven))/MaxSpeed.seven;
    *T = (float) (max(t.one,t.two));
    *T = (float) (max(t.three,*T));
    *T = (float) (max(t.four,*T));
    *T = (float) (max(t.five,*T));
    *T = (float) (max(t.six,*T));
    *T = (float) (max(t.seven,*T));}
EndFunction

```

```

SubtractVectors2 (c,b,delta)
JOINT_ANGLES *c,*b,*delta;{
    delta->one = c->one - b->one;
    delta->two = c->two - b->two;
    delta->three = c->three - b->three;
    delta->four = c->four - b->four;
    delta->five = c->five - b->five;
    delta->six = c->six - b->six;
    delta->seven = c->seven - b->seven;}
EndFunction

```

```

int JointVelCheck(firstIteration,Angles)
int      *firstIteration;
JOINT_ANGLES *Angles;{
static JOINT_ANGLES dTheta,oldAngles;
int      velerr = 0;
    if(*firstIteration){
        oldAngles.one = ThetaInit[1];
        oldAngles.two = ThetaInit[2];
        oldAngles.three = ThetaInit[3];
        oldAngles.four = ThetaInit[4];
        oldAngles.five = ThetaInit[5];
        oldAngles.six = ThetaInit[6];
        oldAngles.seven = ThetaInit[7];
        *firstIteration = 0;}
    SubtractVectors2(Angles,&oldAngles,&dTheta);
    oldAngles = *Angles;
    if (fabs(dTheta.one) > dTMax.one )

```

```
        return(1);
    else if (fabs(dTheta.two) > dTMax.two )
        return(1);
    else if (fabs(dTheta.three) > dTMax.three )
        return(1);
    else if (fabs(dTheta.four) > dTMax.four )
        return(1);
    else if (fabs(dTheta.five) > dTMax.five )
        return(1);
    else if (fabs(dTheta.six) > dTMax.six )
        return(1);
    else if (fabs(dTheta.seven) > dTMax.seven )
        return(1);
    else
        return(0);}
EndFunction
```

Appendix K

Cartesian Space Module (LTM_CartSpace.c)

```

#include "unix.h"
#include "global_data.h"
#include "LTM.h"
#include "MacTypes.h"
#include "QuickDraw.h"
#include <stdio.h>
#include <math.h>

/* declaration of globals */
extern struct global_data ltm,*LTM;
extern int leastnorm;
extern float delta_t;
struct ViewData ViewArm,ViewArm0;
JOINT_ANGLES *Angle;
extern FILE *thePromptW;
extern JOINT_ANGLES dTMax;

/* functions defined on other module */
JointLimit();
VelocityLimit();
GetGradientH();

/***** info on ForwardPosKinematics function *****/
The ForwardPosKinematics function returns a structure of type
HOMOGEN_TRANS which is described in the file "global.h".
Basically the joint angles are obtained and used to compute the
orientation matrix and position vector of the hand which are
returned to the calling function with a structure of type HOMOGEN_TRANS
which is referenced to the third frame.
*****/
start of ForwardPosKinematics*****/
struct ViewData ForwardPosKinematics (c,Cos,Sin,P)
HOMOGEN_TRANS *c;
JOINT_ANGLES Cos,Sin;
MATRIX P;
{
double C4C5C6,S4S6,C4S5,S4C5C6_plus_C4S6,C4C5C6_minus_S4S6,S4S5,S5C6,C5S6;
double e1,e2,e3,w1,w2,w3; /* elbow and wrist coordinates */
struct ViewData view;
/***** hand orientation and position vectors referenced to 3rd frame *****/

/* preliminary multiplications */

C4C5C6_minus_S4S6 = (Cos.four * Cos.five * Cos.six)-(Sin.four * Sin.six);
C4S5 = Cos.four * Sin.five;
S4C5C6_plus_C4S6 = (Sin.four * Cos.five * Cos.six) + (Cos.four * Sin.six);
S4S5 = Sin.four * Sin.five;

```

```

S5C6 = Sin.five * Cos.six;
C5S6 = Cos.five * Sin.six;

/* hand orientation vectors referred to the 3rd frame */

c->s1 = (C4C5C6_minus_S4S6)*Cos.seven + C4S5*Sin.seven ;
c->s2 = (S4C5C6_plus_C4S6)*Cos.seven + S4S5*Sin.seven ;
c->s3 = Cos.five*Sin.seven - S5C6*Cos.seven ;

c->n1 = -(C4C5C6_minus_S4S6)*Sin.seven + C4S5*Cos.seven ;
c->n2 = -(S4C5C6_plus_C4S6)*Sin.seven + S4S5*Cos.seven ;
c->n3 = Cos.five*Cos.seven + S5C6*Sin.seven ;

c->a1 = Sin.four*Cos.six + Cos.four*C5S6 ;
c->a2 = Sin.four*C5S6 - Cos.four*Cos.six ;
c->a3 = -Sin.five*Sin.six ;

/* elbow coordinates referred to frame 3 */

e1 = Constant_a2 * Cos.three;
e2 = 0;
e3 = Constant_a2 * Sin.three;

/* wrist coordinates referred to frame 3 */

w1 = e1 + Constant_a4 * Cos.four;
w2 = Constant_a4 * Sin.four;
w3 = e3;

/* hand coordinate referred to the 3rd frame */

c->pos1 = w1 + Constant_d7 * c->a1;
c->pos2 = w2 + Constant_d7 * c->a2;
c->pos3 = w3 + Constant_d7 * c->a3;

ProjectVector (&c->pos1,&c->pos2,&c->pos3,P,0);
ProjectVector (&c->s1,&c->s2,&c->s3,P,0);
ProjectVector (&c->n1,&c->n2,&c->n3,P,0);
ProjectVector (&c->a1,&c->a2,&c->a3,P,0);

if (LTM->flags.simulate)
{
    ProjectVector (&e1,&e2,&e3,P,0);
    ProjectVector (&w1,&w2,&w3,P,0);

    view.elbow.x = e1;
    view.elbow.y = e2;
    view.elbow.z = e3;
    view.wrist.x = w1;
    view.wrist.y = w2;
    view.wrist.z = w3;
    view.hand.x = c->pos1;
    view.hand.y = c->pos2;
    view.hand.z = c->pos3;

```

```

view.orient.x = c->pos1 + c->s1;
view.orient.y = c->pos2 + c->s2;
view.orient.z = c->pos3 + c->s3;
return (view);}}

```

EndFunction

*****Info on motion between two points *****
 See ROBOT MANIPULATORS (TS191.8.P38),Chapter 5,section 5.7 by Richard P. Paul for more
 information.

 The algorithm for moving between points accepts a starting point P(0) and point
 P(1) to move to. If, at time t-ta, it has available a different point P(2), then it
 will smoothly accelerate by P(1) without actually reaching P(1). If, however, it
 is offered P(1) again, it will decelerate and stop at P(1). If it is desired that
 the manipulator move in a start-stop-start-stop fashion (i.e. it stops at each
 point in a path), then each point must be offered three times. Two times to stop
 at the point and one time to accelerate from that point to another new point.

In order to accelerate from a condition of zero velocity and acceleration, the
 position and orientation of the hand must be assigned to matrix A and also to
 matrix B. This is the one instance that the hand does reach B. Matrix C will
 always take the values of NextPosition and matrix B will take the values of the
 previous C matrix EXCEPT when the hand must stop at C, in which case B is assigned
 the values of the current C matrix (or assigned values of NextPosition the same as C).
 The diagram below illustrates this for a series of three points.

```

                                C1 = NextPosition = P(2) *****
                                C2 = C1 *****
                                B2 = C2 *****
                                *****
                                *****
                                A(2)**** <--- point deceleration starts
                                ****
Co = NextPosition = P(1)      ****
B1 = Co                      ****
                                ****
                                ****
                                A(1)** <--- start of next acceleration
                                **
                                **
                                **
                                **
                                ***
                                ****
                                *** P(0)
                                Ao = startposition = point acceleration starts
                                Bo = Ao

```

The algorithm was developed by Richard P. Paul using a 4th degree polynomial for the position and then differentiating it twice. He considered points A(1),B(1), C(1) above, where A(1) occurs at time t-ta. The times t and ta are the time to move from B(1) to C(1) and the time allowed for acceleration respectively. The acceleration time is actually 2*ta, where A(1) occurs at time -ta (in deriving the equations Paul set the time axis origin in such a way that A(1) occurs at time -ta, but in the general case it occurs at time t-ta) and acceleration stops at time +ta which allows for complete velocity reversals.

The boundary conditions are:

at time = -ta:

position = A(1)
velocity = -(delta_B)/ta
acceleration = 0

at time = ta:

position = not needed
velocity = (delta_C)/t
acceleration = 0

Note that there are five unknowns (the constants of the polynomial) and five independent equations. After solving for the constants and inserting them back into the polynomial and its derivatives the following are obtained:

$$q = [(\delta_C(ta/t) + \delta_B)(2 - h)^2 h^2 - 2\delta_B]h + A, \quad (1)$$

$$q_{\dot{}} = [(\delta_C(ta/t) + \delta_B)(1.5 - h)^2 h - \delta_B](1/ta), \quad (2)$$

$$q_{\ddot{}} = (\delta_C(ta/t)(1-h)(3h)/(ta^2 ta)), \quad (3)$$

where $h = (i + ta)/(2*na)$ and i has an initial time of -ta. At $i = ta$, h is 1 and equation (1) above becomes:

$$q(ta) = \delta_C(ta/t) + B.$$

At time ta + 1 we use:

$$q = \delta_C h + B,$$

$$q_{\dot{}} = \delta_C/t,$$

$$q_{\ddot{}} = 0,$$

where $h = i/t$.

In the code time is converted to time steps. That is

$na = ta/\delta_t$ ---> number of time steps in ta,

$nt = t/\delta_t$ ---> number of time steps in t,

where δ_t is the period of the systems clock.

The function that addresses motion between points is called GetDesiredPosition.

Now you know everything I know about it!

***** end of info on motion between two points *****/

```

/*****      start      of      GetDesiredPosition      *****/

GetDesiredPosition ()
{
    static COUNTER i;
    static int      PerfCrit;
    static int      TheLastNode;
    static HAND_POS_OR NULL_STRUCT = {0,0,0,0,0,0};
    static JOINT_ANGLES Cos,Sin,theta;
    static MATRIX    P; /* P stands for projection matrix, but it's
                           actually a structure of type MATRIX */

/* The following are peculiar to positioning only */
float      t; /* total move time between current points */
double     temp; /* temporary variable */
static int nt; /* number of steps in move time t */
static int na; /* number of steps in acceleration time */
static int two_na; /* two times na */
static int casenum = 0; /* case for switch statement */

/*****      create structured variables      *****/
Most of these variable names correspond to those used by Richard P. Paul on page 135 of
ROBOT MANIPULATORS TS191.8.P38, for more detailed information consult Chapter 5,
Section 5.7.
*****/

static HAND_POS_OR A; /* the pos and orient at which acc begins */
static HAND_POS_OR B; /* the pos and orient that will be approached and passed */
static HAND_POS_OR C; /* the next pos and orient that may be passed or stopped at */
static HAND_POS_OR delta_B; /* A-B */
static HAND_POS_OR delta_C; /* C-B */
static HAND_POS_OR f1,f2;
static HAND_POS_OR desired_pos; /* the next pos and orient between B and C */
static HOMOGEN_TRANS current_pos; /* the current pos and orient between B and C */
INCREMENTS w_err,h_err; /* wrist error, hand error */
JOINT_ANGLES dT; /* output of inverse kinematics function */
/*****      end      of      create structured variables      *****/

/*****      main      body      of      GetDesiredPosition      *****/

Angle = &theta;
switch (casenum){
    case 0:
        casenum = 1;
        PerfCrit = LTM->l.s.hand.perf_crit;
        TheLastNode = LTM->l.s.hand.LastNode;
        NextPosition (&C); /* Assigns next position to C */
        GetInitialAngles (&theta);
        TrigValues (&Cos,&Sin,theta);
        ProjectionElements (Cos,Sin,&P);
        ViewArm0 = ForwardPosKinematics (&current_pos,Cos,Sin,P);
        GetEulerAngles (current_pos,&A); /* set A = current position */

```

```

    B = A;
    delta_B = NULL_STRUCT; /* delta_B = structure with all elements = 0 */
    SubtractVectors1 (&C,&B,&delta_C);
    t = LTM->l.s.hand.move_time;
    nt = t/delta_t + .5;
    na = nt*LTM->l.s.hand.accFactor; /* number of steps in ta */
    if (TheLastNode) nt = nt - (na + na);
    else          nt = nt - na;
    i = -na;
    AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
    Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
    break;
case 1:
    if ( i >= na+1 AND i < nt-na )
        MaintainConstantVelocity (&B,nt,delta_C,&desired_pos,i);
    else
        Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
    break;
case 2:
    Accelerate (A,na,two_na,f1,f2,&desired_pos,i);
    if (i == na){
        LTM->flags.robotics = 0; /* the last position has been reached*/
        casenum = 0;
        break;}
    HandError (current_pos,desired_pos,&h_err);
    WristError (h_err,current_pos,&w_err);
    ProjectVector (&w_err.dx,&w_err.dy,&w_err.dz,P,3);
    ProjectVector (&w_err.dThetaX,&w_err.dThetaY,&w_err.dThetaZ,P,3);
    if(InverseKinematics (w_err,PerfCrit,&dT,Cos,Sin,theta) == 1){
        VelocityLimit();
        LTM->flags.robotics = 0; /* tell arm to stop */
        LTM->flags.VelLimit = 1;
        casenum = 0; }
    UpdateJointAngles (dT,&theta);
    TrigValues (&Cos,&Sin,theta);
    ProjectionElements (Cos,Sin,&P);
    ViewArm = ForwardPosKinematics (&current_pos,Cos,Sin,P);
    JointLimit (&casenum,theta);
    if (!LTM->flags.simulate)
        SendCommand (theta);
    ++i;
    if ((i == nt-(na+1)) AND !TheLastNode){
        LTM->flags.load_next_pos=1;}
    if (i == nt-na AND TheLastNode){
        casenum = 2;
        GetEulerAngles (current_pos,&A);
        B = C;
        if ((fabs(B.psi - A.psi)) > PI)
            A.psi = -A.psi;
        SubtractVectors1 (&A,&B,&delta_B);
        delta_C = NULL_STRUCT;
        AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
        i = -na;}
    else if (i == nt-na){

```



```

GetEulerAngles (current_pos,&A);
B = C;
if ((fabs(B.psi - A.psi)) > PI)
    A.psi = -A.psi;
NextPosition (&C);
SubtractVectors1 (&A,&B,&delta_B);
SubtractVectors1 (&C,&B,&delta_C);
t = LTM->l.s.hand.move_time;
nt = t/delta_t + .5;
na = nt*LTM->l.s.hand.accFactor; /* number of steps in ta */
PerfCrit = LTM->l.s.hand.perf_crit;
TheLastNode = LTM->l.s.hand.LastNode;
if (TheLastNode) nt = nt - na;
AccSetup (na,nt,&two_na,delta_C,delta_B,&f1,&f2);
i = -na;}}
EndFunction

```

/****** start of GetInitialAngles *****

This function returns the current joint angles from the arm encoders at time t = 0. After that current joint angles are taken to be the last calculated joint angles.

```

***** /
GetInitialAngles (theta)
JOINT_ANGLES *theta; {
    theta->one = LTM->l.s.shld.data.pitch_pos;
    theta->two = LTM->l.s.shld.data.yaw_pos;
    theta->three = LTM->l.s.elbow.data.pitch_pos;
    theta->four = LTM->l.s.elbow.data.yaw_pos;
    theta->five = LTM->l.s.wrist.data.pitch_pos;
    theta->six = LTM->l.s.wrist.data.yaw_pos;
    theta->seven = LTM->l.s.wrist.roll_data.pos; }
EndFunction

```

/****** Trigvalues *****/

```

TrigValues (Cos,Sin,theta)
JOINT_ANGLES *Cos,*Sin,theta;{
    Cos->one = cos(theta.one);
    Sin->one = sin(theta.one);
    Cos->two = cos(theta.two);
    Sin->two = sin(theta.two);
    Cos->three = cos(theta.three);
    Sin->three = sin(theta.three);
    Cos->four = cos(theta.four);
    Sin->four = sin(theta.four);
    Cos->five = cos(theta.five);
    Sin->five = sin(theta.five);
    Cos->six = cos(theta.six);
    Sin->six = sin(theta.six);
    Cos->seven = cos(theta.seven);
    Sin->seven = sin(theta.seven);}

```

EndFunction

/****** info on function "ProjectionElements" *****/

The function ProjectionElements fills a global structure of type MATRIX containing the elements of the projection matrix necessary to project vectors referenced to the third frame onto the base frame or vice-versa.

Both types of projection use the same projection elements.

ProjectionElements (Cos,Sin,P)

JOINT_ANGLES Cos,Sin;

MATRIX *P;{

double C1C2,S1C2;

C1C2 = Cos.one * Cos.two;

S1C2 = Sin.one * Cos.two;

P->e11 = (C1C2 * Cos.three) - (Sin.one * Sin.three);

P->e12 = Cos.one * Sin.two;

P->e13 = (C1C2 * Sin.three) + (Sin.one * Cos.three);

P->e21 = (S1C2 * Cos.three) + (Cos.one * Sin.three);

P->e22 = Sin.one * Sin.two;

P->e23 = (S1C2 * Sin.three) - (Cos.one * Cos.three);

P->e31 = -Sin.two * Cos.three;

P->e32 = Cos.two;

P->e33 = -Sin.two * Sin.three;}

EndFunction

/****** info on ProjectVector function *****/

The function ProjectVector transforms a vector expressed in the third reference frame into a vector referenced to the base reference frame or vice-versa depending on the value of the direction flag defined below.

***** /

ProjectVector (v1,v2,v3,P,direction)

double *v1,*v2,*v3; /* elements of vector to be projected */

MATRIX P; /* elements of projection matrix */

FLAG direction; /* direction = 0 projects vectors expressed in the 3rd frame of reference to the base frame.
direction = 3 projects vectors expressed in the base frame of reference to the 3rd frame. */{

double x,y,z; /* temporary storage of vector elements */

x = *v1;

y = *v2;

z = *v3;

if (direction EQUALS 0){

*v1 = (P.e11 * x) + (P.e12 * y) + (P.e13 * z);

*v2 = (P.e21 * x) + (P.e22 * y) + (P.e23 * z);

*v3 = (P.e31 * x) + (P.e32 * y) + (P.e33 * z);}

else if (direction EQUALS 3){

*v1 = (P.e11 * x) + (P.e21 * y) + (P.e31 * z);

*v2 = (P.e12 * x) + (P.e22 * y) + (P.e32 * z);

*v3 = (P.e13 * x) + (P.e23 * y) + (P.e33 * z);}}

EndFunction

/****** info on obtaining Euler angles *****

The Euler angles are obtained thusly:

$$T = \text{Rot}(z,\phi)\text{Rot}(y,\theta)\text{Rot}(z,\psi) \quad (1)$$

$$\{\text{Rot}(z,\phi)\text{inverse}\}T = \text{Rot}(y,\theta)\text{Rot}(z,\psi) \quad (2)$$

where $\text{Rot}(z,\phi)$ designates a rotation around the z-axis an angle ϕ , etc. and T is defined below, then, using Eq.(2) and equating element(2,3) of the left hand side to element(2,3) of the right hand side we obtain

$$-\sin_\phi T_{13} + \cos_\phi T_{23} = 0,$$

where T_{ij} is element(i,j) of the homogeneous matrix T which defines the position and orientation of the hand with reference to the base, and $\sin_\phi$ and $\cos_\phi$ are $\sin(\phi)$ and $\cos(\phi)$ respectively. Solving for ϕ we obtain:

$$\phi = \text{atan2}(T_{23}, T_{13})$$

and

$$\phi = \phi + 180 \text{ degrees.}$$

$$\theta = \text{atan2}(T_{13}\cos_\phi + T_{23}\sin_\phi, T_{33})$$

$$\psi = \text{atan2}(-T_{11}\sin_\phi + T_{21}\cos_\phi, -T_{12}\sin_\phi + T_{22}\cos_\phi)$$

For more on this see Paul, ROBOT MANIPULATORS, Chapter 3, Section 3.2

***** /

```

GetEulerAngles (hand,p)
HOMOGEN_TRANS      hand;
HAND_POS_OR        *p;{
double             ox,oy,nx,ny,ax,ay,az; /* agrees with Paul's notation */
double             cos_phi,sin_phi;
static double      last_ax,last_ay;

    nx = hand.s1;
    ny = hand.s2;
    ox = hand.n1;
    oy = hand.n2;
    ax = hand.a1;
    ay = hand.a2;
    az = hand.a3;
    if (ax == 0.0 AND ay == 0.0)
        p->phi = 0.0;
    else
        p->phi = atan2(ay,ax);
    cos_phi = cos (p->phi);
    sin_phi = sin (p->phi);
    p->theta = atan2(ax*cos_phi + ay*sin_phi, az);
    p->psi = atan2(-nx*sin_phi + ny*cos_phi, -ox*sin_phi + oy*cos_phi);

/* fill up x,y,z, elements of rhand */

```

```

    p->x = hand.pos1;
    p->y = hand.pos2;
    p->z = hand.pos3; }
EndFunction

```

```

double MyAtan2(y,x)
double y,x;{
double angle;
    if (y) {
        if (x)          angle = atan(y/x);
        else if (x<0) angle = atan(y/x) + PI;
        else            angle = PI/2; }
    else if (y<0) {
        if (x)          angle = atan(y/x);
        else if (x<0) angle = atan(y/x) - PI;
        else            angle = -PI/2; }
    else {
        if (x) angle = 0;
        else if (x<0) angle = -PI;
        else printf("ERROR: numerator and denominator both zero atan2()"); }
    return (angle);}
EndFunction

```

```

/***** info on NexNode function *****/
Inspection of LTM.h will reveal that HAND_POS_OR is a structure
defined to have six elements:x,y,z,phi,theta,and psi all of which are
of type double. The function "NextNode" will return a structure
of type HAND_POS_OR by placing the position of the hand in the x,y,z
elements of "C", and placing the Euler angle equivalence of the
orientation matrix in the elements phi,theta, and psi of "C".
***** start of NextNode *****/
NextNode (C)
HAND_POS_OR *C;{
HOMOGEN_TRANS hand; /* passed to GetEulerAngles */
    hand.pos1 = LTM->l.s.hand.pos.i;
    hand.pos2 = LTM->l.s.hand.pos.j;
    hand.pos3 = LTM->l.s.hand.pos.k;
    hand.s1 = LTM->l.s.hand.sliding.i;
    hand.s2 = LTM->l.s.hand.sliding.j;
    hand.s3 = LTM->l.s.hand.sliding.k;
    hand.n1 = LTM->l.s.hand.normal.i;
    hand.n2 = LTM->l.s.hand.normal.j;
    hand.n3 = LTM->l.s.hand.normal.k;
    hand.a1 = LTM->l.s.hand.approach.i;
    hand.a2 = LTM->l.s.hand.approach.j;
    hand.a3 = LTM->l.s.hand.approach.k;
    GetEulerAngles (hand,C);}
EndFunction

```

UpdateJointAngles (dA,theta)

```

JOINT_ANGLES    dA,*theta; {
    theta->one    = theta->one + dA.one;
    theta->two    = theta->two + dA.two;
    theta->three  = theta->three + dA.three;
    theta->four   = theta->four + dA.four;
    theta->five   = theta->five + dA.five;
    theta->six    = theta->six + dA.six;
    theta->seven  = theta->seven + dA.seven;}
EndFunction

```

```

SendCommand (theta)
JOINT_ANGLES    theta;{
    THETA_1      = theta.one;
    THETA_2      = theta.two;
    THETA_3      = theta.three;
    THETA_4      = theta.four;
    THETA_5      = theta.five;
    THETA_6      = theta.six;
    THETA_7      = theta.seven;}
EndFunction

```

```

AccSetup (na,nt,two_na,delta_C,delta_B,f1,f2)
int        na,nt,*two_na;
HAND_POS_OR delta_B,delta_C,*f1,*f2;{
double    temp;
    *two_na = na + na;
    temp    = ((double)na)/nt;
    f1->x    = (delta_C.x * temp) + delta_B.x;
    f1->y    = (delta_C.y * temp) + delta_B.y;
    f1->z    = (delta_C.z * temp) + delta_B.z;
    f1->phi  = (delta_C.phi * temp) + delta_B.phi;
    f1->theta = (delta_C.theta * temp) + delta_B.theta;
    f1->psi  = (delta_C.psi * temp) + delta_B.psi;
    f2->x    = delta_B.x + delta_B.x;
    f2->y    = delta_B.y + delta_B.y;
    f2->z    = delta_B.z + delta_B.z;
    f2->phi  = delta_B.phi + delta_B.phi;
    f2->theta = delta_B.theta + delta_B.theta;
    f2->psi  = delta_B.psi + delta_B.psi; }
EndFunction

```

```

Accelerate (A,na,two_na,f1,f2,desired_pos,i)
HAND_POS_OR A,f1,f2,*desired_pos;
int        na,two_na,i; {
double     h,h2,h3,two_h2,twoh2_h3;
    h = ( (double) i + na)/(two_na);
    h2 = h*h;
    h3 = h2*h;
    two_h2 = h2 + h2;
    twoh2_h3 = two_h2 - h3;
    desired_pos->x    = (f1.x*twoh2_h3 - f2.x)*h + A.x;

```

```

desired_pos->y      = (f1.y*twoh2_h3 - f2.y)*h + A.y;
desired_pos->z      = (f1.z*twoh2_h3 - f2.z)*h + A.z;
desired_pos->phi    = (f1.phi*twoh2_h3 - f2.phi)*h + A.phi;
desired_pos->theta  = (f1.theta*twoh2_h3 - f2.theta)*h + A.theta;
desired_pos->psi    = (f1.psi*twoh2_h3 - f2.psi)*h + A.psi; }
EndFunction

```

```

/***** start of MaintainConstantVelocity *****/
This function supplies the straight line section between points. At
time (ta )or na] when  $h = 2*na/(2*na) = 1$  as defined in Accelerate
function, the desired position is :  $desired\_position = delta\_C*(na/nt) + B$ .
The MaintainConstantVelocity function picks up at  $t = ta + 1$  and the
desired position is defined by:  $desired\_position = delta\_C*(i/nt) + B$ .
This continues until time (t-ta), when the acceleration function takes over.
*****/

```

```

MaintainConstantVelocity (B,nt,delta_C,ddesired_pos,i)
HAND_POS_OR      *B,delta_C,*desired_pos;
int              nt,i; {
double  h;
    h = ((double)i)/nt;
    desired_pos->x = (delta_C.x * h) + B->x;
    desired_pos->y = (delta_C.y * h) + B->y;
    desired_pos->z = (delta_C.z * h) + B->z;
    desired_pos->phi = (delta_C.phi * h) + B->phi;
    desired_pos->theta = (delta_C.theta * h) + B->theta;
    desired_pos->psi = (delta_C.psi * h) + B->psi; }
EndFunction

```

```

SubtractVectors1 (c,b,delta)
HAND_POS_OR      *c,*b,*delta;{
    if ((fabs(c->phi - b->phi)) > PI){
        c->phi -= PI;
        c->theta = -c->theta;}
    delta->x      = c->x - b->x;
    delta->y      = c->y - b->y;
    delta->z      = c->z - b->z;
    delta->phi    = c->phi - b->phi;
    delta->theta  = c->theta - b->theta;
    delta->psi    = c->psi - b->psi;}
EndFunction

```

```

/***** start of HandError *****/
Generates the correction vector from the hand's current position/orien-
tation and the hand's desired position/orientation.
The Euler angle representation of the desired orientation is transformed
into an orientation matrix in order to find the orientation correction.
*****/
HandError (c,ddesired,h)
HOMOGEN_TRANS  c;          /* current hand position/orientation */
HAND_POS_OR    ddesired; /*desired hand position/orientation (using Euler angles)*/
INCREMENTS     *h;         /* calculated hand error */ {
HOMOGEN_TRANS  d;          /* desired hand position/orientation (homogeneous) */

```

```
double          C_phi,S_phi,C_theta,S_theta,C_psi,S_psi,Cphi_Ctheta,Sphi_Spsi;
```

```
/***** this part obtains the desired orientation matrix from Euler Angles *****/
```

```
C_phi = cos(desired.phi);
S_phi = sin(desired.phi);
C_theta = cos(desired.theta);
S_theta = sin(desired.theta);
C_psi = cos(desired.psi);
S_psi = sin(desired.psi);
Cphi_Ctheta = C_phi * C_theta;
Sphi_Spsi = S_phi * S_psi;
d.s1 = Cphi_Ctheta*C_psi - Sphi_Spsi;
d.s2 = S_phi*C_theta*C_psi + C_phi*S_psi;
d.s3 = -S_theta*C_psi;
d.n1 = -Cphi_Ctheta*S_psi - S_phi*C_psi;
d.n2 = -Sphi_Spsi*C_theta + C_phi*C_psi;
d.n3 = S_theta*S_psi;
d.a1 = C_phi*S_theta;
d.a2 = S_phi*S_theta;
d.a3 = C_theta;
```

```
/******this part calculates error for the orientation *****/
```

This scheme uses the formula:

$$\text{orientation error} = 1/2[X_c \times X_d + Y_c \times Y_d + Z_c \times Z_d]$$

Because we are talking about minute (though finite) angle changes, if they are small enough they can be treated as a rotation vector and added. You can derive the above formula yourself by observing that delta_theta_1 (a rotation around the X axis) will carry Yc into Yd and also carry Zc into Zd. Thus to obtain a vector with components (delta_theta_1,0,0) we may add Yc x Yd and Zc x Zd, and multiply by 1/2. Performing a rotation around the other axes we find the same applies. To obtain a general rotation vector we may add all three cross products, but this duplicates the desired vector, as we have seen, so we must divide by 2. This obviously is not a strict proof, but it does bring about an understanding.

*****/

```
h->dThetaX = (c.s2*d.s3-c.s3*d.s2)+(c.n2*d.n3-c.n3*d.n2)+(c.a2*d.a3-c.a3*d.a2);
h->dThetaX *= 0.5;
```

```
h->dThetaY = (c.s3*d.s1-c.s1*d.s3)+(c.n3*d.n1-c.n1*d.n3)+(c.a3*d.a1-c.a1*d.a3);
h->dThetaY *= 0.5;
```

```
h->dThetaZ = (c.s1*d.s2-c.s2*d.s1)+(c.n1*d.n2-c.n2*d.n1)+(c.a1*d.a2-c.a2*d.a1);
h->dThetaZ *= 0.5;
```

```
/******This next part gives the error for position*****/
```

```
h->dx = desired.x - c.pos1;
h->dy = desired.y - c.pos2;
h->dz = desired.z - c.pos3; }
EndFunction
```

```

/***** start of WristError *****/

```

This function obtains the wrist velocity components from those of the hand.
It uses the formulas:

$$Ww = Wh \quad (1)$$

$$Vw = Vh - Wh \times d7 \times Zh \quad (2)$$

where

Ww = angular velocity of the wrist,
Vw = translational velocity of the wrist,
Wh = angular velocity of the hand,
Vh = translational velocity of the hand,
d7 = length of wrist joint,
Zh = approach vector of current orientation matrix of the hand.

```

*****/

```

```

WristError (error,c,w)

```

```

INCREMENTS      error; /* error is the difference in the desired pos and current pos */
HOMOGEN_TRANS    c;     /* c is the current position, from which the approach orientation
                          vector is used. */

```

```

INCREMENTS      *w; /* w is the wrist error */ {
    w->dThetaX = error.dThetaX;
    w->dThetaY = error.dThetaY;
    w->dThetaZ = error.dThetaZ;
    w->dx = error.dx - Constant_d7*(error.dThetaY * c.a3 - error.dThetaZ * c.a2);
    w->dy = error.dy - Constant_d7*(error.dThetaZ * c.a1 - error.dThetaX * c.a3);
    w->dz = error.dz - Constant_d7*(error.dThetaX * c.a2 - error.dThetaY * c.a1); }

```

```

EndFunction

```

```

double DotProduct (x,y)

```

```

JOINT_ANGLES      x;

```

```

JOINT_ANGLES      y;{

```

```

double      r;

```

```

    r = x.one*y.one + x.two*y.two + x.three*y.three + x.four*y.four;

```

```

    r = r + x.five*y.five + x.six*y.six + x.seven*y.seven;

```

```

    return (r);}

```

```

EndFunction

```

```

struct JOINTS      LeastNorm(thetaP,thetaNull)

```

```

JOINT_ANGLES      thetaP,thetaNull; {

```

```

struct JOINTS      least_norm;

```

```

    least_norm.one = thetaP.one - thetaNull.one;

```

```

    least_norm.two = thetaP.two - thetaNull.two;

```

```

    least_norm.three = thetaP.three - thetaNull.three;

```

```

    least_norm.four = thetaP.four - thetaNull.four;

```

```

    least_norm.five = thetaP.five - thetaNull.five;

```

```

    least_norm.six = thetaP.six - thetaNull.six;

```

```

    least_norm.seven = thetaP.seven - thetaNull.seven;

```

```

    return(least_norm); }

```

```

EndFunction

```



```

struct JOINTS      NullProjector(theta,thetaOperator,thetaH)
JOINT_ANGLES      theta,thetaH,thetaOperator; {
double            dotProduct;
struct JOINTS      thetaNull;
    dotProduct      =   DotProduct (theta,thetaOperator);
    thetaNull.one    =   dotProduct*thetaH.one;
    thetaNull.two     =   dotProduct*thetaH.two;
    thetaNull.three   =   dotProduct*thetaH.three;
    thetaNull.four    =   dotProduct*thetaH.four;
    thetaNull.five    =   dotProduct*thetaH.five;
    thetaNull.six     =   dotProduct*thetaH.six;
    thetaNull.seven   =   dotProduct*thetaH.seven;
    return(thetaNull);}
EndFunction

```

```

/***** start of InverseKinematics function *****/
int      InverseKinematics (W,CaseNum,dT,Cos,Sin,theta)
int      CaseNum; /* the case number,re avoidance of singularity,limit,etc,*/
INCREMENTS W; /* the wrist increment vector-structure */
JOINT_ANGLES theta,*dT,Cos,Sin;
{

```

```

    JOINT_ANGLES D_thetaP,D_thetaH,hom_sol;
    JOINT_ANGLES gH,least_norm;
    struct JOINTS thetaNullOperator;

```

```

double      X1,X2,X3,X4,X5,X6;
double      det; /* determinant of submatrix of Jacobian */
double      C6divS6; /* temporary variable */
double      Kmax,Kmin,k;
int          kKind;
int          VelLimit;

```

/* the J's below are elements of the Jacobian referred to the 3rd frame
(the midframe Jacobian). The J3inv's are the inverse of the lower right
nine elements of the midframe Jacobian. */

```

double      J11,J12,J14;
double      J21,J22,J24;
double      J31,J32,J33;
double      J41,J42,J45,J46,J47;
double      J51,J53,J55,J56,J57;
double      J61,J62;
double      J3inv11,J3inv12,J3inv13;
double      J3inv21,J3inv22,J3inv23;
double      J3inv31,J3inv32,J3inv33;
double      a2C2,a4S4,a4C4,S2S3,C4C5,S4C5,C5S6;
double      a4C2S3C4,a2C2S3,a4S2S3,a4C3PLUSa2C4,C2S3,a4S2S3_a2C2S3S4;

```

```

struct JOINTS    NullProjector();
struct JOINTS    LeastNorm();

```

```

kKind = 0;
VelLimit = 0;

```

```

a2C2 = Constant_a2 * Cos.two;
a4S4 = Constant_a4 * Sin.four;
a4C4 = Constant_a4 * Cos.four;
S2S3 = Sin.two * Sin.three;
C4C5 = Cos.four * Cos.five;
S4C5 = Sin.four * Cos.five;
a2C2S3 = a2C2*Sin.three;

```

```

J11 = a4S4*S2S3 + a2C2S3;
J12 = a4S4*Cos.three;
J14 = -a4S4;
J21 = -a4C4*S2S3;
J22 = -a4C4*Cos.three - Constant_a2;
J24 = a4C4;
J31 = -a4S4*Sin.two*Cos.three -a4C4*Cos.two -a2C2*Cos.three;
J32 = a4S4*Sin.three;
J33= -a4C4;
J41 = -Sin.two*Cos.three;
J42 = Sin.three;
J51 = Cos.two;
J61 = -S2S3;
J62 = -Cos.three;

```

```

C6divS6 = Cos.six/Sin.six;

```

```

J3inv11 = (C4C5*C6divS6) - Sin.four;
J3inv12 = (S4C5*C6divS6) + Cos.four;
J3inv13 = -Sin.five*C6divS6;
J3inv21 = Cos.four*Sin.five;
J3inv22 = Sin.four*Sin.five;
J3inv23 = Cos.five;
J3inv31 = C4C5/Sin.six;
J3inv32 = S4C5/Sin.six;
J3inv33 = -Sin.five/Sin.six;

```

```

if (Cos.four<0.0174524 AND Cos.four>-0.0174524) /* Use solution that eliminates
                                                    column 3 from J */

```

```

{
a4C3PLUSa2C4 = Constant_a4*Cos.three + Constant_a2*Cos.four;
C2S3 = Cos.two * Sin.three;
a4C2S3C4 = C2S3*a4C4;
a4S2S3_a2C2S3S4 = Constant_a4*S2S3 + Constant_a2*C2S3*Sin.four;
det = -J12*(a2C2 + a4S4*Sin.two + a4C4*Cos.two*Cos.three);
X1 = Cos.four*W.dx + Sin.four*W.dy;
X2 = W.dz;

```

```

/***** finding the D_thetaP's for column 3 solution*****/
D_thetaP.three = 0.0;
D_thetaP.one = ( J32*X1 + a4S4*X2)/det;
D_thetaP.two = (-J31*X1 + a4C2S3C4*X2)/det;
D_thetaP.four = Sin.four*W.dx - Cos.four*W.dy - a4S2S3_a2C2S3S4*D_thetaP.one;
D_thetaP.four -= a4C3PLUSa2C4*D_thetaP.two;
D_thetaP.four /= -Constant_a4;

X4 = W.dThetaX - J41*D_thetaP.one - J42*D_thetaP.two;
X5 = W.dThetaY - J51*D_thetaP.one;
X6 = W.dThetaZ - J61*D_thetaP.one - J62*D_thetaP.two - D_thetaP.four;

D_thetaP.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaP.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaP.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

/***** finding the D_thetaH's for column 4 solution*****/
D_thetaH.three = 1.0;
D_thetaH.one = a4S4*a4C4/det;
D_thetaH.two = a4C2S3C4*a4C4/det;
D_thetaH.four = - a4S2S3_a2C2S3S4*D_thetaH.one - a4C3PLUSa2C4*D_thetaH.two;
D_thetaH.four /= -Constant_a4;

X4 = - J41*D_thetaH.one - J42*D_thetaH.two;
X5 = - J51*D_thetaH.one - D_thetaH.three;
X6 = - J61*D_thetaH.one - J62*D_thetaH.two - D_thetaH.four;

D_thetaH.five = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaH.six = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaH.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;
}

else if (Sin.four<0.0174524 AND Sin.four>-0.0174524)/" Use solution that eliminates
column 4 from J */
{
X1 = W.dx;
X2 = W.dy;
X3 = W.dz;
det = J11*J22 - J21*J12;

/***** finding the D_thetaP's for column 4 solution*****/
D_thetaP.one = ( J22*X1 - J12*X2)/det;
D_thetaP.two = (-J21*X1 + J11*X2)/det;
D_thetaP.three = (X3 - J31*D_thetaP.one - J32*D_thetaP.two)/J33;

X4 = W.dThetaX - J41*D_thetaP.one - J42*D_thetaP.two;
X5 = W.dThetaY - J51*D_thetaP.one - D_thetaP.three;
X6 = W.dThetaZ - J61*D_thetaP.one - J62*D_thetaP.two;

D_thetaP.four = 0.0;

```

```

D_thetaP.five      = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaP.six       = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaP.seven     = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

/***** finding the D_thetaH's for column 4 solution*****/

X1 = -J14;
X2 = -J24;

D_thetaH.one       = ( J22*X1 - J12*X2)/det;
D_thetaH.two       = (-J21*X1 + J11*X2)/det;
D_thetaH.three     = (-J31*D_thetaH.one - J32*D_thetaH.two)/J33;
D_thetaH.four      = 1.0;

X4 = -J41*D_thetaH.one - J42*D_thetaH.two;
X5 = -J51*D_thetaH.one - D_thetaH.three;
X6 = -J61*D_thetaH.one - J62*D_thetaH.two - 1;

D_thetaH.five      = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaH.six       = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaH.seven     = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

}

else /* Use solution that eliminates column 1 from J */
{
X1 = W.dx;
X2 = W.dy;
X3 = W.dz;
det = -Constant_a2*a4S4;

/***** finding the D_thetaP's for column 1 solution*****/

D_thetaP.one       = 0.0;
D_thetaP.two       = ( J24*X1 - J14*X2)/det;
D_thetaP.four      = (-J22*X1 + J12*X2)/det;
D_thetaP.three     = ( X3 - J32*D_thetaP.two)/(J33);

X4 = W.dThetaX - J42*D_thetaP.two;
X5 = W.dThetaY - D_thetaP.three;
X6 = W.dThetaZ - J62*D_thetaP.two - D_thetaP.four;

D_thetaP.five      = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaP.six       = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaP.seven     = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

/***** finding the D_thetaH's for column 1 solution*****/

X1 = -J11;
X2 = -J21;

D_thetaH.one       = 1.0;

```

```

D_thetaH.two   = ( J24*X1 - J14*X2)/det;
D_thetaH.four  = (-J22*X1 + J12*X2)/det;
D_thetaH.three = (-J31 - J32*D_thetaH.two)/J33;

```

```

X4 = -J41 - J42*D_thetaH.two;
X5 = -J51 - D_thetaH.three;
X6 = -J61 - J62*D_thetaH.two - D_thetaH.four;

```

```

D_thetaH.five  = J3inv11*X4 + J3inv12*X5 + J3inv13*X6;
D_thetaH.six   = J3inv21*X4 + J3inv22*X5 + J3inv23*X6;
D_thetaH.seven = J3inv31*X4 + J3inv32*X5 + J3inv33*X6;

```

```

}

```

```

NullSpaceOperator(D_thetaH,&thetaNullOperator); /* get the null space operator */
if(leastnorm){
    /*calculate least Euclidean norm solution*/
    *dT = LeastNorm(D_thetaP,NullProjector(D_thetaP,thetaNullOperator,D_thetaH));
    return(LeastNormVelCheck(dT)); }
else {
    /* calculate optimizing solution */
    *dT = LeastNorm(D_thetaP,NullProjector(D_thetaP,thetaNullOperator,D_thetaH));
    GetGradientH(theta,Sin,Cos,CaseNum,&kKind,&gH);
    hom_sol = NullProjector(gH,thetaNullOperator,D_thetaH);
    theDamper = fabs( (DotProduct(gH,D_thetaH))/(GetSize(gH)*GetSize(hom_sol)));
    theDamper = pow(theDamper,.062);
    Kmaxmin (dTMax,*dT,hom_sol,&Kmax,&Kmin);
    if (Kmax > Kmin ) k = (kKind) ? Kmax : Kmin ;
    else
        return(1); /* Kmax<Kmin implies a velocity limit */
    dT->one   = least_norm.one   + k*theDamper*hom_sol.one ;
    dT->two    = least_norm.two    + k*theDamper*hom_sol.two ;
    dT->three  = least_norm.three  + k*theDamper*hom_sol.three ;
    dT->four   = least_norm.four   + k*theDamper*hom_sol.four ;
    dT->five   = least_norm.five   + k*theDamper*hom_sol.five ;
    dT->six    = least_norm.six    + k*theDamper*hom_sol.six ;
    dT->seven  = least_norm.seven  + k*theDamper*hom_sol.seven ;
    return(0);}
EndFunction

```

```

NullSpaceOperator(thetaH,thetaN)
JOINT_ANGLES  thetaH,*thetaN;{
double
    dotProduct;
    dotProduct = DotProduct (thetaH,thetaH);
    thetaN->one = thetaH.one/dotProduct;
    thetaN->two = thetaH.two/dotProduct;
    thetaN->three = thetaH.three/dotProduct;
    thetaN->four = thetaH.four/dotProduct;
    thetaN->five = thetaH.five/dotProduct;
    thetaN->six = thetaH.six/dotProduct;
    thetaN->seven = thetaH.seven/dotProduct;}
EndFunction

```

```

double GetSize(theta)
JOINT_ANGLES theta;{
double      Mtheta,;
            Mtheta = sqrt(DotProduct(thetaH,thetaH));
            return(Mtheta);}

EndFunction

```

```

Kmaxmin (tmax,tp,th,Kmax,Kmin)
JOINT_ANGLES tmax,tp,th;
double      *Kmax,*Kmin; {
double      temp1,temp2;
            if(fabs(th.one)<.0000000001) th.one = .0000000001;
            if(fabs(th.two)<.0000000001) th.two = .0000000001;
            if(fabs(th.three)<.0000000001) th.three = .0000000001;
            if(fabs(th.four)<.0000000001) th.four = .0000000001;
            if(fabs(th.five)<.0000000001) th.five = .0000000001;
            if(fabs(th.six)<.0000000001) th.six = .0000000001;
            if(fabs(th.seven)<.0000000001) th.seven = .0000000001;
            temp1 = (tmax.one - tp.one)/th.one;
            temp2 = (-tmax.one - tp.one)/th.one;
            *Kmax = max(temp1,temp2);
            *Kmin = min(temp1,temp2);
            temp1 = (tmax.two - tp.two)/th.two;
            temp2 = (-tmax.two - tp.two)/th.two;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin);
            temp1 = (tmax.three - tp.three)/th.three;
            temp2 = (-tmax.three - tp.three)/th.three;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin);
            temp1 = (tmax.four - tp.four)/th.four;
            temp2 = (-tmax.four - tp.four)/th.four;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin);
            temp1 = (tmax.five - tp.five)/th.five;
            temp2 = (-tmax.five - tp.five)/th.five;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin);
            temp1 = (tmax.six - tp.six)/th.six;
            temp2 = (-tmax.six - tp.six)/th.six;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin);
            temp1 = (tmax.seven - tp.seven)/th.seven;
            temp2 = (-tmax.seven - tp.seven)/th.seven;
            *Kmax = min(max(temp1,temp2),*Kmax);
            *Kmin = max(min(temp1,temp2),*Kmin); }

EndFunction

```

```

int LeastNormVelCheck(dT)
JOINT_ANGLES *dT; {

```

```

if (fabs(dT->one) > dTMax.one )
    return(1);
else if (fabs(dT->two) > dTMax.two )
    return(1);
else if (fabs(dT->three) > dTMax.three )
    return(1);
else if (fabs(dT->four) > dTMax.four )
    return(1);
else if (fabs(dT->five) > dTMax.five )
    return(1);
else if (fabs(dT->six) > dTMax.six )
    return(1);
else if (fabs(dT->seven) > dTMax.seven )
    return(1);
else
    return(0); }

```

EndFunction

```

int GetSign(number)
double number;{
    if(number>=0)
        return(1);
    else
        return(-1);}

```

EndFunction

Appendix L

Gradient Module (GetGradientH.c)

```
#include "unix.h"
#include "LTM.h"
#include <math.h>
```

/***** obtains the gradient of optimization criterion *****/

```
GetGradientH(theta,Sin,Cos,CaseNum,kKind,gH)
JOINT_ANGLES  theta,Cos,Sin,*gH;
int           CaseNum,*kKind;{
    switch (CaseNum){
        case 1: /* JOINT LIMITS */
            gH->one = 0;
            gH->two = 0;
            gH->three = 2*( theta.three - .9163 );
            gH->four = 0;
            gH->five = 2*( theta.five - .9163 );
            gH->six = 0;
            gH->seven = 0;
            *kKind = 0;
            break;
        case 2:
            gH->one = 2*( theta.one + 1.57 );
            gH->two = 0;
            gH->three = 0;
            gH->four = 0;
            gH->five = 2*( theta.five - .9163 );
            gH->six = 0;
            gH->seven = 0;
            *kKind = 0;
            break;
        case 3:
            /*
             Singularity (Joint 3 = 90 degrees and Joint 4 = 90 degrees )
             To be minimized
            */
            *kKind = 0;
            gH->one = 0;
            gH->two = 0;
            gH->three = 0;
            gH->four = 2*(theta.four -.1745 );/* 15 degrees */
            gH->five = 0;
            gH->six = 0;
            gH->seven = 0;
            break;
        case 4:
            *kKind = 0;
            gH->one = 0;
```



```

gH->two = 0;
gH->three = 0;
gH->four = 2*(theta.four - .2618 );
gH->five = 0;
gH->six = 0;
gH->seven = 2*( theta.seven);
break;
case 5: /*
    Stationary hand
    */
    gH->one = 0;
    gH->two = 0;
    gH->three = 0;
    gH->four = 2*( theta.four - 1.57);
    gH->five = 0;
    gH->six = 0;
    gH->seven = 2*( theta.seven);
    *kKind = 0;
    break;
case 6: /*
    Singularity theta2 == -90    theta4 == 0
    */
    gH->one = 0;
    gH->two = 0;
    gH->three = 0;
    gH->four = 2*( theta.four - .0873);
    gH->five = 0;
    gH->six = 0;
    gH->seven = 2*( theta.seven );
    *kKind = 0;
    break;
case 7: /*
    Singularity theta2 == 90    theta4 == 180
    */
    gH->one = 0;
    gH->two = 0;
    gH->three = 0;
    gH->four = 2*( theta.four - 2.88);
    gH->five = 0;
    gH->six = 0;
    gH->seven = 0;
    *kKind = 0;
    break;
})

```

EndFunction

VITA

Robert B. Magness was born in McMinnville, Tennessee on December 15, 1946. His father's work took the family to Venezuela where he spent his early childhood, and then to Iran where he completed junior high school. He was graduated from the Judson School in Scottsdale, Arizona in May 1965.

Robert entered the United States Army in February 1966. He spent 6 months in the Republic of Vietnam as an infantryman in the 1st Cavalry Division. After two Purple Hearts for wounds received in combat, he spent another 6 months at Walter Reed Medical Hospital recovering from the wounds.

After being honorably discharged from the Army he received an Associate in Arts degree in computer technology from Los Angeles City College in May 1972. He went back to college in 1982 and was graduated from the Massachusetts Institute of Technology with a Bachelor of Science degree in mechanical engineering in February 1987. He then went on to graduate school at the University of Tennessee where he received the Master of Science degree with a major in mechanical engineering in August 1989.