NOTICE

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

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

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

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

AVIS

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

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

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

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

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

Canada

An Environment for Robot Trajectory Planning and Generation

Stuart E. Thompson

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering at
Concordia University
Montréal, Québec, Canada

April 1988

ISBN 0-315-41652-1

ABSTRACT

An Environment for Robot Trajectory Planning and Generation

Stuart E. Thompson

A new programming environment called RIGID for describing, constructing and verifying complex motion trajectories of rigidly-linked bodies, such as those of a robot, is described. RIGID allows a transformation-based description of the world and a motion task to be completely integrated using kinematic equations and a trajectory construction procedure which uses the theory of B-splines. RIGID consists of a library of C language functions that are used to describe the world of the rigid-body, the rigid-body itself and to simulate the path which the rigid-body should follow.

This thesis avoids the creation of a new and specialized language for robot programming so that more effort could be concentrated on the mathematical algorithms and data structures required to solve the motion planning problems. The RIGID library requires no specific hardware, and has been implemented on a VAX 11/780 under UNIX 4.3BSD, an IBM PC/AT using the MicroSoft C compiler 4.00 and a MacIntosh Plus using Lightspeed C with very minimal effort.

The system is composed of two subsystems: the world modeler and the trajectory generator. The world modeler subsystem inputs homogeneous

transformation equations and converts them into lists of dynamic data structures. These equations, or closed kinematic chains, describe the relative locations of objects and path points in the world of the rigid-body. The trajectory generation subsystem inputs motion of the rigid-body, as specified by a time sequence of positions and orientations of the last link of the body, or end effector. These are transformed into sets of manipulator joint displacements.

Research on robot programming languages has been carried out for more than a decade, and many languages have been implemented successfully. A review of the goals and features of these languages is provided in this thesis. Earlier research in trajectory generation is presented, followed by an evaluation of various methods for approximating geometric functions. B-splines are used to fit the time sequence of manipulator motion for each joint. The procedure, consisting of simple recursive algorithms, allows constraints to be imposed both on the magnitudes of the joint velocities and accelerations and on their initial and final values. A trajectory constructed by this method has the property that local modifications can be made to a predefined path by tweaking specific control points without recomputing the entire trajectory. On-line path generation using B-splines is possible with only a few look-ahead points being required due to the flexibility of the path constraint formulation.

Acknowledgements

I am grateful to Dr. R.V. Patel, who not only provided me with the incentive and confidence I needed to succeed as a graduate student, but who also taught me how to explore new ideas. Many thanks for slotting me into his busy time schedule.

I am eternally indebt to my wife, Penny Campbell, who put up with my late hours and rambling about computers and robots.

I am also grateful to the members of the Concordia University graduate control group for their useful suggestions for this thesis and encouraged discussion on future projects.

An Environment for Robot Trajectory Planning and Generation

Table of Contents

List of Figures

Chapter 1 - Introduction

The past decade has been marked by substantial growth in the area of robotics. This scientific discipline has been drawing expertise from the fields of mechanical and electrical engineering and computer science in an attempt to improve the current state of manufacturing technology.

As part of a search for more powerful and sophisticated levels of automation, robotics is being introduced to many sectors, including manufacturing, medical technology, hazardous chemical experimentation, oceanography and space exploration. In order to perform independently and "intelligently" in these situations, robots must be aware of their environment and be able to respond effectively to unexpected events. The accomplishment of such directives necessitates solutions to problems in computer vision, tactile sensing, automated reasoning/planning and real-time control. The simple day-to-day operations often taken for granted nevertheless require tremendous complexities of interaction. For example, the simple action of reaching down to pick up a pencil requires hand-eye-touch coordination of such complexity that computers devote great computational efforts to synthesize the necessary motion and to gain an understanding of the visual information in order to incrementally accomplish this task.

An industrial **manipulating robot** is a computer-controlled, programmable, general purpose machine consisting of three major subsystems: the mechanical components, which constitute the manipulator and end effector, including the actuators that move them; sensors to measure the internal state of the robot and determine its external environment; and the control structure allowing the desired

task to be specified and executed, given the initial state of the arm and the world external to the robot. Conversely, a **mobile robot** is largely a propulsion and communication system. It may be tethered to a power and communication line or equipped with onboard systems. Although the material presented in this thesis focuses on planning motion for rigidly-linked bodies, almost all industrial robots are considered to belong to the class of rigidly-linked bodies; therefore, the principles and problems presented are valid for most industrial robot manipulators.

The great complexity in the control of a rigidly-linked body results from the nonlinear joint coupling dynamics. A vast number of numerical techniques exist for solving the types of nonlinear differential equations occurring in robotics. Robot control is usually resolved in two stages: **trajectory generation**, or path planning, and **trajectory tracking**, or path tracking. The first stage is concerned with the generation and approximation of robot joint angles under realistic manipulator constraints. The trajectory generator receives a spatial path description as input. From this, it generates a time sequence of desired positions, velocities and accelerations. The trajectory tracker is responsible for matching the actual position, velocity and acceleration of the robot with the desired values provided by the trajectory generator. The tracking algorithm must be able to compensate for dynamic response and payload loading on the manipulator. Much of the attention of the robotics engineering community is currently being directed to these two stages.

There are different strategies for describing the location of the robot along trajectories in joint space. It is desirable to interpolate the robot motion with continuous or very dense motion information so that the robot's movement is smooth. The most popular method currently used in industry involves leading the

manipulator through a sample run of the task. An interactive device, such as a joystick or teach pendant is used. During the sample runs the computer stores a large number of densely packed joint location lists from which the robot then operates in a continuous playback mode. This is referred to as **teaching-by-doing**. Inherent severe drawbacks, however, restrict this method of teaching to simple applications such as spray painting and spot welding. The principal drawbacks of this method are the flexibility limitations imposed by the dense location list representation. The repetition of a recorded sequence of motions with only minor changes requires fastidious re-programming and is difficult to edit. Pre-programmed location lists do not permit sensory data to be used to overcome uncertainties in the robot's environment. If any sort of closed-loop feedback is to be used, the determination of velocity and acceleration in order to predict certain actions requires the searching of large lists of joint locations, the accumulation of statistics and periodic checks. The coordination of several interacting robots cannot be accomplished by teaching-by-doing methods alone. On the other hand, the approximation of robot trajectories by means of geometric functions requires much less storage than dense location lists, allows a path to be easily duplicated and edited, and provides a straightforward means of computing its derivatives.

In the future, the use of high-level symbolic robot programming languages, which use verbs for actions and nouns as objects, will provide the most sophisticated and integrated user-interface for all aspects of trajectory generation, world modelling, task description and sensory feedback during trajectory tracking. Trajectory generation using approximating functions has the potential to become the basis of the motion subsystem of a robot programming language. In this thesis, an attempt to partially fulfill the development of such a system has been made. Figure 1.0 shows the three block components of RIGID system, consisting of a trajectory

generator, world modeler and rigid-body interface. Function calls to the RIGID programming language library manipulate the world modelling data structures. Other function calls create motion requests from the world models which become transformed into individual motion displacements corresponding to some rigid-body mechanism. These joint displacements are formed into joint space trajectories using a B-spline trajectory generation procedure which generates physically feasible robot movements.

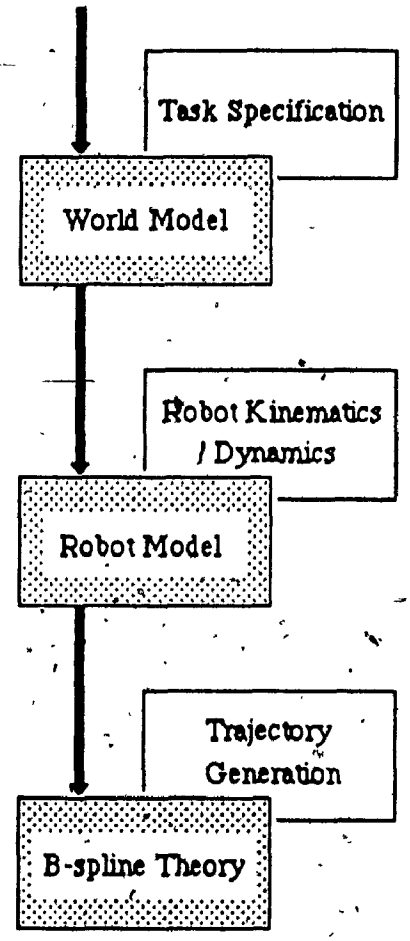


Figure 1.0: RIGID system overview.

Chapter 2 - Rigidly-Linked Bodies

The performance of any type of path planning or feedback control of a rigidly-linked body demands a thorough understanding of its anatomy. In geometrical terms, the construction of any particular rigidly-linked mechanism can be anatomically categorized and a generalized solution developed using kinematic equations in order to predict and plan the motion of each link of the mechanism.

2.1 Terminology and Anatomical Structure

There are two basic types of moving joints. A **prismatic**, or sliding joint, is that in which the length of the link varies, usually along Cartesian coordinates. With three joints prismatically-linked in orthogonal directions, it is possible to position the end-effector of the last link of a mechanism at any arbitrary point in 3D space. The more popular type of joint in industrial robots is the **revolute**, or angular, joint that rotates radially about an axis so that there is a variation in the angle between two links. Three revolute joints also allow the last link of a mechanism to be positioned at any arbitrary point in 3D space; but, unlike the case of the prismatic joint, the orientation of the last link will not be the same for each position in space. Figures 2.1a,b depict manipulators with three prismatic joints and three revolute joints, respectively.

In general, there are six geometric parameters required for the complete specification of any arbitrary position and orientation of a rigid body within some

confined workspace. This can be accomplished with six **degrees of freedom**, or number of linked joints in the body. Common industrial robots usually have six or more degrees of freedom (DOF) and consist of combinations of both prismatic and revolute joints. Typically, three large-scale motion axes provide position and three wrist motions provide orientation. A servoed gripper, or any such end effector, is considered as another DOF. Robots that have fewer than six DOF cannot get to any orientation for a given position. However, even robots with six or more joints are restricted from some movements due to possible intersection of their own bodily geometry.

Robot drive mechanisms are classified as either kinematically **open-chained** or **closed-chained**. An open kinematic chain has physical link i driven by the i^{th} actuator, whereas a closed kinematic chain permits link i to be driven in parallel by more than one actuator simultaneously. The purpose of closed-chain structures in a linked mechanism is the reduction of the loading on any one particular joint.

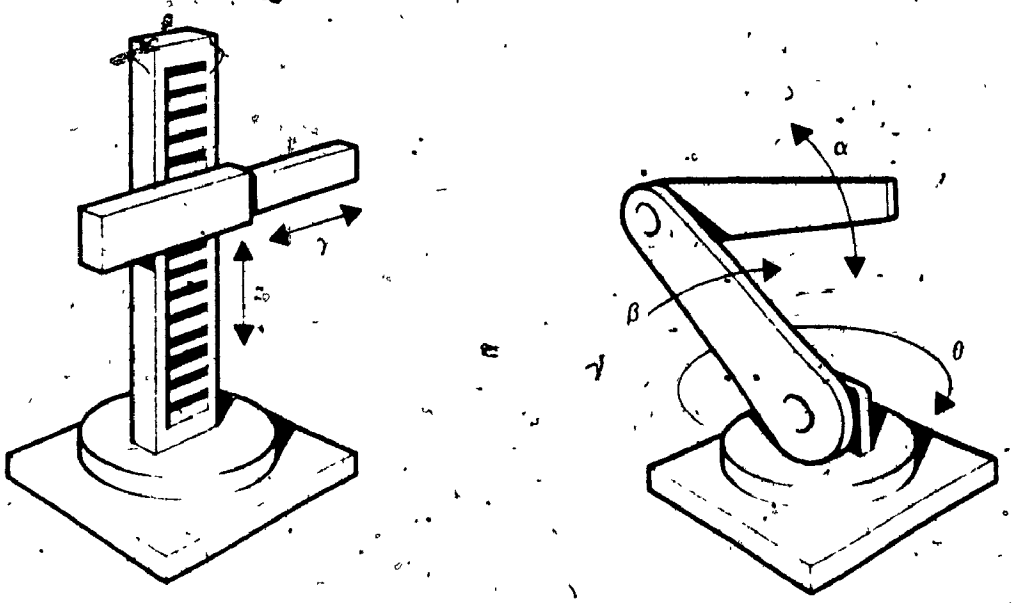


Figure 2.1a,b: Prismatic and revolute joints.

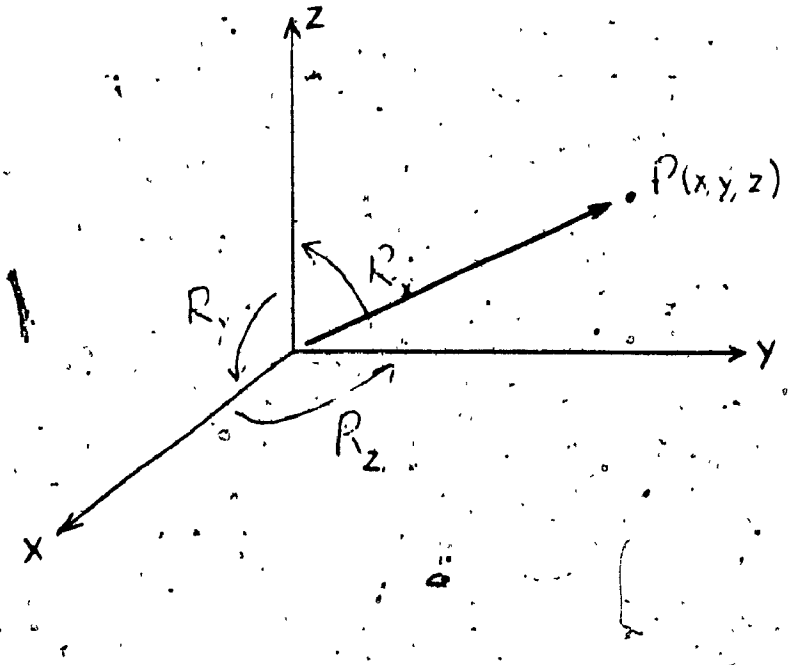


Figure 2.2: Right-handed coordinate system.

The first step in controlling a rigidly-linked body, physically constructed with combinations of prismatic and/or revolute joints, is the ability to position the last link at some location (position and orientation) in 3D space. The highly nonlinear and complex geometric coupling between adjacent links of the robot must be expressed in some mathematical form which will relate **joint space**, or movement of the joints, to the geometry of **world space**, in which the specification of a robot's task is very straightforward. In world space, position is often specified as Cartesian coordinates and orientation given as either Euler or RPY (roll, pitch, yaw) angles [1]. The problem of end-effector placement then becomes one of conversion between world and joint space coordinate systems. In reality, this conversion is easily made through the application of a well-known homogeneous matrix transformation method called the **forward kinematic solution** [1, 2]. This method develops the world space position and orientation of each link successively and systematically from the base of the robot given the joint space description. The converse problem, i.e. world space to joint space, or **inverse kinematic solution**, is much more complicated and often results in redundant solutions and singularities in the solution space. Although there are some general rules, the inverse solution is not always obvious and often requires good geometric insight to obtain a solution. As would be expected, the kinematic analysis of a closed-chain mechanism is much more difficult than that of an open-chain mechanism.

It follows from the above discussion, that the amount of **redundancy**, or number of multiple solutions, is for the most part proportional to the number of degrees of freedom of the mechanism. For example, the manipulator in Figure 2.1b is capable of reaching a specific location within most of its workspace with either an elbow-up or elbow-down solution. Redundancy is a desirable property

since it gives a manipulator added flexibility of manoeuvring, as in collision avoidance, and the ability to produce more energy-efficient motions.

Robot performance is measured by resolution, repeatability, accuracy, maximum acceleration, velocity and load capacity. These geometric and dynamic parameters are mandated by the digital nature of the control system which must compensate for the robot dynamics. The problem posed in manipulator control involves the compensation of the dynamic responses due to kinetic and potential energies in the robot motion [3]. The term used to describe the solution relating actuator forces and torques to the resulting world space velocities and accelerations of the end effector is the **forward dynamics solution**. The solution to the converse problem is called the **inverse dynamics solution**. Dynamics analysis is useful for modelling dynamic constraints (such as forces and torques) when planning optimal performance trajectories and when designing controllers based on a robot's dynamic model [4]. This problem becomes complicated by the computational issues in solving the Jacobian matrix, which has singularities for certain angular combinations. Much research has been devoted to this subject and will be reviewed in Chapter 4.

2.2 Homogeneous Transformations

The mathematical basis for the development of the forward and inverse kinematic solution is the manipulation of homogeneous transforms. A review of the basic 3D transforms is given below. Assuming a right-hand coordinate system, these transforms represent rotations about a coordinate system by an angle θ and a translation along a vector (x, y, z). The rotation matrices are:

$$\text{rot}(x, \theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{rot}(y, \theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\text{rot}(z, \theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The translation matrix is:

$$\text{trans}(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The right-hand coordinate system is shown in figure 2.2. The left-hand coordinate system results in similar matrices, with appropriate sign interchanges.

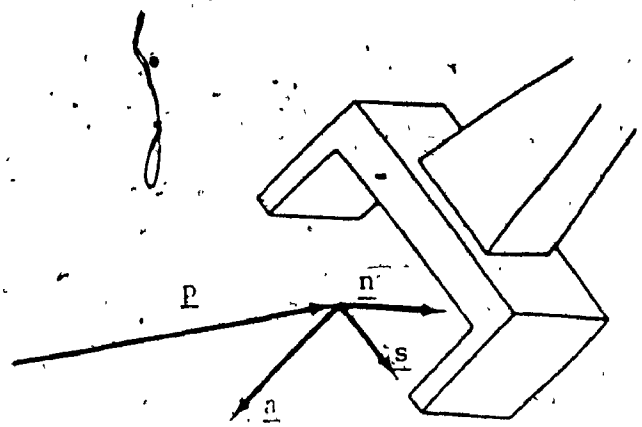


Figure 2.3: Orthogonal orientation and position vectors.

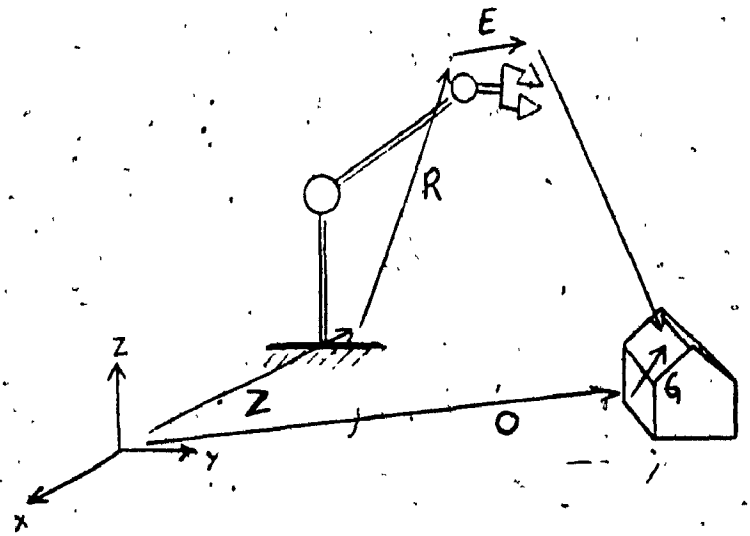


Figure 2.4: Robot location scenario.

The order of transformation matrix multiplication defines the unique course of motion, since, in general, for any two homogeneous 4x4 transformation matrices T_1 and T_2 the product is not commutative, $T_1 * T_2 \neq T_2 * T_1$. The order of a transformation matrix product has a significant geometric interpretation. Transformations in the order

$$T = \left(\prod_{i=1}^{d} T_{d-i+1} \right) T_0 = T_d * T_{d-1} * \dots * T_1 * T_0$$

where d is the number of transformations, are made with respect to the base frame T_0. In other words, if the base frame T_0 is pre-multiplied by the transformation matrix T_{d-i+1}, then that transformation is made with respect to the base frame T_0. Transformations made in the reverse order

$$T = T_0 \left(\prod_{i=1}^{d} T_i \right) = T_0 * T_1 * \dots * T_{d-1} * T_d$$

are made with respect to the coordinate frame T_{i-1}. In other words, if the current frame T_{i-1} is post-multiplied by a transformation T_i, then that transformation is made with respect to the current frame T_{i-1}.

A homogeneous transformation matrix T consists of both orientation information Ψ_{3x3} and positional information p_{3x1} as shown below.

$$T = \begin{pmatrix} & \Psi & & p \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

More specifically,

$$T = \begin{pmatrix} n_x & s_x & a_x & p_x \\ n_y & s_y & a_y & p_y \\ n_z & s_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $\underline{n}$ is the unit normal vector, $\underline{s}$ is the unit slide vector, $\underline{a}$ is the unit approach vector and $\underline{p}$ is the position vector. Figure 2.3 depicts the orthogonality of the orientation vector. Since matrix Ψ is an orthogonal matrix, the inverse transform T^{-1} is defined as:

$$T^{-1} = \begin{pmatrix} n_x & n_y & n_z & -\underline{p} \cdot \underline{n} \\ s_x & s_y & s_z & -\underline{p} \cdot \underline{s} \\ a_x & a_y & a_z & -\underline{p} \cdot \underline{a} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the dot "•" represents the vector dot product. When many matrix inversions are required this way of computing the inverse homogeneous transform represents a great computational saving over a general matrix inversion method.

2.3 Kinematic Equations

The homogeneous transformation matrices introduced in the previous section are used to specify the motion of the manipulator between its end effector and the objects in the robot's environment. The placement of objects is described by the 4x4 transformation matrix with respect to some global reference coordinate system. The concept of a **directed transform graph** (DTG) is used to aid the specification of robot start and goal motions. In the scenario described by Figure 2.4, the manipulator is located with respect to the base coordinates by the transformation Z, the last link of the manipulator is related to the robot base by the transformation R and the end effector is related to last link by the transformation E. The object is located with respect to base coordinates by the transformation O and the proper gripping position for picking and placing the object is represented by the transformation G with respect to the first vertex of the object. If the task to be performed by the robot is the pick-up of an object, the end effector and object gripping position should be equated, i.e.

$$Z R E = O G$$

Since it is the robot configuration which must produce the motions required for the accomplishment of the desired task, the kinematic equation is re-arranged as

$$R = Z^{-1} O G E^{-1}$$

The use of a DTG simplifies the formulation of transform equations and may be implemented by linked lists of transforms. It is this concept which forms the basis of many robot programming languages [1,5,6]. The corresponding DTG for the scenario of Figure 2.4 is shown in Figure 2.5. The solution to the DTG can be graphically visualized in Figure 2.5. In order to obtain an expression for R, the graph is traversed from the arrow side of R in a backward direction until R is again

reached. A simple rule applies here, that is, when a graph segment is traversed in a direction opposite to its indicated direction, it has the effect of post-multiplying the current reference frame by an inverse transform. Intermediate vertical lines mark the individual coordinate reference frames.

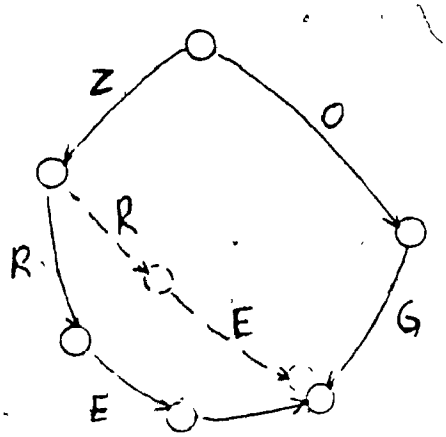


Figure 2.5: Directed transform graph.

2.4 Survey of Manipulator Solutions

The most common method of establishing geometric relationships between two rigidly-linked objects is to develop kinematic equations of A matrices (a historical name). An A matrix is the product of a link angle rotation and a link offset translation, followed by a link length translation and a twist angle rotation. In the A matrix technique, the forward kinematic equation is simple to compute, but the inverse kinematic equation is difficult to formally express and requires quite a lot of insight to sequentially isolate each joint variable from the equation. The theory of A matrices has been well documented in [1,7] and will not be repeated in this thesis. A VLSI chip has been developed which computes kinematic solutions using the A matrix method [8]. A wealth of kinematic solutions exists for many types of actual industrial robots as well as some hypothetical manipulators.

Closed-form kinematic solutions may be obtained by means of brute force application of classical vectorial mechanics for a particular manipulator structure [9]. One such example where kinematic expressions for positions, velocities and accelerations are derived for the PUMA 560 [10] robot is given by Elgazzar [11]. Another inverse kinematic solution to the PUMA 560 robot using brute force analytic geometry has been suggested by Crochetiere [12].

Due to both the lack of closed-form solutions and the trigonometric singularities involved in the inverse kinematic solution, researchers have evolved various iterative numerical techniques that are not subject to such inherent kinematic difficulties. Goldenberg, et al. [13] developed a numerical method based upon a constrained nonlinear optimization algorithm using a modified Newton-Raphson

technique to solve a system of nonlinear equations. This method can be used for a manipulator having a general structure.

There is one class of open-chain manipulators where the end effector axes intersects two-by-two rather than having three intersecting axes at the end effector (roll, pitch, yaw). In this particular case, it is very difficult to obtain a closed-form inverse kinematic solution. Lumelsky [14] proposed an iterative procedure based on A matrices. Sciavicco and Siciliano [15] later demonstrated a novel Jacobian formulation technique to solve the problem.

Solutions for forward and inverse kinematics based on the link transformation matrix method have been developed for many different types of industrial robots, i.e.

PUMA 560 or 600 [7, 16-18],
Stanford arm [1],
Rhino XR-2 [19],
hypothetical 6 DOF elbow manipulator [1],
Yasukawa Motoman L-3 [7].

The complete closed-form dynamic model for the PUMA robot has also been derived [20]. Another type of robot consists of multiple arms (dual grippers) with a common robot controller. The kinematics of such a mechanism have been discussed in [21].

In summary, homogeneous transformations may be used to describe the position and orientation in space of coordinate frames. Objects can be geometrically related to each other by homogeneous transformation matrix equations and directed transform graphs, thus allowing forward and inverse kinematic solutions to spatial placement problems. Links of rigidly-linked bodies can be mathematically related

by transform equations of A matrices. In terms of trajectory generation, the desired task of the robot can be expressed in joint space as a sequence of joint revolute angles and prismatic lengths.

Chapter 3 - Task-Level Planning

The problem of transforming a **task-level** description of an assembly task into a manipulator-level program, the execution of which will make the robot accomplish the task, involves trajectory generation, collision avoidance, automated grasping, fine motion synthesis, coordinated motion and task-level programming. Trajectory generation is discussed in Chapters 4 and 5. This chapter provides an overview of the current research in each of the other topics involved in task planning, as well as a list of historical references.

3.1 Collision Avoidance

Collision avoidance is the ability to prevent unexpected collisions between the robot arm and the real world, or between payloads and obstacles that are inside the work envelope of the robot. A closely related topic to collision avoidance is the **find-path problem**, the ability to synthesize a collision-free path among obstacles. This problem becomes very complex when a rigidly-linked body must navigate its way and protect each of its links from obstacle contact. Most commercial robot systems operate on the assumption that their work environment is well-known and completely preprogrammed. Such systems are not flexible, being incapable of adapting to unexpected situations. It is estimated that 80% of their cost is incorporated in unifunctional jigs and fixtures needed to pick or place parts [22]. Flexibility makes robotics easier to cost-justify, as systems that allow changing product designs let manufacturers react quickly to shifts in the market-place. An extensive collection of papers on the many techniques for collision avoidance and the find-path problem are contained in the reference list [23-53]. Most attempts to

solve the general collision avoidance problem become burdened by the enormous amounts of computation involved in the generation of global models of obstacle surfaces and the search for collision-free regions in which the manipulator may move.

3.2 Automated Grasping

Another closely-related problem is **automated grasping**. This problem consists of finding surfaces on either known or unknown objects which are suitable for grasping with a gripper or articulated fingers. The object must not slip during collision-free movement and the object surfaces must be accessible to the gripper geometry. Handling rules must be incorporated into the grasp planner for handling objects, e.g. a pin to be inserted should not be grasped lengthwise. Grasping is a form of marginal collision where the robot touches the obstacle. Few attempts have been made to find a general solution, although some results exist [54-60].

3.3 Fine Motion Synthesis

The term **fine motion synthesis** describes the methods used in high tolerance applications to provide incremental corrective motions to the end effector. This type of motion is often synonymous with the term **compliance** [61], but is differentiated from gross motion planning which involves the manipulation of a payload to and from fine motion activities. Compliance can be thought of as the positive force being applied while a screw is being driven into a hole so that the screwdriver keeps in contact with the screw. Part mating is one of these applications, requiring several sensor-based feedback mechanisms. The most commonly researched fine motion synthesis problem is that of inserting a pin into a

hole. If the pin hits the hole chamfer (surface edge of hole) first, a force sensor should indicate this and trigger a corrective motion to displace the pin's approach vector towards the hole center. Until recently, very little work existed that was aimed at the automatic synthesis of fine motions [62, 63].

3.4 Coordinated Motion

In situations where multiple manipulators use an overlapping workspace, studies have shown that industrial robots waste most of their time waiting for the other robot to leave the workspace [64]. One team of researchers has been examining manipulator-to-manipulator coordination for two cyclindrical, stick-figure, two DOF robots with one revolute and one prismatic joint [65]. Actual collision space is first computed, then a right-of-way precedence is established. A detailed, three-level hierarchical decision system is devised, mixing global information with local information about the assembly task. Other techniques for coordinating two robots have been investigated [66, 67].

3.5 Task-Level Programming Languages

Many styles of user-interfaces for robot programming have been developed. This section covers the explicit programming language aspect of robot operations. In high-level task-oriented languages, physical objects are referred to with nouns that represent the object (example: flange, bolt, plate_handle) and manipulated by action verbs (example: insert, move_up, twist) with optional modifiers (example: straight_line, slowly) and are subject to constraints (example: until_max_torque). The literal specifications are compiled into much lower forms where objects are represented as 3D coordinate frames and manipulated by spatial transformations.

On a yet lower motion level the joint angle movements are calculated from the 3D coordinate frame representation. At this level joint motion constraints are applied to the manipulator in terms of joint rates, loads and allowable positions of the individual joint actuators. Good examples of such types of manipulator-specific languages are AL [68, 69] and VAL II [70].

The other approach to robot programming is to provide a framework in some existing computer language, and provide the function library which supports the control primitives required for task-level programming. Examples of this are: JARS [71] developed by JPL, which is based on the Pascal programming language, and RCCL [5] using the C programming language and a modified version of the UNIX kernel to support extra communications hardware interaction with the robot controller. It has been proposed that Ada would be a suitable language [72] since it supports concurrency, a real-time executive and the most advanced data abstractions available.

Some work has been done in the ability to infer positions of bodies from specified spatial relationships using a language called RAPT [73]. Given a manipulator and some geometric rules about its environment, the system automatically forms sequences of motion equations which are applied against any task specified by the user.

A number of powerful task-oriented robot programming languages are emerging that allow external sensors to be interfaced with the robot controller so that intelligent performance can be obtained. With such a state-of-the-art, controllers must be able to react quickly to a series of changing directives. One approach, that of Henderson, Fai and Hansen [74], is to provide a UNIX-like

interface that can be logically reconfigured via a sensor operating system kernel, to real-time data acquisition elements. Other recent publications exist on languages for sensor-based control [75].

Many detailed reports and surveys on commercial and research laboratory languages exist [76-79]. Although most robots are still programmed using guide-through with a teach pendant and playback of learned points, complex assembly and part manipulating tasks will demand high-level task-oriented languages which need to interface with a variety of factory networks that link CAD/CAM systems, and support concurrent sensory data acquisition and process planning. Researchers surveying this area [76, 77] have concluded that in such an application as robotics, where program execution concurrency should be at its highest, this issue has been ignored by robot programming language developers. As a whole, links necessary for the formation of a complete framework for task planning are missing at present.

Chapter 4 - Geometric Modelling for Trajectory Generation

This chapter is divided into two parts. The first section provides an overview of the methodology of fitting joint space robot trajectories, and investigates various geometric approximating techniques to determine that approximation function which best suits the requirements of a trajectory generator. The second part presents a comparative survey of the most significant research work in trajectory generation, along with a brief discussion of the strengths and weaknesses of each.

The desired robot trajectory may be specified in either Cartesian (world space) or robot joint space coordinates. Both methods require transformations between the two coordinate systems. During path tracking, the end effector is usually required to travel between adjacent sampling points along straight line segments for high tolerance applications. The piece-wise linear path along which the end effector actually moves inscribes the desired path. The amount of error between these two paths depends on the sampling rate. As path inscription error is caused by the non-Cartesian motion of joint actuation, it is directly dependent upon the particular physical construction of the manipulator. A more detailed description of the techniques that can be used to reduce path inscription error are presented in section 4.2. For now, we proceed on the assumption that path planning is performed in joint space coordinates.

The concept of the homogeneous transformation matrix was discussed in section 2.2. In robotics, this concept is more commonly referred to as a **hand matrix** H(t) at some instant in time t along the trajectory of the end effector. The

hand matrix is obtained from the given task in world space coordinates. Joint values corresponding to H(t) can be generated by the inverse kinematic solution.

Letting the set $\{H(t_1), H(t_2), \ldots H(t_m)\}$ be the collection of hand position matrices for a sequence of m sample points, or **knots**, the joint solutions will be a series of joint vectors for d degrees of freedom, as in

$$[q_{11}, q_{21}, \ldots, q_{m1}], [q_{12}, q_{22}, \ldots, q_{m2}], \ldots, [q_{1d}, q_{2d}, \ldots, q_{md}]$$

where q_{ij} is the displacement of link j of a rigid body at knot i corresponding to H_i (t). An approximating curve is then constructed for each link j to fit the sequence $q_{1j}, q_{2j}, \ldots, q_{mj}$. The trajectory of the rigid body will be represented in joint space as a function of time q_{ij} (t).

4.1 Evaluation of Geometric Modelling Functions

To determine the type of interpolating function which best fits a joint trajectory, various geometric modelling techniques from simple polynomials to parametric splines are investigated. One important concept often used is continuity. A curve $\Phi(x)\ x_1 \le x \le x_n$ is said to be C^r continuous if Φ has a continuous r^{th} derivative with respect to x on the interval $[x_1, x_n]$.

Polynomials

A polynomial p(x) of degree r has a number of desirable properties. It is C^r continuous (no singular points), may be differentiated and can be easily evaluated for any value of x (unlike trigonometric functions which require reference tables). If the joint trajectory has m intermediate points, the minimum degree of the polynomial that passes through all the points is m-1. Such polynomials have m-1

minima and maxima [80]; their oscillations cause problems in position control, given the limited work-space of the robot. Infact, interpolating functions of low degree are available which do not oscillate as much.

Piecewise Polynomials

By connecting a series of low-degree polynomial segments, the overshoot and wander may be reduced. Unfortunately, the slopes at the ends of the curve segments are usually discontinuous [81]. In robot trajectory formulation this manifests itself as discontinuous velocities or accelerations, resulting in jerky and energy-inefficient motion of the robot.

Hermite Interpolation

Although these low-degree polynomials provide better curve segment blending capabilities than piecewise polynomials, they require a comprehensive advance knowledge of both the function to be interpolated and its first derivative. Hermite's formula is very sensitive to changes in derivative values which are not always available in advance for robot path planning. Hermite interpolation also exhibits local path interpolation. In this sense, although the alteration of any piece of data has influence over a limited region of the curve, it is at the expense of a discontinuous second derivative (jerky motion) [82].

Splines

Cubic and quartic splines are quite common in computer graphics. First and second derivatives are always continuous, and result in smooth-looking curves and surfaces. Quartic splines allow additional boundary point constraints to be placed on the interpolating function [81, 83]. In addition, the spline function basis allows curves to be fitted with little computation. Many forms and applications of spline

generation have been investigated by researchers in the field [84]. The disadvantage of using these natural spline functions is that a local modification to any section of a trajectory involves the recomputation of the entire trajectory [81]. This is a computational burden, particularly when the desired trajectory contains hundreds of points.

Bézier Curves

The natural Bézier curve possesses global interpolation capability only (the movement of a single control vertex affects the entire curve), whereas an alternate Bézier curve formulation permits local control at the expense of slope continuity [82]. The resulting curve segments simply share end vertices and possess a jagged appearance (jerky motion).

B-Splines

These are locally supportive splines which are non-zero functions over some finite span [83]. B-splines (basis spline) of degree r-1 are C^{r-2} continuous, thus allowing low-degree B-splines to achieve a continuous acceleration profile (C^2). Similar to natural splines, B-splines require little computation to fit a trajectory. Their major advantage, however, stems from their local support basis, i.e., local changes in a trajectory may be performed by simple arithmetic formulas, without the need to recompute data for the entire spline function.

ß-Spline

The ß-spline (beta-spline) [82, 85-87], a generalization of the B-spline, was developed for computer graphics and CAD applications. Rather than the traditional constraints of first and second derivatives for obtaining geometric continuity (which results in a smooth-looking function), the constraints here are more appropriate unit

tangent and curvature vectors. These are formulated in such a way that two new shape parameters (bias and tension) can manipulate the shape characteristic of local regions of curves. The tension parameter allows the curve to be tightened in such a way that the graph segment between two adjacent knots becomes a straight line, rather than the usual rolling curve.

ν-Spline

The ν-spline (nu-spline) [88, 89] is a generalization of the ß-spline. The local modification of a ß-spline tension parameter is a computationally expensive procedure. The ν-spline was developed to overcome this difficulty. These splines use more tractable piece-wise cubic curve segments that are as smooth as standard cubic splines. The resulting curve is a global representation with respect to its knots but has a distinct tension value at each knot.

The consideration of evaluation of various geometric modelling functions, based on the ability to make local modifications easily, maintain C^2 continuity, and introduce boundary conditions, has led us to choose B-splines as the best possible curve-fitting technique to meet the requirements for approximating robot trajectories. Both the ν-spline and ß-spline curve representations are overly sophisticated for trajectory generation and are more computationally expensive than B-splines. To understand better the specific demands placed upon robot trajectories, we have provided a summary of previous trajectory generation schemes in the following section.

4.2 Review of Trajectory Generation Research

Many of the methods currently used in industrial applications are based on **bang-bang** trajectories [90]. In these trajectories, the robot joints are forced between full velocity and full stop for each desired location of the robot end effector. In between desired locations, the manipulator is programmed to move along the path with constant velocity and acceleration. The magnitude of these velocities and accelerations is usually chosen at the factory by trial and error so that the actuators will not saturate at any point along the path. As the actuators may be near saturation at only a few points along the path, they will operate at most points at less than their full capacity. This extremely inefficient trajectory generation method is a by-product of the simple modelling techniques and control algorithms used in the design of current robot controllers. In an early comparison of trajectory generation methods, Mujtaba [91] looked at the quintic polynomial, cosine function, sum of a sine function and a linear trajectory, sum of decaying exponentials and other bang-bang methods. This section provides a review of many of the more recent efforts to develop a more sophisticated and efficient robot trajectory planner.

As stated earlier, the generation of robot motion may be broken down into two stages: trajectory generation, or the determination of those time functions which specify a course of time-dependent actions, such as manipulator joint angles; and trajectory tracking, the implementation of the motion through feedback control. This section focuses primarily on trajectory generation, but, also includes an account of some research efforts to combine the two stages into a unified theory. The studies reviewed represent the major contributions to trajectory generation up to

the present time, and is organized according to the type of trajectory approximation function employed.

Piecewise Polynomials

Path planning may be achieved by specifying the path of the robot arm by means of Cartesian coordinates or by joint angle positions. Planning in Cartesian coordinates usually requires the robot end effector to follow straight line segments between adjacent sampling points. However, the actual path traversed by the robot inscribes the desired path. This is the result of both the complex geometry of the robot joint linkages and the distance between the points at which Cartesian to joint transformations are performed. Some applications, such as part insertion, tracking a conveyor belt, transportation of liquid in open containers, and seam welding require straight line motion. To achieve the **straight line motion**, Whitney [92, 93] suggested a method by which velocities and positions in Cartesian space are repeatedly transformed into joint space rates and angles by means of a nonlinear Jacobian matrix equation. This method has been shown to be too time-consuming for any type of real time computation.

A different approach to straight line motion was reported by Paul [1, 94]. In this approach, only selected points on the straight line segments are transformed to joint space coordinates. Assuming a three segment curve consisting of acceleration, constant velocity, and deceleration, a fixed time in which the transformation calculations may be computed is allotted during the course of the trajectory. Although this procedure eliminates the stopping at each linear segment transition inherent in Whitney's method, it reduces the sampling rate of path control. Paul's approach does not lend itself to any type of optimal control, or any other type of

more advanced sensory ability. This is because of the computational requirements of the method.

To avoid the online coordinate transformation of the sampling points, Taylor [95] selected sufficient intermediate points on the Cartesian straight line segments, then transformed them into joint coordinates during an offline planning stage. This iterative algorithm adds sufficient intermediate points to allow the path inscription error to become smaller than some prescribed value. In his method, Taylor assumed that the maximum inscription error occurs at segment midpoints. This has since been shown to be untrue.

Khalil attempted to develop online straight line motion planners [96, 97] by composing a path of straight lines connected by circular arcs. Maximum velocity and acceleration constraints are satisfied, but the path is only C^1 continuous because quadratic interpolating functions are employed. Again, the velocity is assumed to be a trapezoidal-shaped function (acceleration, cruise, deceleration), resulting in a jerky motion causing excessive wear and tear on the motion mechanisms. Castain and Paul [98] have presented an online trajectory fitting method similar to Khalil's. Their interpolating functions are of higher degree, so that the acceleration waveform resembles a trapezoid. To properly satisfy all of the velocity and acceleration constraints, a combination of (cubic - quadratic - cubic) polynomials were used. The resulting path is C^2 continuous and may be formed with knowledge of only a few of the next points along the desired path.

The minimum-time path planning problem was investigated by Luh and Lin [99]. They presented a method for achieving the desired position and velocities along a path with minimum travelling time, subject to Cartesian space constraints on

linear and angular velocities and accelerations. The difficulty with this approach lies in the computational effort required to continually transform Cartesian coordinates into joint space coordinates. Like most optimization procedures, this one requires complete advance knowledge of the entire path before the algorithm can begin to generate a trajectory.

Splines

Lin, Chan, and Luh [100] used cubic splines to approximate joint movements, and a time interval scheduling algorithm to traverse the trajectory in minimum time while still satisfying the velocity and acceleration constraints. A nonlinear optimization technique was adopted to search through the complete set of points for each joint, locating the minima/maxima of the joint velocity, acceleration and jerk profiles. The time intervals between each pair of adjacent points were adjusted to ensure that none of the specified constraints on velocity, acceleration and jerk of the robot arm were violated during the course of the trajectory. This procedure also requires a knowledge of all sampling points prior to execution in order to determine the optimal approximating spline knot placement, thus making it unsuitable for any sort of online collision avoidance action. In a later paper, Lin and Chang [101] modified this approach to include the dynamic response of the manipulator so that the method minimizes the travelling time in a more precise manner. The practice of incorporating the dynamic response with the trajectory generator is discussed in the next section.

One approach to the online fitting of spline functions to a robot path was developed by Lin and Chang [102]. Two types of interpolating functions, quartic splines and X-splines, are necessary to meet all of the constraints. The X-spline is a generalization of the cubic spline in which the second derivative is allowed to be

discontinuous at the sample points. By using the X-spline, the local joint trajectory can be determined for the next two or three joint values without requiring a knowledge of the complete set of points. This is suitable for online trajectory generation. The effect of the discontinuity of the second derivative may be reduced by decreasing the time interval between sample points. Using quartic splines, it is possible to set the velocity and acceleration to specified values at the end points of the trajectory, while at the same time ensuring C^2 continuity. Lin and Chang claim that, at most, only three successive points are required to fit a segment of the approximating curve.

The problem of applying spline functions to achieve straight line motion was addressed by Luh and Lin [103]. A combination of cubic splines and quartic splines, with a least-squares-error criterion for "best fit" was used to suggest many possible variations on a basic method. Conditions were imposed on initial and final velocities of each curve segment as well as path inscription error. The resulting cubic spline function is C^2 continuous. This is an example of an offline approach.

A more refined method of online spline interpolation without complete discontinuous acceleration was demonstrated by Chand and Doty [104]. The authors accomplished this by a new spline construction technique in which only partial look-ahead knowledge of the total path was available. A trade-off between the amount of look-ahead knowledge, derivative continuity and the prescribed path deviation error value had to be established in order to fit a curve.

Dynamic Response with Splines

Other contributions to robot trajectory generation fall into the category of trajectory optimization for some criterion with the dynamic response of the

manipulator being considered. The motivation for this approach is the fact that the usual assumption of constant velocity and acceleration upper bounds is in many cases not valid. In practice, these quantities vary with angular position, payload mass and even payload path geometry. All other research approaches are based on a chosen worst case constant bound, and tend to underutilize the full capability of the robot. The approach of Vukobratovic and Kircanski [105] is an example of a robot trajectory optimization technique with optimal total energy consumption minimization for the actuators (hydraulic or electric DC motors). Using the Lagrangian formulation of robot dynamics [1], physical constraints on velocity are satisfied. Acceleration limits are not specified and the function is only C^1 continuous. Most optimization methods in dynamic response path planning use minimum-time criteria similar to those of Kim and Shin [106], Shin and McKay [107], Bobrow, et al. [108], or Lee and Lee [109].

The principle common to these algorithms is the assumption of a three-piece movement curve of constant acceleration, constant velocity and then constant deceleration around each sample point. Using approximate Lagrangian formulations of the robot dynamics (omitting some dynamic terms, such as coriolis and centrifugal forces), the minimum time problem is decomposed into a local optimization for each point. These methods differ in their particular approach to the search algorithm for switching points, geometric path description, and manner of imposing constraints. In the dynamic response approach, it is more natural to first refer to torque and velocity constraints, then convert them into local angular acceleration and velocity bounds using the Lagrangian formulation. Another interesting scheme [110] incorporates the minimum-time optimization of pre-defined paths within the control loops on the robot joint actuators. This reduces the problem to one of the choice of path segment times between given sample points

while the robot is operating under closed-loop feedback control (trajectory tracking).

Shin and McKay [111, 112] presented an improvement to the minimum-time motion problem using dynamic programming to find the optimal phase plane trajectory. In their previous work [107], they made the assumption that joint torques (acceleration jump or jerk) can be changed instantaneously. The advantage of this approach is that the high-dimensionality that is normally associated with dynamic programming is avoided. Unlike most formulations, this method does not require specific assumptions about a particular manipulator structure in order to obtain a solution.

Except in a very few of the online trajectory generation procedures, the collision avoidance problem has in general not been treated. Most of the schemes presented do not allow for a pre-computed trajectory to be quickly updated by an outside source such as a sensory feedback centre.

Based on the above discussion and the advantages and shortcomings of the various methods examined, the following list has been compiled for the requirements that should be satisfied by a robot joint trajectory formulation scheme:

- a uniform interpolating function (as opposed to a mix of many types of splines or polynomial functions of varying degrees,
- an ability to constrain first and second derivatives (velocity and acceleration) to some finite values at the end points of a curve segment,
- an ability to satisfy upper and lower bounds on the magnitudes of the first and second derivatives over the range of the complete trajectory,
- an efficient offline and online formulation technique,

- an overall C^2 continuity,
- an ability to locally modify a preplanned path without total recomputation
- a negligible positional deviation limit.

As we shall see in the next chapter, these requirements can be satisfied by the use of B-splines.

Chapter 5 - Trajectory Generation using B-Splines

The first section details the geometric theory of B-splines. Particular emphasis is given on how B-splines may be used to model the physical situations which arise during the control of robot joint actuators. Properties such as derivatives, end point constraints, and boundary conditions of B-splines are discussed with a mathematical development of each. Two methods are described for making local modifications to a B-spline trajectory, **tweaking** and **dynamic trajectory injection**. These are useful for developing the scheme for online generation of robot trajectories. The procedure for constructing robot trajectories was described in [113, 114].

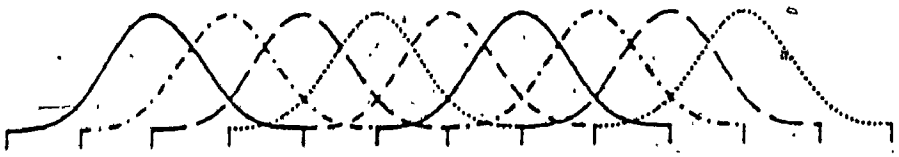
5.1 General B-Spline Theory

The B-spline or spline of minimal support of order k (or degree k-1) is characterized by two fundamental properties: it is nonzero over only k consecutive spans, and within this range it is positive definite. This results in two features that are extremely useful in geometric applications: the restriction of the values that a B-spline function can assume in a given parameter range to the convex hull associated with that range, and the possibility of making strictly local changes to B-spline functions [115]. B-spline hulls are illustrated in figure 5.1. The method of inducing strictly local changes by modifying selected vertices of the B-spline basis is called **tweaking**.

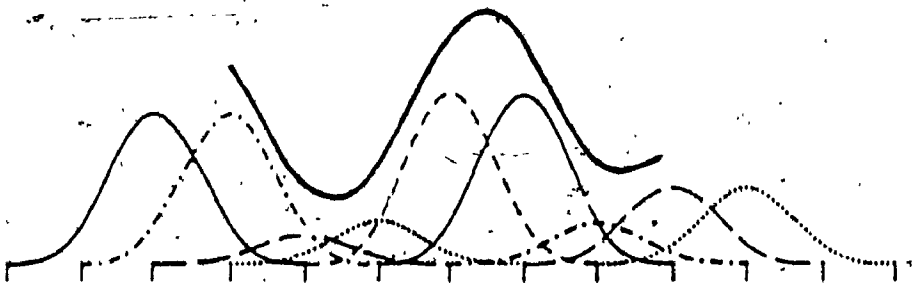
A spline is defined analytically as a set of polynomials over a knot vector. A B-spline curve of order k at a given point lies within the convex hull of its neighboring

k knots; in other words, all points on a B-spline curve must lie within the union of all such convex hulls formed by taking k successive knots. To begin our discussion, we introduce some nomenclature. A **data point** is a point through which the approximating spline function must pass, whereas a **knot** is a point within the data point set at which the B-spline function formulation algorithm is applied. As it will be shown in equation (5.1), knots and control vertices completely describe a B-spline curve. A data point is denoted by the symbol x_i and a knot by η_i. The concept of an **extended knot** set will be introduced in which we refer to an extended knot by the symbol ξ_i.

It is possible to make a trajectory approximation using B-splines of order k over m knots $\eta_1, ..., \eta_m$ on some open interval of data points (x_1, x_{m+k}). The m+k data points must be distinct and nondecreasing with a corresponding function value set $f_1, ..., f_{m+k}$. The placement of the knot set within the data set will be discussed later. In general, any spline of order k can be expressed as a sum of products of B-splines defined on the original knot set $\eta_1, ..., \eta_m$ extended by k additional knots at each end of the knot set. The additional knots are added for the sole purpose of defining a complete set of m+k successive B-splines; the value of these may be chosen arbitrarily, but for our purposes, they will be chosen in specific ways. A full m+k successive B-spline basis, in which each B-spline is nonzero over just k consecutive spans, allows the extra freedom in positioning the knot set over the open interval of data points, and in specifying some boundary conditions.



(a) Unweighted B-Splines



(b) Function Approximation with Weighted B-Splines

Figure 5.1a,b: B-spline hulls.

Multiplicity

One important factor that should be discussed is **multiplicity**. The strict monotonicity in the B-spline knot set can be relaxed to allow multiple knots, i.e. two or more identical consecutive parameter values, at the expense of continuity. The number of identical values is called the multiplicity h of a knot. If all knots are simple (h=1), the basis functions are C^{k-2} continuous, but if the support on the interval (η_i, η_{i+k}) of $N_{k,i}(\eta)$ contains a knot of multiplicity h (≥ 2), the continuity of $N_{k,i}(\eta)$ is reduced to C^{k-h-1} at that knot (where negative values denote a simple discontinuity). This is the reason for the requirements that the knot set should be monotonic and nondecreasing [83].

B-Spline Basis

All the B-splines used in this development are normalized B-splines and will be denoted as $N_{k,i}(x)$ where k is the order of the B-spline at the i^{th} data point, such that

$$\sum_{i=1}^{m+k} N_{k,i}(x) = 1 \text{ for all } x \text{ contained in } (x_1, x_{m+k})$$

The m+k successive B-splines of order k form the unique composite spline function:

$$\Phi(x) = \sum_{i=1}^{m+k} a_i N_{k,i}(x) \qquad (5.1)$$

The coefficients $\{a_1, ..., a_{m+k}\}$ are determined by equating $\Phi(x)$, where x is contained in (x_1, x_{m+k}), to the function being approximated $\{f_1, ..., f_{m+k}\}$ and solving the resulting set of linear algebraic equations. The coefficients, otherwise

known as **control vertices**, serve as the controls to manipulate how the B-spline curve is required to pass through the knots. A B-spline approximates, but does not interpolate a set of control vertices.

To ensure that a solution for the a_i's can be found in an efficient manner, some conditions must be satisfied [83]:

i) the number of knots m must be ≥ 2,

ii) the order of the spline function must be within the range

$$1 \leq k \leq m+k,$$

iii) the points must be ordered so that

$$x_1 \leq x_2 \leq \ldots \leq x_{m+k},$$

iv) the knots must be ordered so that

$$\eta_1 \leq \eta_2 \leq \ldots \leq \eta_m,$$

v) the Schoenberg-Whitney conditions

$$x_i < \eta_i < x_{i+k}, \quad (i = 1, \ldots, m).$$

To write $\Phi(x)$ as a linear combination of m+k B-splines an additional 2k knots are added to the original knot set to form the **extended knot** set, given by ξ_i, i = 1, ..., m+2k. They are determined as

$$\xi_k = \xi_{k-1} = \ldots = \xi_1 = x_1 \tag{5.2}$$

and

$$\xi_{m+2k} = \xi_{m+2k-1} = \ldots = \xi_{m+k+1} = x_{m+k} \tag{5.3}$$

and the intermediate values of ξ_i by

$$\xi_{j+k} = \eta_j \quad , (j = 1, \ldots, m) \tag{5.4}$$

The normalized B-spline $N_{k,i}$ is now defined over the extended knot set $\{\xi_1, \ldots, \xi_{m+2k}\}$, but we refer to the B-splines as being defined over the data point set for convenience, since there are exactly m+k B-splines spanning the extended knot set

and there are also exactly m+k data points. Also, the B-splines must satisfy the condition $N_{k,i}(x) = 0$ for all x not in $[x_i, x_{i+k}]$. The spline function is evaluated only at the original data points f_j, (j = 1, ..., m+k). This results in a set of linear algebraic equations

$$\sum_{i=1}^{m+k} a_i N_{k,i}(x_j) = f_j \qquad , (j = 1, ..., m+k) \tag{5.5}$$

which can be written as $N\,\underline{a} = \underline{f}$. The expressions for $N_{k,i}$ will be given shortly. Because the knots, η_i satisfy the Schoenberg-Whitney conditions, the spline function at a point depends only on k B-splines. It follows then, that there are at most k non-zero elements in each row of the coefficient matrix N and that these occur in adjacent positions. The general form of the coefficient matrix N is

$$N = \begin{pmatrix} N_{k,1}(x_1) & N_{k,2}(x_1) & \cdots & N_{k,m+k}(x_1) \\ N_{k,1}(x_2) & N_{k,2}(x_2) & \cdots & N_{k,m+k}(x_2) \\ \cdots & \cdots & \cdots & \cdots \\ N_{k,1}(x_{m+k}) & N_{k,2}(x_{m+k}) & \cdots & N_{k,m+k}(x_{m+k}) \end{pmatrix} \tag{5.6}$$

In addition, the first nonzero element of each row is either in the same column as, or to the right of the first nonzero element in the previous row. It can be shown from equations (5.2) and (5.3) that the first and last rows of N contain only one nonzero element, namely $N_{k,1}(x_1)$ and $N_{k,m+k}(x_{m+k})$, which have the value 1 due to the normalized nature of B-splines. The matrix N is therefore a band matrix.

The Schoenberg-Whitney-conditions imply that N is a nonsingular matrix so that (5.5) has a unique solution for the a_i's.

It now remains to show how the B-spline basis function $N_{k,i}(x)$ is formed. At a given point x, only k numbers are needed. The recurrence relation of de Boor [116] and Cox [117] allows one to compute $N_{k,i}(x)$ by repeatedly forming positive linear combinations of positive quantities according to

$$N_{k,i}(x) = \frac{(x - \xi_i)}{(\xi_{i+k-1} - \xi_i)} N_{k-1,i}(x) + \frac{(\xi_{i+k} - x)}{(\xi_{i+k} - \xi_{i+1})} N_{k-1,i+1}(x) \qquad (5.7)$$

starting with

$$N_{1,i}(x) = \begin{cases} 1 & \text{if } \xi_i \le x < \xi_{i+1} \\ 0 & \text{otherwise.} \end{cases}$$

The formulation of B-splines is based on the procedure used in [83]. Equation (5.7) implies that the B-spline of order k in the i^{th} span is the weighted average of the B-splines of order k-1 on the i^{th} and $(i+1)^{th}$ spans, each weight being the ratio of the distance between the parameter x and the end knot to the length of the k-1 spans [115]. It should be noted that the matrix N depends only on the data points x_i, i = 1, ..., m+k and the extended knot set ξ_j, j = 1, ..., m+2k. Therefore, if the function values (ordinates) being approximated are changed and the knot set (abscissas) remains the same, then the matrix N does not need to be recomputed for each new function set.

Derivatives

A B-spline has locally linear segments smoothly (C^{k-2}) continuous embedded in them. A B-spline approximation is a local approximation scheme, and possesses

the property that the approximation is always smoother than the primitive function. It is possible to compute the derivatives $\Phi^j(x)$ of a B-spline for $j = 1, ..., k$ namely

$$\Phi^{j-1}(x) = \frac{d^{j-1}}{dx^{j-1}} \sum_{i=1}^{m+k} a_i N_{k,i}(x) \qquad , 1 \le j \le k$$

for a value of x in the range $\xi_k \le x \le \xi_{m+k+1}$. The theory of derivatives of B-splines is given by deBoor [83]. The derivatives can be obtained as follows:

$$\Phi^{j-1}(x) = \sum_{i=r}^{\gamma} a_{i,j} N_{k-j+1,i}(x) \qquad (5.8)$$

For any value of x we let γ be that integer value which satisfies the Schoenberg-Whitney condition, i.e. $\xi_\gamma \le x < \xi_{\gamma+1}$. Then $r = \gamma-k+j$, and the constants $a_{i,j}$ are calculated from the recurrence relation

$$a_{i,j} = (k-j+1)\frac{(a_{i,j-1} - a_{i-1,j-1})}{(\xi_{i+k-j+1} - \xi_i)}, \qquad j \ge 2 \qquad (5.9)$$

starting with

$$a_{i,1} = a_i .$$

5.2 Trajectory Generation

Once a trajectory has been fitted to a set of B-splines, the problem remains to adjust the time intervals between the knots so that the maximum velocity and acceleration of each joint is not exceeded. To do this we require the velocity and acceleration curves from the position curve of the B-spline approximation. When several trajectories consisting of a series of position points are to be spliced together into a smooth trajectory, it is necessary that the velocity and acceleration profiles

must be joined in such a way as to ensure a smooth operation of the robot through the entire trajectory.

If only one stand-alone trajectory is considered, a practical constraint that should be satisfied is

$\Phi'(\eta_1) = \Phi''(\eta_1) = 0$ (initial constraint)

$\Phi'(\eta_m) = \Phi''(\eta_m) = 0$ (final constraint)

which implies that the robot starts moving from rest and comes to a full stop at the end of the trajectory. Each of the path constraints can in general be set to any arbitrary constant. It should be mentioned that the jerk in this case is not constrained, i.e. it is allowed to have some arbitrary finite value at the end knots.

Time Dilation

Given the coefficients for velocity and acceleration at all data points, within an open interval (x_1, x_{m+k}), it is possible to ensure that the B-spline fit is a feasible one by adjusting the speed of operation, i.e. the travelling time for the robot. By adjusting the time intervals between each pair of adjacent knots it is always possible to obtain a feasible solution to the trajectory planning problem. This technique is referred to as **time dilation**, and may take two forms: uniform or nonuniform. Uniform time dilation is the easiest to implement, but nonuniform dilation results in a better solution, i.e. in a faster robot travelling time. We shall be concerned with uniform time dilation only. References to nonuniform time dilation, especially for generating minimum-time trajectories are in Chapter 4.

Let $\Phi_j(x)$ represent the position of a joint j in terms of the variable x. It is desired to replace x by the time variable t. Let

$$t = \lambda x \quad \text{or} \quad x = t/\lambda$$

where λ is the time dilation factor.

For the velocity of the joint,

$$\frac{d\Phi_j}{dt} = \frac{1}{\lambda} \frac{d\Phi_j}{dx} \tag{5.10}$$

For the acceleration of the joint,

$$\frac{d^2\Phi_j}{dt^2} = \frac{1}{\lambda^2} \frac{d^2\Phi_j}{dx^2} \tag{5.11}$$

The feasibility problem is then mathematically formulated as matching velocities and accelerations to prespecified constraint values for each of the links of the rigid body. The constraints are specified as:

$$\left|\Phi_j'(t)\right| \le VC_j \quad \text{(velocity)}$$

and

$$\left|\Phi_j''(t)\right| \le AC_j \quad \text{(acceleration)}$$

for all t and j = 1, ...,d. The time dilation factor λ can then be chosen in the following way:

$$\lambda_1 = \max_j \left(\max_x \frac{\left|\Phi_j'(x)\right|}{VC_j} \right) \tag{5.12}$$

$$\lambda_2 = \max_j \left(\max_x \frac{\left|\Phi_j''(x)\right|}{AC_j} \right) \tag{5.13}$$

with $\lambda = \max(\lambda_1, \sqrt{\lambda_2})$.

If a uniformly spaced B-spline set is used, the method for locating the maxima in equations (5.12-13) is trivial. If there exists some positive real number u, such that $\xi_{l+1} - \xi_l = u$ for all $k \le l \le m+k$ then $\{\xi_i\}$ is said to be a uniform vector, otherwise it is nonuniform. For the velocity profile, starting from zero, the approximate maxima occur at the odd knots and the approximate minima occur at the even knots. Similarly, for the acceleration profile, the approximate extrema are located at one half a knot behind the corresponding extrema of the velocity profile, i.e. when velocity crosses the x-axis. Examples of the curve extrema pattern are visualized by the examples presented in Chapter 7. In the next section where end point constraints are imposed on the B-spline curve, this simple and accurate search algorithm becomes inaccurate for the first and last k-3 knots. Due to the simple recursive method of evaluating derivatives using equations (5.8-9), no major difficulty is introduced by the constraints.

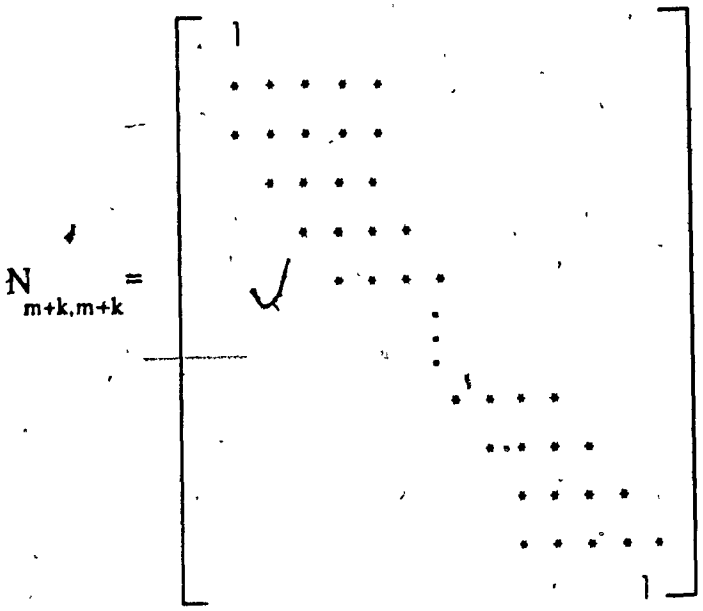
End Point Constraints

One method to achieve the end point constraints is to introduce a quintic polynomial at both ends of the B-spline curve. The quintic polynomial is C^4 continuous which permits position, velocity, and acceleration to be constrained to be continuous [1]. The polynomial is a parametric one in which some parameter is varied from 0 to 1 in order to traverse the polynomial curve. By calculating the derivatives of the B-spline curve at η_2 and η_{m+k-1} points, the quintic polynomial is able to merge with the Φ, Φ', Φ'' curves. However, there is wasteful oscillation occuring between the end knots at each end of the polynomial curve. A better way to satisfy the velocity and acceleration constraints at the end points is to incorporate the constraints into the B-spline linear equation (5.5), so that the coefficients a_i, $i = 1, ..., m+k$ automatically yield the desired end point conditions,

and maintain a homogeneous B-spline implementation. Barsky [118] also suggests setting up other ways of boundary and end point conditions.

The first and second derivatives of the B-spline curve are taken at the first and last knots, η_1 and η_m at which the velocity and acceleration are required to attain some specified value, say zero. The resulting equations for the derivatives can be manipulated into a form which allows them to be inserted into the linear equation (5.5).

Depending on the number of end point constraints, the approximate value of k (order of the B-spline) is chosen. The order must be sufficiently high to offer enough freedom to satisfy various constraints, but should be kept as low as possible to reduce the tendency of oscillation and to increase computational efficiency. The value of k affects the structure of the matrix N given by equation (5.6): as k increases, the width of the diagonal band increases. Based on the natural B-spline theory that has been presented, we obtain the matrix structure shown in equation (5.15) for k = 5 (quartic degree B-spline), which serves our purpose. The nonzero elements of N which correspond to knots are highlighted in bold.



$$N_{m+k,m+k} = \qquad (5.15)$$

In order to be able to insert the end point constraints and still ensure an exact unique solution for the a_i's, the resulting coefficient matrix must be nonsingular. For our purpose, we wish to set velocity and acceleration constraints for η_1 and η_m which means that four rows of (5.15) must be replaced by constraint equations. Four "dummy" knots are added to the original m knots to produce a set of m+k+4 linearly independent equations. It can be shown that proper placement of the "dummy" knots (with $k \geq 5$) ensures the nonsingularity of the coefficient matrix when the equations corresponding to the dummy knots are replaced by the constraint equations.

The placement of the dummy knots is critical to the proper evaluation of the constrained trajectory. Recalling that the knot points only are fitted, two dummy knots can be inserted before the first knot. To introduce two rows between rows three and four of (5.15), corresponding to knot η_1 for constraints $\Phi'(\eta_1)$ and $\Phi''(\eta_1)$. Introduce the $\Phi'(\eta_m)$ and $\Phi''(\eta_m)$ constraints, we place two dummy knots between knots η_{m-1} and η_m, resulting in two rows being added between rows m+k-3 and m+k-2 of (5.15). We insert them in the set of linear equations.

Starting with expression (5.8) for the derivatives of B-splines for k = 5 (quartic B-splines), we have

$$\Phi'(x) = \sum_{i=\gamma-3}^{\gamma} a_{i,2}\, N_{4,i}(x) \tag{5.16}$$

and

$$\Phi''(x) = \sum_{i=\gamma-2}^{\gamma} a_{i,3}\, N_{3,i}(x) \tag{5.17}$$

Expanding equation (5.16) yields

$$\Phi'(x) = a_{\gamma-3,2}\, N_{4,\gamma-3} + a_{\gamma-2,2}\, N_{4,\gamma-2} + a_{\gamma-1,2}\, N_{4,\gamma-1} + a_{\gamma,2}\, N_{4,\gamma} \tag{5.18}$$

Applying the relation of (5.9), we get

$$\Phi'(x) = \frac{4\,(a_{\gamma-3} - a_{\gamma-4})}{(\xi_{\gamma+1} - \xi_{\gamma-3})}\, N_{4,\gamma-3} + \frac{4\,(a_{\gamma-2} - a_{\gamma-3})}{(\xi_{\gamma+2} - \xi_{\gamma-2})}\, N_{4,\gamma-2}$$

$$+ \frac{4\,(a_{\gamma-1} - a_{\gamma-2})}{(\xi_{\gamma+3} - \xi_{\gamma-1})}\, N_{4,\gamma-1} + \frac{4\,(a_{\gamma} - a_{\gamma-1})}{(\xi_{\gamma+4} - \xi_{\gamma})}\, N_{4,\gamma} \tag{5.19}$$

The above expression can be factored as

$$\Phi'(x) = a_{\gamma-4}\left(\frac{-4\, N_{4,\gamma-3}}{\xi_{\gamma+1} - \xi_{\gamma-3}}\right) + a_{\gamma-3}\left(\frac{4\, N_{4,\gamma-3}}{\xi_{\gamma+1} - \xi_{\gamma-3}} - \frac{4\, N_{4,\gamma-2}}{\xi_{\gamma+2} - \xi_{\gamma-2}}\right)$$

$$+ a_{\gamma-2}\left(\frac{4\, N_{4,\gamma-2}}{\xi_{\gamma+2} - \xi_{\gamma-2}} - \frac{4\, N_{4,\gamma-1}}{\xi_{\gamma+3} - \xi_{\gamma-1}}\right)$$

$$+ a_{\gamma-1}\left(\frac{4\, N_{4,\gamma-1}}{\xi_{\gamma+3} - \xi_{\gamma-1}} - \frac{4\, N_{4,\gamma}}{\xi_{\gamma+4} - \xi_{\gamma}}\right) + a_{\gamma}\left(\frac{4\, N_{4,\gamma}}{\xi_{\gamma+4} - \xi_{\gamma}}\right) \tag{5.20}$$

Following a similar procedure for (5.17), we get

$$\Phi''(x) = a_{\gamma-2,3}\, N_{3,\gamma-2} + a_{\gamma-1,3}\, N_{3,\gamma-1} + a_{\gamma,3}\, N_{3,\gamma} \tag{5.21}$$

Expanding (5.21) and factoring, we get

$$\Phi''(x)\,(\xi_{\gamma+1} - \xi_{\gamma-2})\,(\xi_{\gamma+2} - \xi_{\gamma-1})(\xi_{\gamma+3} - \xi_{\gamma}) =$$

$$a_{\gamma-4}\left(\frac{12\,(\xi_{\gamma+2} - \xi_{\gamma-1})\,(\xi_{\gamma+3} - \xi_{\gamma})\,N_{3,\gamma-2}}{(\xi_{\gamma+1} - \xi_{\gamma-3})}\right) + 12\,a_{\gamma-3}\,(\xi_{\gamma+3} - \xi_{\gamma})$$

$$\left\{-(\xi_{\gamma+2} - \xi_{\gamma-1})\,N_{3,\gamma-2}\left(\frac{1}{(\xi_{\gamma+2} - \xi_{\gamma-2})} + \frac{1}{(\xi_{\gamma+1} - \xi_{\gamma-3})}\right)\right.$$

$$\left. + N_{3,\gamma-1}\frac{(\xi_{\gamma+1} - \xi_{\gamma-2})}{(\xi_{\gamma+2} - \xi_{\gamma-2})}\right\} + 12\,a_{\gamma-2}\,N_{3,\gamma-2}\left(\frac{(\xi_{\gamma+2} - \xi_{\gamma-1})\,(\xi_{\gamma+3} - \xi_{\gamma})}{(\xi_{\gamma+2} - \xi_{\gamma-2})}\right)$$

$$+ 12\,a_{\gamma-2}\,(\xi_{\gamma+1} - \xi_{\gamma-2})\left\{-(\xi_{\gamma+3} - \xi_{\gamma})\,N_{3,\gamma-1}\left(\frac{1}{(\xi_{\gamma+3} - \xi_{\gamma-1})} + \frac{1}{(\xi_{\gamma+2} - \xi_{\gamma-2})}\right)\right.$$

$$\left. + N_{3,\gamma}\frac{(\xi_{\gamma+2} - \xi_{\gamma-1})}{(\xi_{\gamma+3} - \xi_{\gamma-1})}\right\} + 12\,a_{\gamma-1}\,(\xi_{\gamma+1} - \xi_{\gamma-2})\left\{\frac{(\xi_{\gamma+3} - \xi_{\gamma})}{(\xi_{\gamma+3} - \xi_{\gamma-1})}\,N_{3,\gamma-1}\right.$$

$$\left. - N_{3,\gamma}\,(\xi_{\gamma+2} - \xi_{\gamma-1})\left(\frac{1}{(\xi_{\gamma+4} - \xi_{\gamma})} + \frac{1}{(\xi_{\gamma+3} - \xi_{\gamma-1})}\right)\right\}$$

$$+ a_{\gamma}\left(\frac{12\,(\xi_{\gamma+1} - \xi_{\gamma-2})\,(\xi_{\gamma+2} - \xi_{\gamma-1})\,N_{3,\gamma}}{(\xi_{\gamma+4} - \xi_{\gamma})}\right) \tag{5.22}$$

We can see from equations (5.20) and (5.22) that the velocity and acceleration constraint equations depend on the same a_i's as the position equations for the first and last knots. Inserting the constraint equations into (5.15) then yields a set of linear equations $N^0 \underline{a}^0 = \underline{f}^0$ of dimension m+k+4, with the matrix structure shown in equation (5.23). Dummy knot elements are represented by the character d, and the rows of N^0 corresponding to knots are highlighted in bold.

$$N_{m+k+4,m+k+4} = \left[\begin{array}{c} \mathbf{1} \\ \mathbf{\cdot\ \cdot\ \cdot\ \cdot\ \cdot} \\ \mathbf{\cdot\ \cdot\ \cdot\ \cdot\ \cdot} \\ \mathbf{d\ d\ d\ d} \\ \mathbf{d\ d\ d\ d} \\ \mathbf{\cdot\ \cdot\ \cdot\ \cdot} \\ \cdot\ \cdot\ \cdot\ \cdot \\ \cdot\ \cdot\ \cdot\ \cdot \\ \vdots \\ \cdot\ \cdot\ \cdot\ \cdot \\ \cdot\ \cdot\ \cdot\ \cdot \\ \mathbf{d\ d\ d\ d} \\ \mathbf{d\ d\ d\ d} \\ \mathbf{\cdot\ \cdot\ \cdot\ \cdot} \\ \mathbf{\cdot\ \cdot\ \cdot\ \cdot\ \cdot} \\ \mathbf{1} \end{array}\right] \qquad (5.23).$$

An algorithm for solving a linear system with a band-diagonal matrix has been implemented. The resulting routine requires storage of only those k*(m+k+2) elements of N^0 which are nonzero. It is worth noting that N^0 depends only on the extended knot set. Therefore, if these are fixed, the matrix N^0 does not change, so that any change in the desired trajectory $\underline{f}^0$, or end point constraints, can be carried out by resolving the linear system, without having to repeat the entire procedure for forming the matrix N^0.

It is worth noting that if constraints are to be placed at only one of the initial or final knots, then the order of the matrix can be reduced by zeroing the corresponding rows of N^o except for a one on the diagonal element(s) of the row(s), and a zero in the corresponding location(s) of f^o. It is also possible to use cubic B-splines (k = 4). Then only two dummy knots are needed; but, the complexity of maintaining a curve consisting of two different B-spline bases (k = 4, 5) may not be worth the effort this would impose on the software implementation.

The general procedure for constructing trajectories for multi-linked rigid bodies is summarized by the pseudo-code shown in figure 5.2, appropriate equations are given within brackets.

```
/*
** Function to generate a simple B-spline trajectory
*/
GenerateTrajectory( )
{
        /* Set up the constrained system matrix. */
        get number of knots in trajectory;
        form knot set;
        add dummy knots;
        form extended knot set;
        form constraint equations (5.20 and 5.22);
        form the matrix N^O (5.23);

        /* Approximate the trajectory for each joint with time dilation.*/
        WHILE (not all joints done) {
                get joint constraints;
                get joint trajectory;
                form right hand side vector;
                solve linear system for trajectory coefficients;
                FOR (complete knot set) {
                        find maximum velocity along trajectory using (5.8);
                        find maximum acceleration along trajectory using (5.8);
                        calculate time dilation factor for present joint;
                }
        }
        compute overall time dilation factor
}

/*
** Function to evaluate a simple B-spline trajectory.
** Note:  To move along the trajectory, only the extended knot set, the
** trajectory  coefficients, and the overall time dilation factor are required.
*/
EvaluateTrajectory ( )
{
        FOR (complete knot set) {
                WHILE (not all joints done) {
                        evaluate position on trajectory (5.5);
                        evaluate velocity on trajectory (5.8);
                        evaluate acceleration on trajectory (5.8);
                }
        }
}
```

Figure 5.2: Trajectory generation pseudo-code

5.3 Local Path Modifications

Two methods exist for creating local modifications to a B-spline trajectory. The first method has already been referred to as **tweaking**, a procedure where selected control vertices are perturbed by some amount to produce the desired deflection in the trajectory curve. The motivation for tweaking is that a new linear system of equations need not be solved every time a path modification is required. The second method allows an existing path to switch to any other path by **dynamic trajectory injection**. In this method a preplanned path is temporarily suspended while an injection trajectory is entered. When this is completed the control is transferred back to the original path. Both methods can be accomplished without relaxing any of the constraints or losing curve continuity.

From the band structure of N^O and equation (5.5), a following relation exists for any given f_j, $j = k+1, \ldots, m+k-1$:

$$f_j = a_{j-2} N_{k,j-2}(x_j) + a_{j-1} N_{k,j-1}(x_j) + a_j N_{k,j}(x_j) + a_{j+1} N_{k,j+1}(x_j) \tag{5.24}$$

By perturbing or tweaking certain a_j's in equation (5.24) from their calculated values for a preplanned trajectory, a corresponding deflection in f_j results. Again, by observing the matrix structure of N^O it is evident that each column in the strictly band matrix region has at most k non-zero elements; therefore, each a_j is involved in at most k linear equations. From this we can conclude that control vertex will affect at most k-1 adjacently neighboring knots along the path. The control vertex perturbation has greater effect on velocity and acceleration curves. From the

derivative recursive equations (5.8-9) we see that the velocity and acceleration profiles posess a greater interdependency on each control vertex. In addition, since new minima/maxima will occur over the tweaked interval, a rescaling of the tweaked knots is necessary to ensure feasibility of the solution. Many interactive computer-aided surface modeling systems display the actual location of control vertices with respect to the curve (surface) being approximated, so that the user is free to edit the curve (surface) by dragging a cursor about each control vertex (polygon). At this time no suitable interface exists in RIGID to produce local curve modification using control vertex tweaking.

Dynamic Trajectory Injection

If the evasive action involves backtracking or entirely new trajectories, a more adaptive approach is required than tweaking. It is possible to temporarily suspend the primary trajectory and enter a second injection trajectory that is capable of the evasive action, and then return control to the primary trajectory from where the detour began. By applying the theory presented in section 5.2, injection of feasible trajectories into the primary path is possible. A path segment jump table is used to contain information about the next segment to be executed and its relation to the primary segment. Depending on the details of the evasive action, the locations at which the robot departs and returns to the existing path are determined and registered in the jump table. In addition, the injection trajectory is independent of the primary path, except at its end points.

This property can be used to ensure that none of the physical constraints on the joints are violated. The respective scale factors must also be stored in the jump table. Following the procedure for end point constraints, the position, velocity, and acceleration at the end points of the injection trajectory are made equal to the departure and return knots of the primary trajectory. The greatest advantage of using B-splines is that only the coefficients of the injection trajectory need be calculated - none of the existing path coefficients are modified.

Since trajectory injection is only meaningful in an online setting, it has not been incorporated into RIGID. Although something quite similar exists, joining piecewise B-spline segments together and including a previously formed trajectory as part of a new trajectory. The implementation details of these are presented in the latter sections of Chapter 6.

Chapter 6 - RIGID Language Implementation Details

The fundamental objective in a robot programming language is to be able to accurately and realistically model the events of the real world. Discrepencies between internal and external reality result in a poor representation of the robot workspace and physically impossible movements.

Manipulator programs require the programmer to supply the initial conditions, specifications of robot and object placements and a general idea of how the manipulator should proceed through the task being programmed. Initial conditions have a great influence on the velocity and acceleration of the trajectory.

It is imperative to check the manipulator's reach during the course of the trajectory. A rigid body with sufficient degrees of freedom may be able to recover from an unreachable command by a configuration inversion, such as flipping the wrist link. RIGID allows the programmer to control the manipulator configuration.

RIGID is basically an offline programming system which allows the user to develop an application without involving the actual physical equipment. RIGID is capable of working with many different robot models.

6.1 Review of the C Programming Language

Every attempt has been made to make the user interface of RIGID as seamless and easy to use as possible, but using RIGID does require some knowledge of the C programming language. For this reason, some of the features of the C programming language are reviewed to assist in the explanation of RIGID. The C

language is well known for its portable and structured programming environment, and is the language in which the UNIX operating system is implemented [119]. The portability of RIGID is demonstrated by its implementation on three different hardware systems: IBM PC/AT (Microsoft C compiler ver. 4.0 [120]), MacIntosh (Think Lightspeed C compiler ver. 2.03 [121]) and DEC VAX 11/780 under UNIX 4.3BSD.

For the most part, the actual statements, commands for control flow and function calls are well explained in most C programming language text books [122-125]. Appendix B provides a summary of the C programming language syntax. Excellent accounts on data structure design and advanced algorithms can be found in [123, 126, 127]. The review provided here concentrates on data structures and complex C language declarations, to assist in explaining the RIGID user interface and language implementation details. In the examples some specific notation will be used: keywords and standard library function calls in the C programming language will appear in bold; explanations and program comments will appear within the C program comment characters (/* */).

Data Types

A comparison of the data types on IBM PC/AT, MacIntosh and DEC VAX 780 is provided. It is worth noting that the PDP-11 and the IBM PC/AT have the same word sizes for each of the C language data types.

Signed integers:	IBM PC/AT (bits)	MacIntosh (bits)	DEC VAX 780
int	16	16	32
long	32	32	32
short	16	16	16
Unsigned integers:			
char	8	8	8

Use of the keyword **unsigned** before a signed integer type declares the variable to be unsigned. This has the effect of doubling the positive integer range while not permitting any negative integers.

Floating point:			
float	32	32	32
double	64 ($10^{\pm 307}$)	80 ($10^{\pm 4932}$)	64 ($10^{\pm 38}$)

In the implementation of RIGID, all functions use double. The reasoning lies in the fact that C uses double in all of its internal floating point operations. In fact, there is often more computer overhead in converting to and from double to float than in consistently being double.

A data type may be declared **static**, meaning that its value is not to be destroyed once program control is passed outside the function from where the variable is declared. In a header file or before the main program, the static keyword allows variables to be initialized prior to program execution.

A variable may be specified as **register** to optimize its use. A register variable is usually assigned to a variable which is the counter in a loop and often appears as an index for an array. Register directs the compiler to contain the variable in a

computer register, rather than on the stack which needs to be loaded into a register before its value can be used by the program.

Arrays

An array is a collection of elements of a single data type. The array subscripts are each declared within square brackets ([]). Array subscripts are zero-based, which means that an array declaration of length n has subscripts which range from 0 to n-1. Multi-dimensional arrays are internally stored row-wise, unlike FORTRAN for example where arrays are internally stored column-wise. Example 1 is a six row by three column array of signed integers. Example 2 is a static array of length four floating-point numbers which are being preset to the values within the curly brackets. Example 3 is a character string whose dimension is being set to the length of the character string between double quotes, plus one more character for the NULL character which must terminate all strings.

1) **int** code[6][3];

2) **static double** vect[4] = {-0.7, 1.0, -2.3, 4.9};

3) **char** name[] = "platform_base";

Structures

A structure is a collection of elements of diverse data types. The **struct** keyword identifies what data type variables come next as a structure. Often, the **typedef** keyword is associated with structures to conveniently define new data type names. A **union** lets different data types use the same memory addresses. Unions are treated in much the same way as structures. Example 1 defines a type of structure containing a floating-point array of length four, a union of three variables spanning two different data types: **char** and **int**. Note that the data types

do not have to be the same length since the compiler will perform the byte alignment and padding neccessary for different data types. The **union** permits the user to access the data within the union option via the three methods shown in examples 4 through 6, but first a variable of type object_t must be created. Examples 2 and 3 declare variables of type object_t and initialize them to some constants. A careful look at the examples reveals that 2 and 3 are equivalent declarations since ascii(i) = 73. Examples 4 and 5 show how to access data without a union. Initializations like those of examples 2 and 3 must be performed outside the main program.

```
1)    typedef struct {
            double vect[4];
            union {
                  int AND, OR;
                  char id;
            } option;
      } object_t;

2)    object_t part1 = {1.0, 0.0, 0.0, 0.0, 73};
3)    object_t part2 = {1.0, 0.0, 0.0, 0.0, 'I'};
4)    part1.option.AND
5)    part1.option.id
```

Pointers

Pointers allow addresses to be used in a symbolic way. Pointers are especially efficient for arrays and structures. An array name is also a pointer to the first element of the array. The ampersand (&) is the address operator which obtains the address of a variable. The asterix (*) is the value operator which obtains the value contained at an address. For scalar variables, pointers are easy to demonstrate. Example 1 transfers the data in variable tag to atag using the intermediate step of using a pointer. Example 2 demonstrates the equivalence of using pointers. Example 3 shows alternative ways to access structure elements using pointers. Example 4 declares an array of pointers to **char** values. Example 5 declares a pointer to an array of **char** values. The next two examples demonstrate how

complex some data type declatations involving pointers can be. Example 6 declares an array of arrays of pointers to pointers to object_t data types, i.e. variable part is an array of five elements where each element is a five-element array of pointers to pointers to object_t structures. Example 7 declares an array of pointers to arrays of pointers to object_t data types, i.e. part is a five-element array of pointers to five-element arrays of pointers to object_t structures. Note the subtle differences in meaning between example pairs 4,5 and 6,7 by placement of brackets.

1)
```
int tag, *ptag, atag, **pp;

tag = 10;       /* initialize value of data */
ptag = &tag;    /* point to address of data */
atag = *ptag;   /* get value from address of data */
*pp = ptag;     /* pointer to pointer to address */
```

2)
```
int sizes[8][2], *psizes;

psizes = &sizes[0][0];
*(psizes + 1) equivalent to sizes[0][1]
*(psizes + 5) equivalent to sizes[2][1]
*(psizes + 10) equivalent to sizes[5][0]
```

3)
```
object_t *part;

part = &part1; /* part1 previously defined *.
part->vect[3] equivalent to part1.vect[3]
part->option.OR equivalent to part1.option.OR
```

4)
```
char *tzone[2];
char *tzone[2] = {"EST", "EDT"};
```

5)
```
char (*menu)[5];
char *s = "QUIT";
*menu = s;
```

6)
```
object_t **part[5][5];
```

7)
```
object_t *(*part[5])[5];
```

Dynamic memory allocation

Dynamic memory allocation provides the flexibility for data to be created and destroyed as needed at run-time. Structures and unions provide an excellent

framework for dynamic memory allocation of complex data associations such as linked-lists, trees and graphs. Since a pointer to a data is only a pointer and not the actual data, no memory is allocated for the data being pointed to until a directive to allocate memory is given. In some instances where two or more pointers point to the same data, memory needs to be allocated just once. The C language provides a couple of methods for doing this. The following example uses the **calloc()** function to allocate for memory for an object_t data structure, tests to verify successful completion of the **calloc()** function, then again destroys memory for the object using the **free()** function. A block of memory can also be expanded or compressed using the **realloc()** function. Character arrays or best allocated with the **malloc()** function, but **calloc()** is more suitable for integer or floating-point numbers because **calloc()** initializes the data values to zero.

```
object_t *part;

part = (object_t *) calloc (1, sizeof(object_t));
if (part == NULL)
        fprintf (stderr, "Can't allocate memory for object\n");
free (part);
```

Functions

Example 1 declares a function which has no return value. Example 2 declares a function which returns a long value. Example 3 declares a function that returns a pointer to an object_t data structure. Example 4 declares a five-element array of pointers to functions where each function has no return value, this declaration is especially useful for function menu-selection applications. Example 5 declares a function returning a pointer to an array of five double values. Note the differences that the placement of the brackets makes on the interpretation of the declaration.

1) **void** ExitPgrm();

2) **long** GetAddress();

```
3)      object_t *AllocateObject();
4)      void (*PrinterTypes[5])();
5)      double (*Evaluate ())[5];
```

Using dynamic memory allocation, the function corresponding to the declaration of example 3 can be realized as shown below. The function is generalized to allocate n consecutive object_t data structures. Notice the typecast operation to ensure that a positive number n is passed as the argument for the **calloc()** function call.

```
object_t *AllocateObject(n)
int n;
{
        object_t *part;

        part = (object_t *) calloc ((unsigned int) n, sizeof(object_t));
        if (part == NULL)
                return (NIL);
        return (part);
}
```

6.2 Programming with the RIGID Language

This section ties together the mathematical constructs presented in the previous chapters of the thesis and explains how they are unified and represented in the RIGID language library.

Frames

The term **frame** refers to the homogeneous coordinate transformation, introduced in Chapter 2. Frames are used to model objects in the world, or the work envelope of a rigidly-linked mechanism. Frames are specified by standard rotations and translations presented in many texts on geometry, such as [1,7]. Appendix C gives a summary of the formulas used by RIGID to transform frames.

Relations

A **relation** is a kinematic relationship used to affix frames to a closed kinematic chain. A relationship exists between two affixed frames, and remains a valid relationship even if either of the two affixed frames changes its location throughout the model of the world. Relations can have two purposes: to solve for an unknown relationship between any pair of known frames; to form macros which represent a grouping of frames for simplifying program coordinate specification in RIGID.

Trajectories

A **trajectory** is described by some displacement of frames under some form of optional velocity constraints at specified knots. Trajectory generation is available in two modes: Cartesian or joint-interpolated. In Cartesian mode, the joint space motions are obtained using the inverse kinematic solution of the robot arm each time the trajectory in Cartesian space is sampled. Joint-interpolated mode directly generates the trajectory in joint space so that no inverse kinematic solution is required at each sampling interval. Smooth and feasible trajectories are generated in the RIGID system in both cases using the B-spline theory presented in this thesis. The Cartesian mode generates robot trajectories which possess straight-line segments, but require more computing resources to complete; whereas, the joint-interpolated mode is more efficient kinematically, but makes the intermediate path segments of the robot harder to predict. A trajectory is highly sensitive to initial conditions, especially the initial robot arm position and orientation. The initial position of the arm also influences the velocity with which the arm will be moving during a segment. Increasing the speed introduces a different trajectory due to the mathematical nature of the derivative-constrained splining method used.

6.3 Implementation Details

Examples of the C programming language excerpts are highlighted in bold type. A full description of the RIGID function calls and data structures mentioned in the discussion is given in Appendix A.

Coordinate definition

Each coordinate function has a forward (***ffun[0]**) and inverse (***ffun[1]**) coordinate transform function pointer. The character string **cname** field is the description name of the coordinate transform function which is used to aid the interpretation of program output (**SIZENAME** is a constant). The **coord_t** data structure is:

```
typedef struct coord {
        char *cname;
        void  (*fun[2])();
} coord_t;
```

Frame definition

The most fundamental data type in RIGID is a coordinate frame, known as a **frame_t** data structure, used to describe some feature of an object. Each **frame_t** contains both the forward (**fwd**) and inverse (**inv**) homogeneous transformation matrices of the coordinate frame specified. Both relations and trajectories use data in the matrices **fwd** and **inv** when performing calculations involving the frame. The translation/rotation vector **vec** stores the transformation matrix equivalent of the **fwd** field. The character string **fname** field is the description name of the frame. Depending on the type of geometrical coordinate system (i.e.: RPY, EUL, OAT, ...) used to describe the coordinates of a frame, the **ffun** field maintains

pointers to the forward and inverse coordinate functions used to transform translation/rotation vectors into matrix form. The **frame_t** data structure is:

```
typedef struct frame {
        char fname[SIZENAME];
        double vec[6];
        double fwd[4][4];
        double inv[4][4];
        struct coord *ffun;
} frame_t;
```

Velocity constraint definition

The velocity constraint data type, known as a **velocity_t** data structure, has many fields similar to those of a **frame_t**. Trajectories point to the velocity data in the linear/angular velocity vector **vel** when imposing the velocity constraint to some trajectory segment. The **fname** field contains the description name of the velocity constraint. The **ffun** field maintains function pointers to the coordinate system used to transform linear/angular velocity vectors into matrix form. The **velocity_t** data structure is:

```
typedef struct velocity {
        char name[SIZENAME];
        double vel[6];
        struct coord *vfun;
} velocity_t;
```

Lists of lists

Relations and trajectories are implemented by various types of dynamically allocated linked lists of lists. The central idea of lists of lists is that each node of the primary list is a list itself. In such an application as RIGID the number of frames, contents of relations and size of trajectories cannot be predicted prior to run time; hence, an implementation of dynamic linked lists is the most efficient. List processing algorithms are best designed by pointers to data structures, rather than declaring a fixed number of arrays. A list is constructed using self-referencing data

structures, a data structure which contains an element that points to the address of the next data structure of itself, or node of a list. The lists used in RIGID are based on singly-linked lists, where each node in the primary list serves as a head node which has two pointers controlling the address bounds of the list - a pointer to the first and last node of the secondary list.

Building relations

In a relation, the primary list is referred to as the relation-list and the secondary list as the part-list. The relation-list is composed of **relation_t** data structures and the part-list is composed of **part_t** data structures. For each new relation introduced, a new node on the relation-list is created, this relation-list node serves as the header of the part-list. The **rdata** and **reol** fields of **relation_t** point to the beginning and end, respectively, of the part-list. The **nextrel** field points to the address of the next node of the relation-list.

The part-list uses the **part_t** data structure to catalogue the sequence of frames specified in the relation. The part-list is hidden from the user. The **part_t** data structure is designed so that the program can switch from one data type to another by following pointers and allowing the same generic operations to be performed on mixed data types, i.e.: frames and relations. This is accomplished using the **union pptr** which may address either the **relation_t** or **frame_t** data types. Depending on the **side** argument value of a call to either **EquateFrame()** or **EquateRelation()** the **side** field is assigned its value (either **RHS** or **LHS**). The integer **ptype** indicates which data type the **part_t** points to: a relation (**ptype = RELTYPE**) or a frame (**ptype = FRMTYPE**). The **nextpart** field points to the address of the next node of the part-list. The **part_t** data structure is:

```
typedef struct part {
```

```
        int side;
        int ptype;
        union {
              struct frame *frm;
              struct relation *rel;
        } pptr;
        struct part *nextpart;
} part_t;
```

The **relation_t** data structure is:

```
typedef struct relation {
        char rname[SIZENAME];
        struct part *rdata;
        struct part *reol;
        struct relation *nextrel;
} relation_t;
```

As can be seen by the nature of **part_t** and **relation_t**, they do not contain geometric data themselves; rather, they maintain addresses and sequencing information of frames and other recursive relations.

To further assist the explanation of the formation of complex lists, icon schematic diagrams of data structures are used. In most cases, the icons with their pointers labelled are the currently active data structures in the diagram. The top line in figure 6.1a represents a pointer to a data structure, the bottom line represents a NIL pointer, or terminating pointer. The **frame_t** icon in figure 6.1b has two small boxes labelled 'f' and 'i' which represent the forward and inverse transformation matrices, respectively. A **frame_t** icon may have its frame name written beside the icon. A **part_t** icon, shown in figure 6.1c, has two pointers of significance represented by two small boxes, the left box corresponds to the **nextpart** field and the right box to the **union pptr** field. The icon for a **relation_t**, depicted by figure 6.1d, has three pointers of significance shown as three small boxes. The leftmost box corresponds to the **nextrel** field, the middle box to **reol** field and the right box to the **rdata** field. Figure 6.1e shows a relation

inverter which is used to indicate that the relation is to be inverted upon its evaluation.

Figures 6.2a,b show the effect **OpenRelation()** has on the relation-list. No **part_t** data structures have been allocated yet. Figure 6.2a illustrates the case where the first new node in the relation-list is inserted, while figure 6.2b is the general case where a new node in the relation-list is inserted. In both cases, the two pointers to the part-list are NIL at this time.

When a call is made to either **EquateFrame()** or **EquateRelation()** a **part_t** data structure is allocated and added to the part-list; either the **rdata** or **reol** pointer of the currently active relation-list node gets updated. The pointer from the currently active part-list node is directed to the selected frame, or recursive relation. If the **side** argument of **EquateFrame()** is set to the right hand side (RHS), then a node is appended to the tail of the part-list and directs the **pptr.frm** pointer to the address of the frame. This can be seen in figures 6.3a,b. A left hand side (LHS) argument causes a node to be inserted at the head of the part-list and also directs the **pptr.frm** pointer to the address of the frame. This can be seen in figure 6.3c. The kinematic equations realized by figure 6.3c are

$$\text{THREE} * \text{PRES} = \text{ONE} * \text{TWO}$$
$$\text{PRES} = \text{THREE}^{-1} * \text{ONE} * \text{TWO}$$

where PRES is the unknown transformation matrix solution, or relation. The corresponding RIGID program segment for this example is:

```
OpenRelation(NULL);
    EquateFrame(LHS, THREE);
    EquateFrame(RHS, ONE);
    EquateFrame(RHS, TWO);
PRES = CloseRelation();
```

Figure 6.1 a-e: Relation icon definitions

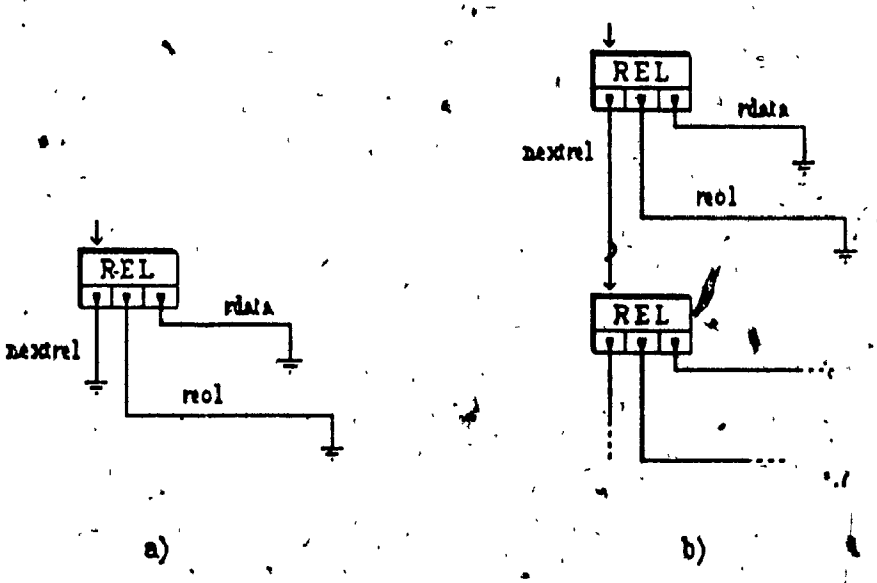


Figure 6.2a,b: OpenRelation() icon schematic.

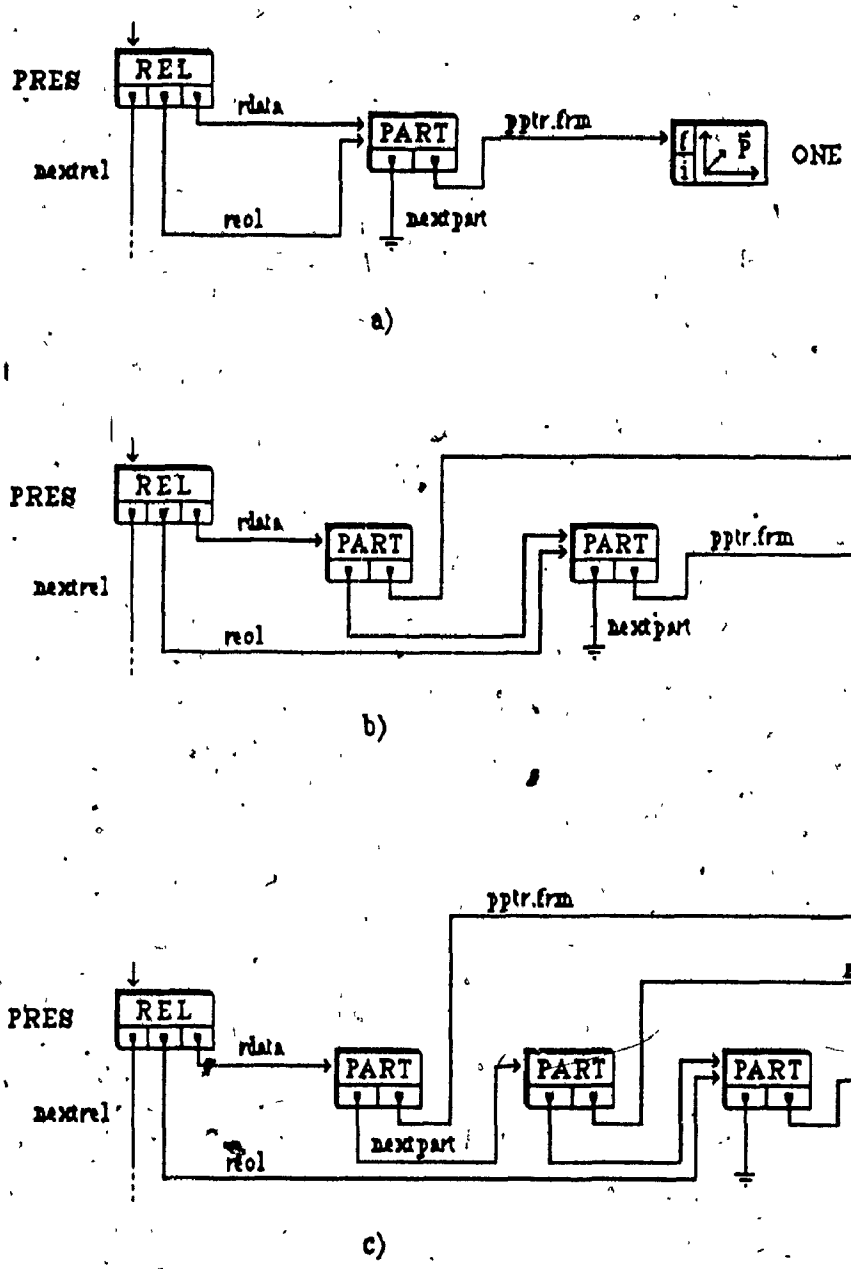


Figure 6.3a-c: EqualFrame() icon schematic.

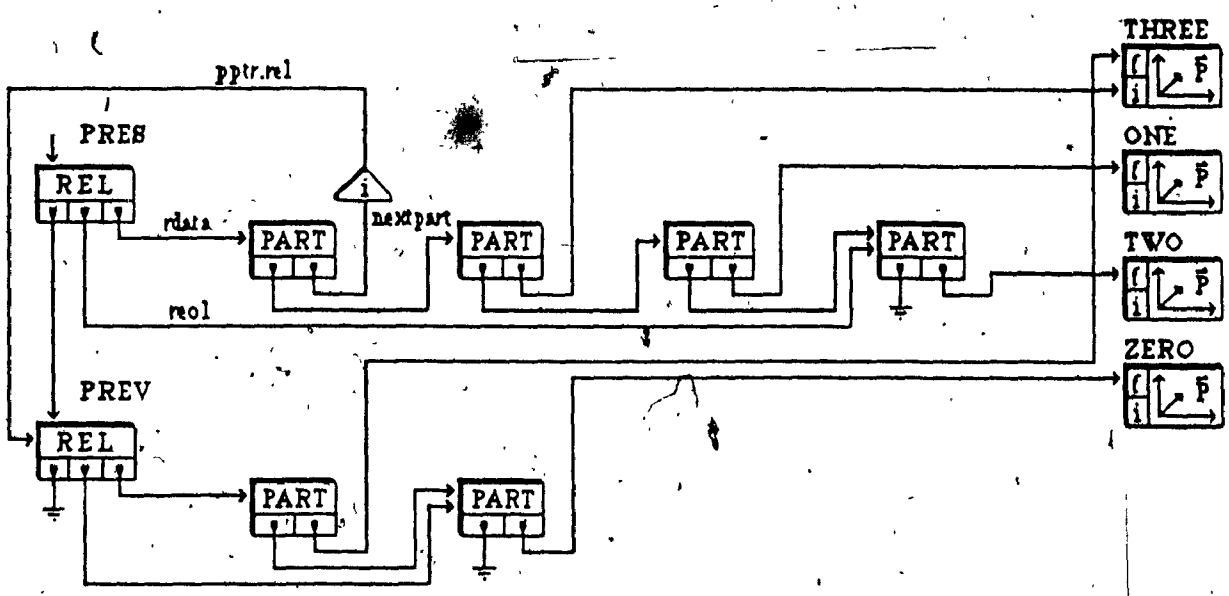


Figure 6.4: EquateRelation() icon schematic.

The icon schematic of figure 6.4 illustrates the effect of the **EquateRelation()** function on the part-list. The **pptr.rel** field in the currently active **part_t** data structure gets directed to the node in the relation-list corresponding to the recursive relation selected. The same RHS, LHS argument logic for appending/inserting part-list nodes applies when pointing to relations. The triangle with an 'i' inside indicates that the inverse of the relation is desired. When a solution for the PREV relation is computed, its solution matrix will be inverted and premultiplied to matrix $THREE^{-1}$. The kinematic equations realized by figure 6.4 are:

$$PREV = THREE * ZERO$$

and

$$THREE * PREV * PRES = ONE * TWO$$

which can be manipulated into the forms:

$$PRES = PREV^{-1} * THREE^{-1} * ONE * TWO$$

or

$$PRES = (THREE * ZERO)^{-1} * THREE^{-1} * ONE * TWO$$

The corresponding RIGID program segment is:

```
OpenRelation(NULL);
        EquateFrame(RHS, THREE);
        EquateFrame(RHS, ZERO);
PREV  = CloseRelation();
OpenRelation(NULL);
        EquateFrame(LHS, THREE);
        EquateRelation(LHS, PREV);
        EquateFrame(RHS, ONE);
        EquateFrame(RHS, TWO);
PRES  = CloseRelation();
```

Notice how the RIGID program expresses the mathematical formulation of the geometric problem in simple terms. A function call to **CloseRelation()** terminates the definition of the current relation by manipulating some global list building variables.

Evaluating relations

Once a relation has been formed it can be evaluated at any time by a call to **EvalRelation()**. The matrix solution for a relation is computed by traversing each node of the part-list and multiplying together the appropriate matrices of the frames pointed to by **pptr.frm**. If a node of the part-list points to another relation then **EvalRelation()** is recursively invoked on the data at the address pointed to by the **pptr.rel** field. The transformation matrix operations make use of the orthogonal nature of the homogeneous transformation matrices to manipulate frames, by employing such strategies as the matrix inversion method described in section 2.2, and multiplication involving only nonzero elements of each matrix.

Building trajectories

In a trajectory, the primary list is referred to as the trajectory-list, the secondary list as the segment-list, and the tertiary list as the knot-list. The resulting data structures form a linked list of lists of lists. The trajectory-list is composed of **trajectory_t** data structures, the segment-list of **segment_t** data structures, and the knot-list of **knot_t** data structures. For each new trajectory introduced a new node in the trajectory-list is created, this node becomes the header of the segment-list. The segment-list contains segment nodes which themselves contain pointers to a list of knots and one optional initial velocity constraint for the current segment. Therefore, each trajectory may consist of one or more segments where each segment is a sequence of knots optionally bounded by velocity and acceleration constraints. The intermediate segment-list is a necessary construct used to provide a systematic algorithm for organizing velocity constraints within any given sequence of knots. The **knot_t** data structure is:

```
typedef struct knot {
    int ktype;
```

```
        union {
            double *bpos;
            struct frame *frm;
            struct relation *rel;
            struct trajectory *traj;
        } kptr;
        struct knot *nextknot;
    } knot_t;
```

The segment_t data structure is:

```
    typedef struct segment {
        int numKnots;
        boolean isvel;
        boolean isacc;
        struct spline *path;
        int stype;
        union {
            double *bvel;
            struct velocity *vel;
        } sptr;
        double *bacc;
        struct knot *sdata;
        struct knot *seol;
        struct segment *nextseg;
    } segment_t;
```

The trajectory_t data structure is:

```
    typedef struct trajectory {
        char tname[SIZENAME];
        int numSegs;
        int mode;
        double dilation;
        struct segment *tdata;
        struct segment *teol;
        struct trajectory *nextraj;
    } trajectory_t;
```

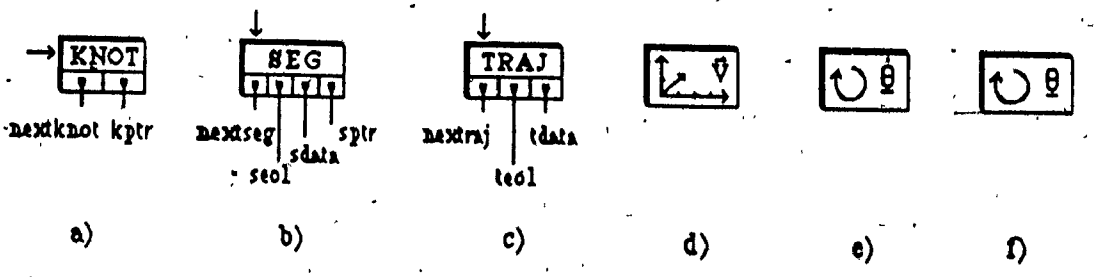


Figure 6.5a-f: Trajectory icon definitions.

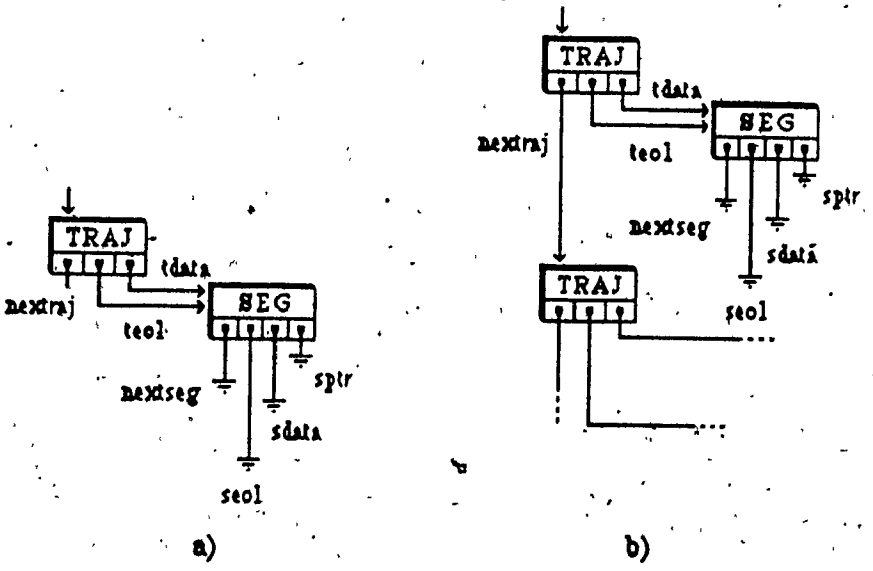


Figure 6.6a,b: OpenTrajectory() icon schematic.

Figures 6.5a-f define a new set of icons which are used to describe the trajectory building process. Figures 6.5a-c represent the **knot_t**, **segment_t** and **trajectory_t** data structures, respectively. A velocity constraint (**velocity_t**) is represented by figure 6.5d. Another type of velocity constraint, generated by the **BodyVelocity()** function, is depicted by figure 6.5e; similarly, a joint position vector, generated by a call to **BodyPosition()**, is shown in figure 6.5f. The acceleration constraint, generated by **BodyAcceleration()**, is not shown since its operation is identical to that of **BodyVelocity()**. The **tdata** and **teol** fields of these data structures are consistent with other data structures in that they point to the beginning and end of lists. Some icons may have labels, or description names associated with them.

The action of **OpenTrajectory()** is shown in figures 6.6a,b. One **segment_t** data structure is allocated and linked to the active trajectory-list node, forming the beginning of the segment-list. Figure 6.6b shows the general case where a new node in the trajectory-list gets inserted. Following a call to **OpenTrajectory()**, an optional proceding call to **SetVelocity()** will attach the **sptr.vel** pointer of the segment-list node to the velocity constraint matrix specified. A call to **BodyVelocity()** will attach the **sptr.bvel** pointer to the argument list vector of the **BodyVelocity()** function, and a call to **BodyAcceleration()** will attach the **bacc** pointer to the argument list vector of the **BodyAcceleration()** function. The **stype** field indicates which data type the **union sptr** points to: **velocity_t** data structure (**stype = FRMTYPE**), or joint velocity vector (**stype = BDYTYPE**). See figure 6.7a. Subsequent calls to either **MoveToFrame()**, **MoveToRelation()**, **MoveToTrajectory()** or **BodyPosition()** add new nodes to the knot-list. As figure 6.7b shows, if a repetitive call is made to

SetVelocity() within the current trajectory, then the current segment is terminated and a new node is appended to the segment-list, upon which the knot-list is subsequently built. Figures 6.8a,b show the similar list building effects for calls to the **BodyVelocity()** function; the **BodyAcceleration()** function is not shown since its operation is identical to that of **BodyVelocity()** except that the **bacc** pointer is manipulated. The **knot_t** is similar to the **part_t** data structure used in relation building, but has the extra provisions that the **union kptr** may be used to point to a joint position vector or a node in the trajectory-list. Correspondingly, the **ktype** field may indicate that the **knot_t** data structure points to either a joint position vector (**ktype = BDYTYPE**), or another trajectory (**ktype = TRAJTYPE**). Unlike the relation building process, it is only required that nodes in the knot-list are appended to each other monotonically in the order they are specified after the call to **OpenTrajectory()**. Examples of **MoveToFrame()** are given by figures 6.9a, b. For each new knot in the trajectory segment, a **knot_t** data structure is allocated and the appropriate pointer in the **union kptr** is assigned the address of the knot data. The integer **numKnots** contains the number of knots, or length of the knot-list, for the B-spline curves generated for this segment. The B-spline curves that are generated, at a later stage in the trajectory generation process, become stored in a **spline_t** data structure and are pointed to by the **path** field for this **segment_t**. Splines are explained in the next subsection.

For each **trajectory_t,** the attached segment-list and their knot-lists continue to be built by the appropriate function calls discussed above, until a terminating call to **CloseTrajectory()**, causing certain global list building variables to be reset for the next trajectory. Figures 6.10, 6.11 and 6.12 demonstrate the list building process for calls to **MoveToRelation()**, **MoveToTrajectory()** and **BodyPosition()**, respectively. During the call to **CloseTrajectory()** the B-

spline trajectory generation procedure is invoked for the trajectory and the B-spline curve data is stored in the **spline_t** structure, for each segment of the trajectory.

An important note concerning the trajectory building functions should be repeated here. After a function call to **OpenTrajectory()**, any of the **SetVelocity()**, **MoveToFrame()**, **MoveToRelation()** or **MoveToTrajectory()** functions can be intermixed until the terminating call to **CloseTrajectory()**. However, since the **BodyPosition()**, **BodyVelocity()** and **BodyAcceleration()** functions directly indicate joint displacements of a rigid-body through their argument lists, they may not be mixed with any other trajectory generating function calls within a given trajectory. Likewise, a trajectory formed with **BodyPosition()**, **BodyVelocity()** and **BodyAcceleration()** functions should not be included in any another trajectory specification using the **MoveToTrajectory()** function, unless the included trajectory has also been formed using only and **BodyPosition()**, **BodyVelocity()** and **BodyAcceleration()**.

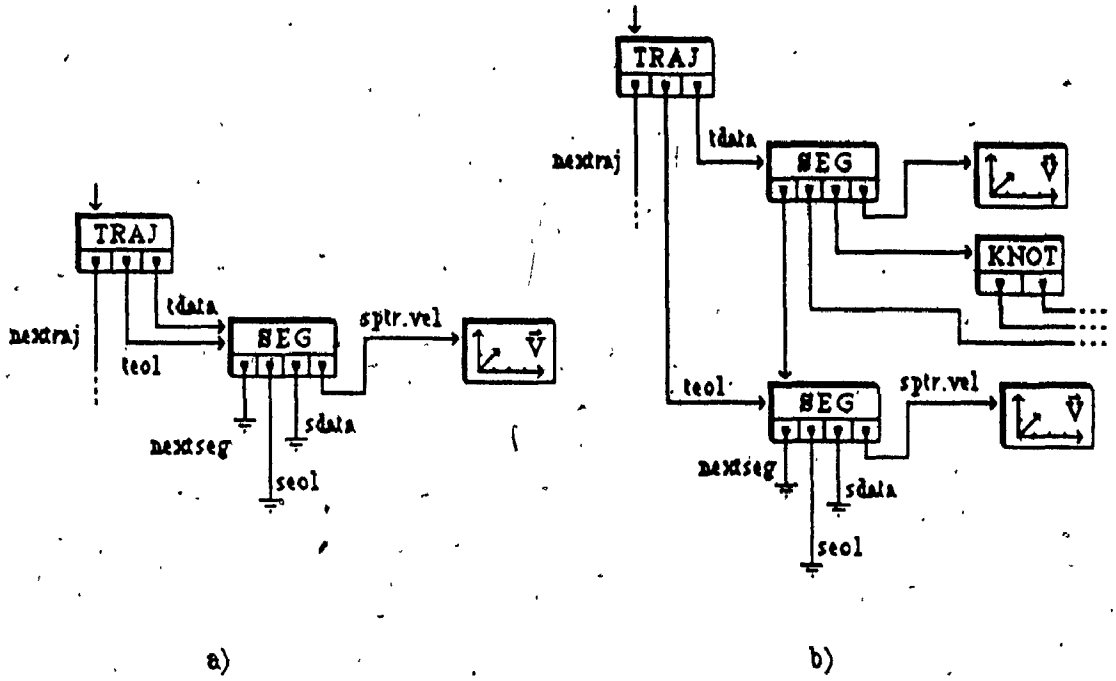


Figure 6.7 a, b: SetVelocity() icon schematic.

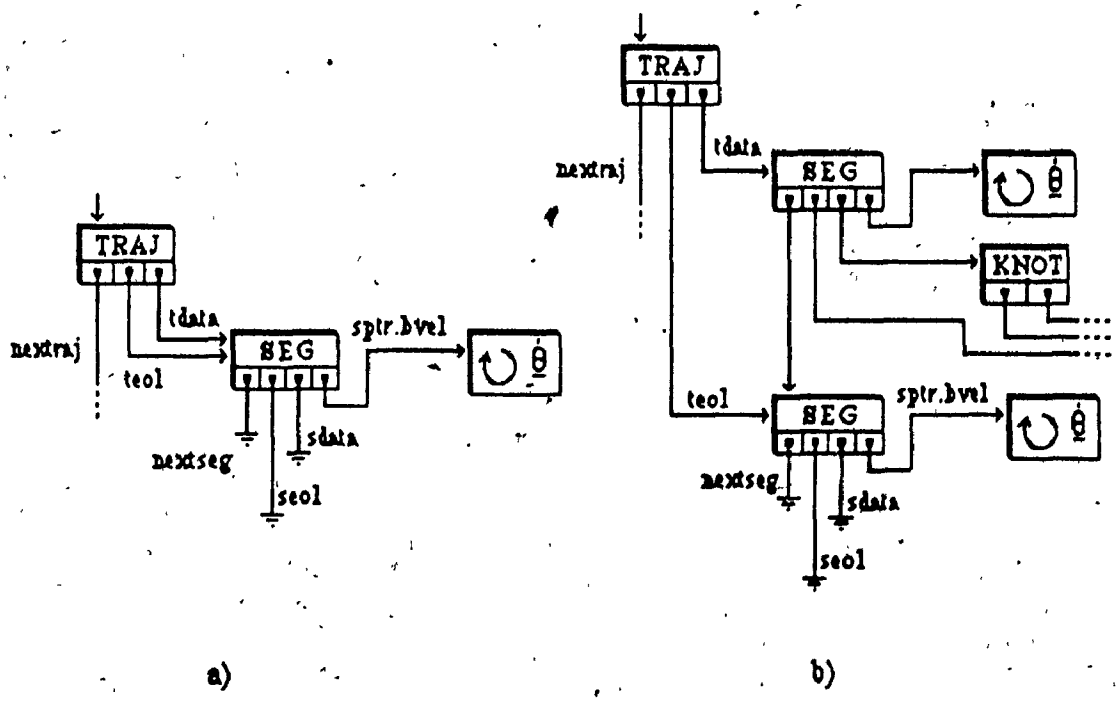
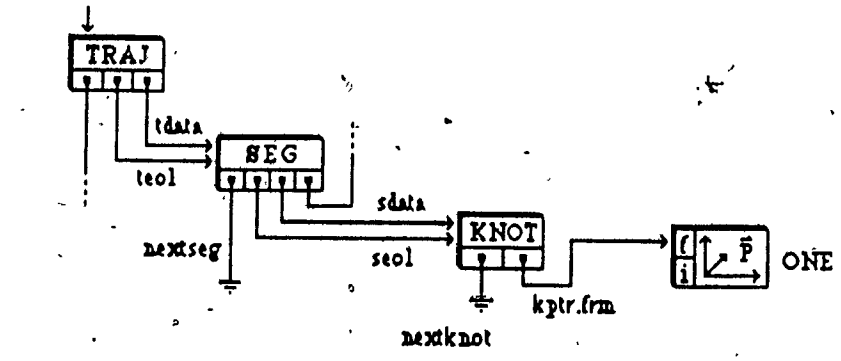
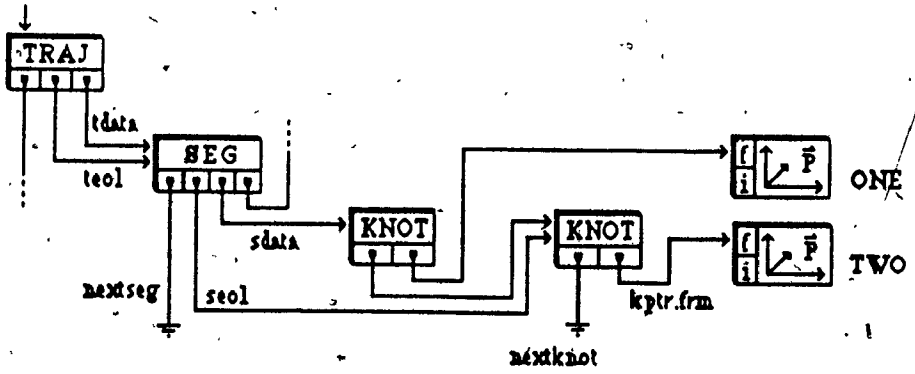


Figure 6.8 a, b: BodyVelocity() icon schematic.

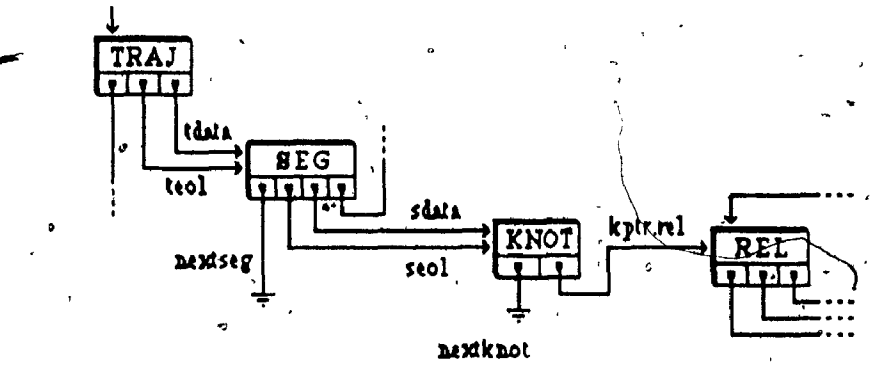


a)

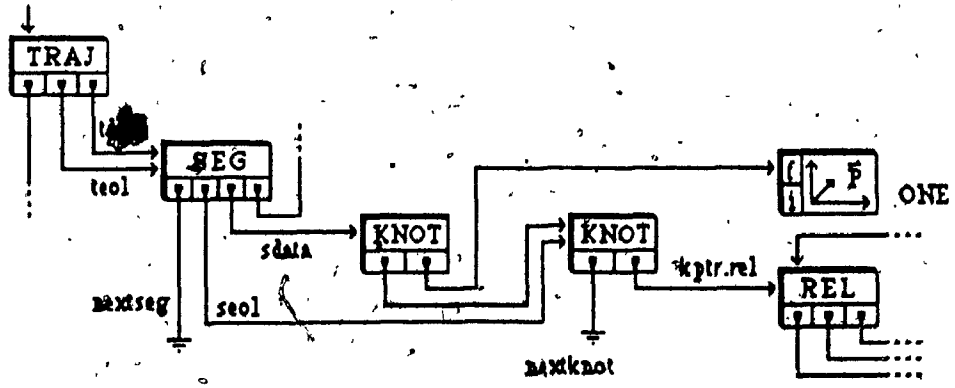


b)

Figure 6.9 a, b: MoveToFrame() icon schematic.



a)



b)

Figure 6.10 a, b: MoveToRelation() icon schematic.

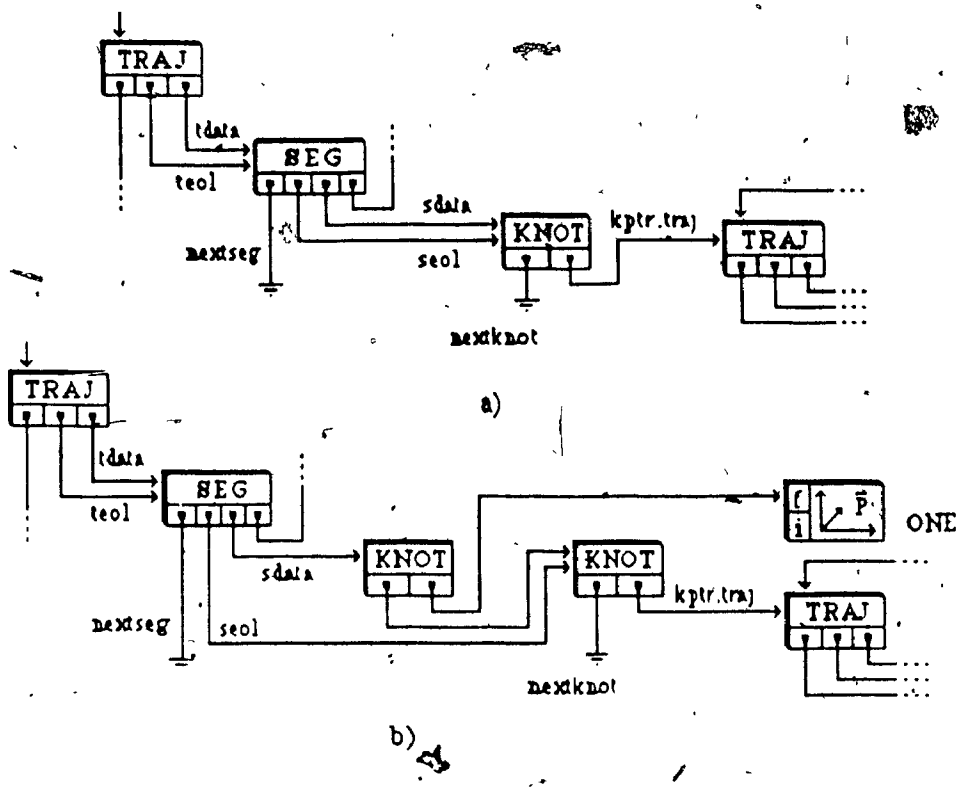


Figure 6.11 a, b MoveToTrajectory() icon schematic

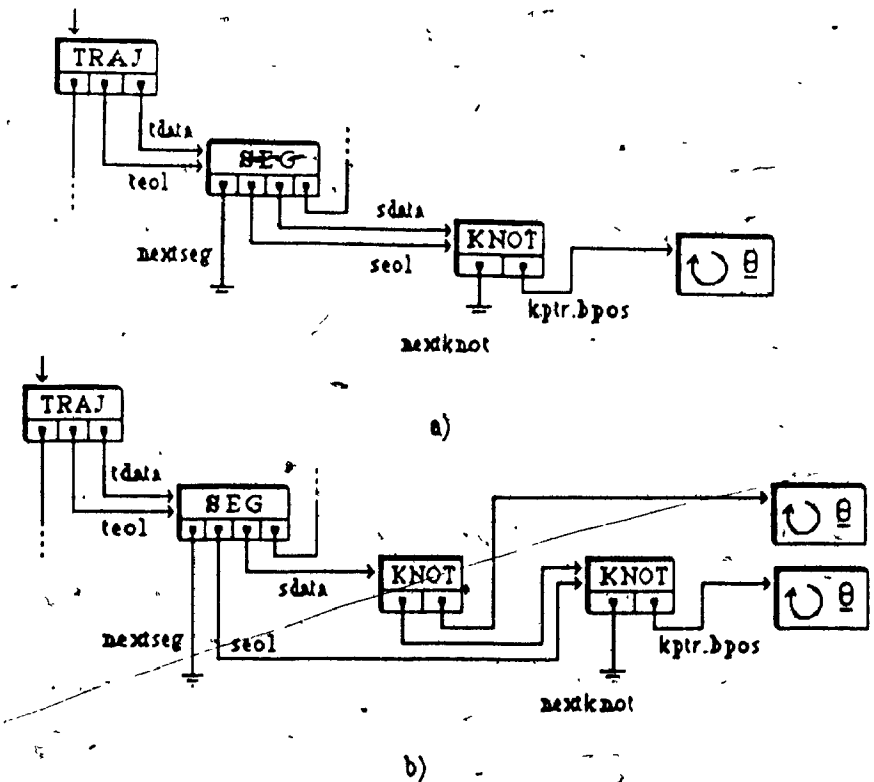


Figure 6.12 a, b: BodyPosition() icon schematic.

Building splines

A B-spline curve for a complete trajectory is generated by traversing each node of the segment-list and its branching knot-list nodes. For each segment-list node, a trajectory segment is constructed which spans all the knots of a knot-list using the B-spline construction algorithms presented in Chapter 5. The endpoint velocity constraints for segment i are obtained from the **union sptr** of **segment_t**$_i$ and **segment_t**$_{i+1}$ which bound the knots comprising the trajectory segment i; similarly the **bacc** field specifies the end point accelerations. By examining the **isvel** and **isacc** fields of the bounding segment-list nodes, the presence or absence of endpoint constraints for segment i is detected and relayed to the B-spline trajectory generation modules. RIGID automatically imposes constraints on adjacent segments of a trajectory if no constraints are set, to ensure a C^2 continuous B-spline curve throughout the trajectory.

A call to **EvalTrajectory()** begins a process which fits a trajectory to each segment. The knots from each segment are queued into a generic queue which provides the mechanism to dequeue and transform all the knots of a segment into a common coordinate system. The theory of queues is explained in many programming texts such as [123], so only a few implementation details will be given here. The queue implemented here is comprised of two data structures, **fifo_t** and **node_t** defined as follows:

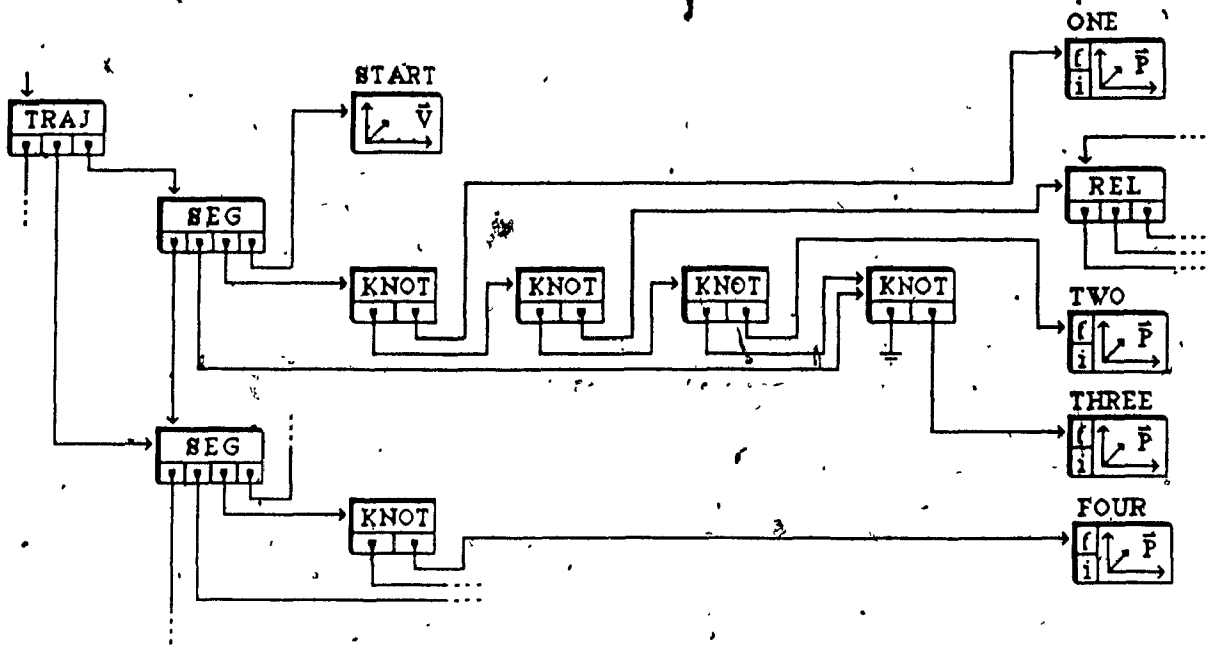
```
typedef struct fifo {
        int length;             /* length of queue */
        struct node *front;     /* front of queue */
        struct node *rear;      /* rear of queue */
} fifo_t;

typedef struct node {
        char *item;             /* enqueued data */
        struct node *nextnode;  /* next node in queue */
```

```
} node_t;
```

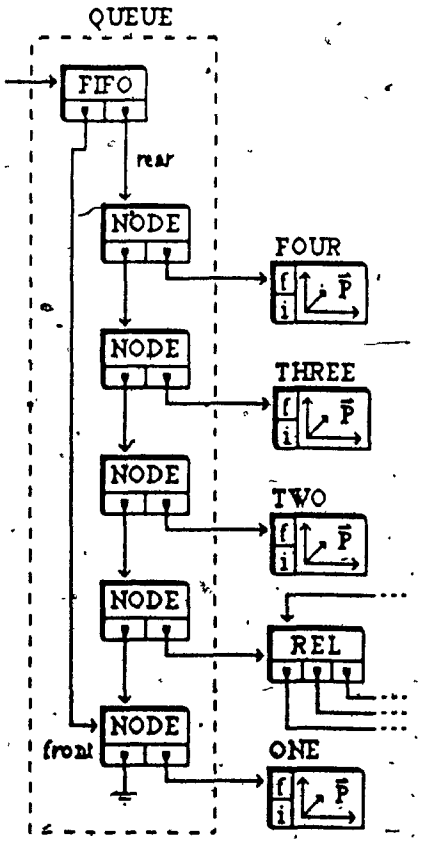
The **spline_t** data structure is:

```
typedef struct spline {
        int numKnots;
        int dof;
        int numPts;
        double *knots;
        double *coef;
        double *basis;
} spline_t;
```

B-spline curve generation is applied to a trajectory using a segment-by-segment algorithm. For segment i of a trajectory all **knot_t**'s pertaining to the segment i are queued. If there is a look-ahead **knot_t** into the next segment i+1, it is also queued at this time so that the last knot of segment i will smoothly join the first knot of segment i+1. The absence of a look-ahead knot indicates the last segment of the trajectory. With the exception of the first knot of the trajectory, if velocity constraints are imposed they are always effective on the look-ahead knot. In general, four combinations of velocity constraints are possible given any two segments (S_i, S_{i+1}), they are: (U, U), (U, C), (C, U), (C, C) where C = constrained, U = unconstrained. Figure 6.13 illustrates the action of the trajectory queue mechanism, note how the look-ahead knot of the next segment becomes the last knot of the current segment in the queue.



a)



b)

Figure 6.13 a, b: Action of trajectory queue.

After knot data is queued, the trajectory generation module dequeues each **knot_t** and transforms it into a set of coordinates for the rigid body selected. If the current knot has a velocity constraint, it is also transformed into the rigid body coordinate system. If a dequeued **knot_t** data structure points to a relation, the relation is evaluated and the resulting matrix solution becomes another knot in the segment. The complete segment is transformed into joint space, as shown in figure 6.14.

The transformation from world space task specification to joint space coordinates of the rigid body are accomplished using a **body_t** data structure. This structure contains all the necessary parameters and vectors to accomplish the forward/inverse kinematic and dynamic transformation. A particular rigid body (**body_t**) is assigned to a trajectory with the **SetBodyType()** function. The **body_t** fields contain: the degree of freedom (**dof**), the number of configuration parameters (**numConfig**), used to control redundant joint solutions, the configuration vector itself (**config**), the world space position, velocity and acceleration vectors (**p, v, a**), the joint space joint positions, velocities and accelerations (**qp, qv, qa**) and most importantly, the pointers to the kinematic transformation functions (**bfun.fun[0]** for the forward **bfun.fun[1]** for inverse kinematics). The physical joint velocity and acceleration limits of the rigid body are set using the **SetVelConstraint()** and **SetAccConstraint()** functions, respectively; the associated boolean flags (**isMaxVel** and **isMaxAcc**) indicate the presence of these constraints for the time dilation calculation. Memory is allocated to all the vectors in the **body_t** data structure during the **SetBodyType()** function call. The time dilation factor for a trajectory is stored in the **dilation** field in each

trajectory_t node and can be obtained prior to trajectory execution or evaluation by the **EnquireTrajectory()** function.

The **body_t** data structure is:

```
typedef struct body {
        int dof;
        int numConfig;
        int *config;
        double p[6], v[6], a[6];
        double *qp, *qv, *qa;
        struct coord bfun;
        boolean isMaxVel;
        double *maxVel
        boolean isMaxAcc;
        double *maxAcc;
} body_t;
```

Once the joint coordinates of each knot of a segment have been calculated, the trajectory generation algorithm presented in Chapter 5 is carried out for that segment. First, an extended knot set is formed, then the appropriate constraint equations which match the segment are inserted into the B-spline system matrix, and finally a set of B-spline coefficients is generated which describe the B-spline curves of each of the joints of the rigid body mechanism for the current segment. The time dilation factor is computed for the segment, and eventually for the entire trajectory. Referring back to the **spline_t** data structure, the B-spline coefficients are stored in **coef**, the extended knot set in **basis** and the original knot set in **knots**. The data in each **spline_t** structure is used when it is requested to execute the trajectory using functions such as **EvalTrajectory()** and **WriteTrajectory()**.

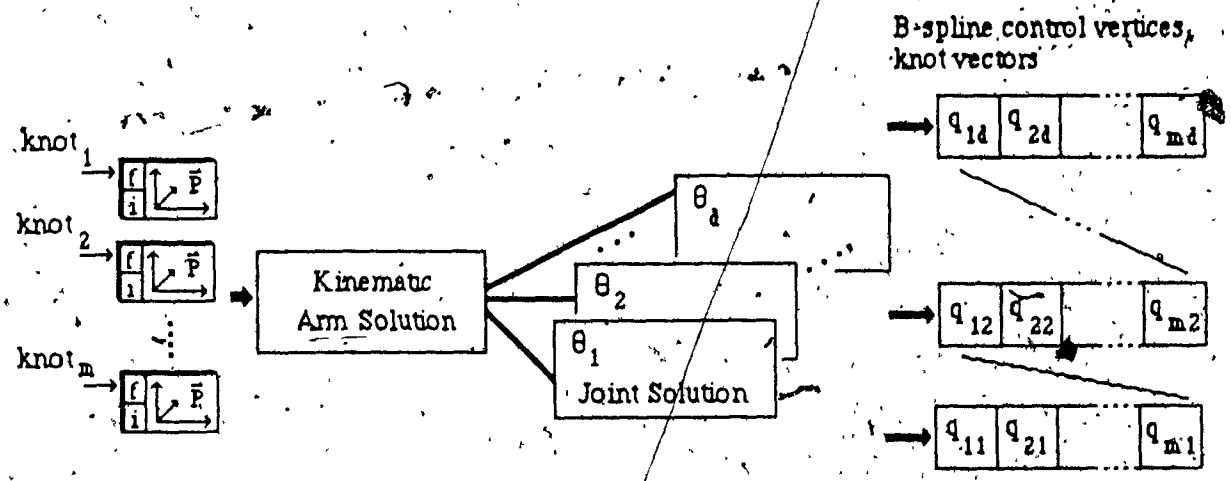


Figure 6.14: Transforming queued knots into spline_t structure.

Chapter 7 - Ilustrative Examples

Some examples are given in this chapter to illustrate the use of relations and the capabilities of trajectory generation for different types of manipulators. From experience, the development of an application in RIGID can best be summarized by the following four steps:

step 1 Define the problem using objects and establish object coordinate systems,

step 2 Plan piecewise motions in world space between coordinate systems and intermediate via points,

step 3 Select a rigid body and set its configurations, examine trajectories by checking for a sufficient frequency of knots,

step 4 Integrate all motions to get the complete operation.

7.1 Relation Examples

Relations are useful to describe features of an object such as those shown in figure 7.1 where the approach, stack and grasp coordinates for each block type object in a scene are the same. Two examples are given, example 7.1 demonstrates the use of a simple relation to group frames together, example 7.2 demonstrates a complex interaction to show how relations can be used to establish dynamic relations between object coordinates and feature coordinates.

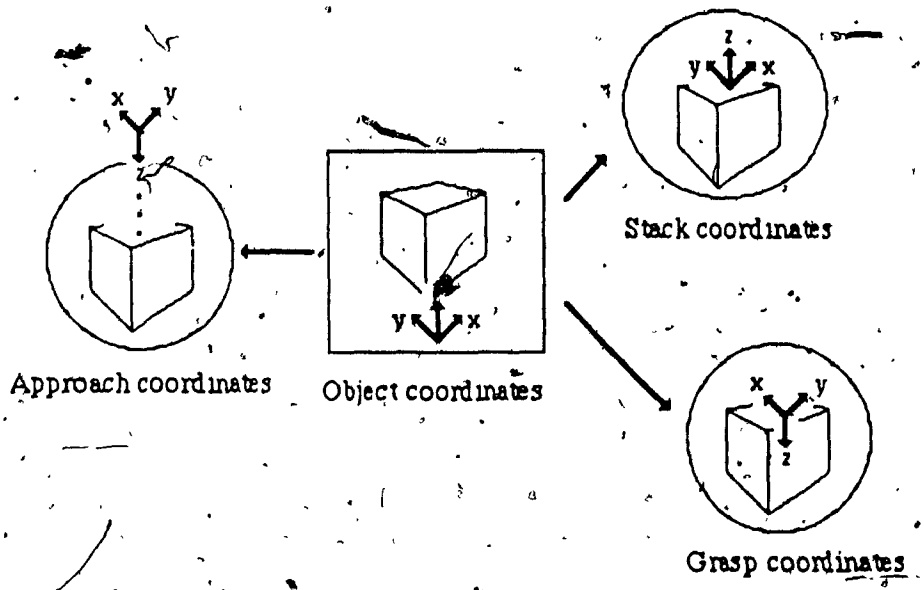


Figure 7.1: Object features.

Example 7.1: Grasp block B

A block is defined at coordinate frame **blockA**; the **grasp** transform represents the relative coordinates between a block and its grasping coordinates. Referring to figure 7.2 the task is to determine where to grasp block B from the conveyor belt. The frames of interest are:

world	the world coordinate system (not declared),
conv	coordinates of the conveyor belt with respect to **world**,
blockB	coordinates of block B relative to the conveyor belt,
grasp	grasping coordinates relative to any block definition.

In geometrical terms, the task of grasping block B can be written as:

obtainB = conv * blockB * grasp

where relation **obtainB** is the solution that expresses the grasping coordinates of block B with respect to world coordinates. The RIGID program and results for this task are given below. The program results are printed in both matrix and vector forms, a repetition of the input problem is printed and the solution to relation **obtainB** is given.

```
#include "rigid.h"

/*
** Determine where to grasp blockB;
*/
main()
{
frame_t *conv;              /* location of conveyor belt w.r.t. world */
frame_t *blockB;            /* location of block B w.r.t. conveyor belt */
frame_t *grasp;             /* relative grasping location w.r.t. block B */
relation_t *obtainB;        /* location where to grasp block B */

conv = BuildFrame("Conveyor", RPY, 80.0, 10.0, -20.0, 0.0, 0.0, 0.0);
blockB = BuildFrame("Block B", RPY, 15.0, 10.0, 5.0, 0.0, 0.0, 90.0);
grasp = BuildFrame("grasp at", RPY, 5.0, 5.0, 5.0, 180.0, 0.0, 0.0);

OpenRelation("obtainB");
        EquateFrame(RHS, conv);
        EquateFrame(RHS, blockB);
        EquateFrame(RHS, grasp);
```

```
obtainB = CloseRelation();

SetCoordSys(RPY);
WriteFrame("%6.2lf", 3, conv, blockB, grasp);
WriteRelation("%6.2lf", 1, obtainB);
SetOutputMode(VECTOR);
WriteFrame("%6.2lf", 3, conv, blockB, grasp);
WriteRelation("%6.2lf", 1, obtainB);
}
```

```
Conveyor (RPY)
forward transform:          inverse transform:
1.00  0.00  0.00 80.00      1.00  0.00  0.00 -80.00
0.00  1.00  0.00 10.00      0.00  1.00  0.00 -10.00
0.00  0.00  1.00 -20.00     0.00  0.00  1.00  20.00
0.00  0.00  0.00  1.00      0.00  0.00  0.00   1.00
Block B (RPY)
forward transform:          inverse transform:
0.00 -1.00  0.00 15.00       0.00  1.00  0.00 -10.00
1.00  0.00  0.00 10.00      -1.00  0.00  0.00  15.00
0.00  0.00  1.00  5.00       0.00  0.00  1.00  -5.00
0.00  0.00  0.00  1.00       0.00  0.00  0.00   1.00
grasp at (RPY)
forward transform:          inverse transform:
1.00  0.00  0.00  5.00       1.00  0.00  0.00  -5.00
0.00 -1.00 -0.00  5.00       0.00 -1.00  0.00   5.00
0.00  0.00 -1.00  5.00       0.00 -0.00 -1.00   5.00
0.00  0.00  0.00  1.00       0.00  0.00  0.00   1.00
obtainB (RPY)
0.00  1.00  0.00 90.00
1.00 -0.00 -0.00 25.00
0.00  0.00 -1.00 -10.00
0.00  0.00  0.00  1.00
Conveyor (RPY)
80.00  10.00 -20.00   0.00   0.00   0.00
Block B (RPY)
15.00  10.00   5.00   0.00   0.00  90.00
grasp at (RPY)
5.00   5.00   5.00 180.00   0.00   0.00
obtainB (RPY)
90.00  25.00 -10.00 180.00   0.00  90.00
```

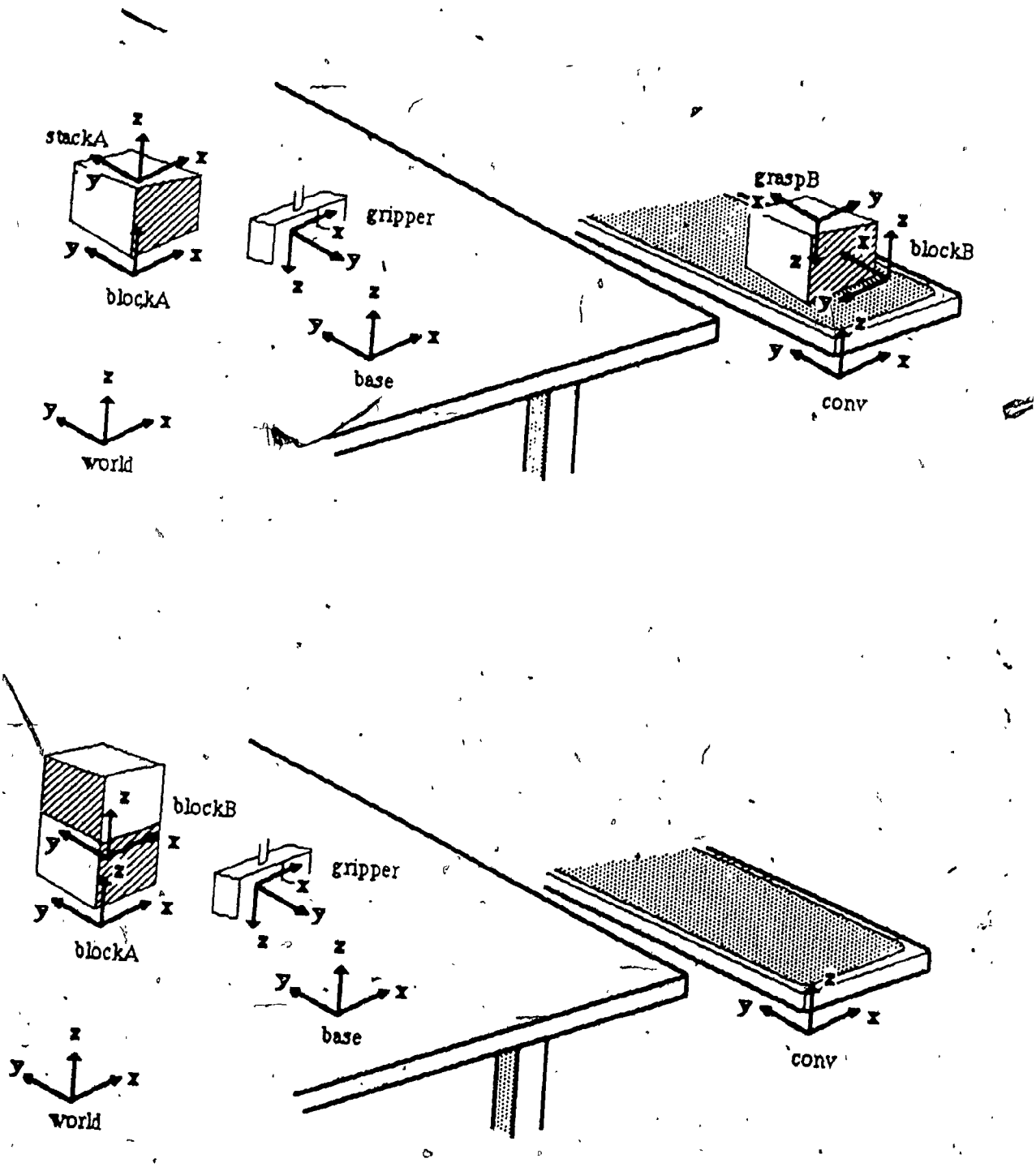


Figure 7.2: Block stacking scene.

Example 7.2: Robot stacking two blocks

Two blocks are defined at coordinate frames **blockA** and **blockB**, respectively. The **grasp** and **stack** transforms represent the relative coordinates between a block and its grasping and stacking coordinates. The frames of interest are:

world	the world coordinate system (not declared),
conv	coordinates of the conveyor belt with respect to **world**,
blockA	coordinates of block A with respect to **world**,
blockB	coordinates of block B relative to **conv**,
base	robot base coordinates with respect to **world**,
gripper	end of robot arm relative to robot gripping coordinates,
grasp	grasping coordinates relative to any block definition,
stack	stacking coordinates relative to any block definition.

By constructing relations between object features, task description primitives can be formed.

igrasp	inverse sense of grasping coordinates (**grasp**),
graspB	grasping coordinates of **blockB** relative to **blockB**,
stackA	stacking coordinates for **blockA** with respect to **world**,
stackBA	relative displacement between **blockB** and **stackA**,
depart	gripper coordinates to grasp **blockB** relative to **base**,
arm_depart	arm displacement to grasp **blockB** relative to **base**,
deposit	gripper coordinates to stack **blockB** on **blockA** relative to **base**,
arm_deposit	arm displacement to stack **blockB** on **blockA** relative to **base**.

The **arm_depart** and **arm_deposit** solutions represent the necessary displacements of the robot arm to achieve the correct coordinate placements, with the robot gripper taken into account.

To demonstrate the dynamic nature of frames and relations, the **blockB** coordinates are changed to simulate the advancement of the conveyor belt. Upon subsequent evaluation of any relations involving **blockB**, the results show that the

relations have been updated. For an equivalent mathematical analysis, the kinematic equations can be written as:

```
graspB = blockB * grasp
stackA = blockA * stack
conv * blockB * stackBA = stackA
base * depart = conv * graspB
arm_depart = depart * gripper
base * deposit = stackA * igrasp
arm_deposit = deposit * gripper
```

The driver program and results are given below.

```
#include "rigid.h"

/*
** Robot transfers block B from moving conveyor belt onto block A.
*/
main()
{
        static char *FMT = "%6.2lf";
        frame_t *conv, *blockA, *blockB, *base, *gripper;
        frame_t *grasp, *stack;
        relation_t *graspB, *stackA, *stackBA, *igrasp;
        relation_t *depart, *arm_depart, *deposit, *arm_deposit;

        /* Define objects. */
        conv = BuildFrame("Conveyor", RPY, 80.0, 10.0, -20.0, 0.0, 0.0, 0.0);
        blockA = BuildFrame("Block A", RPY, 30.0, 40.0, 20.0, 0.0, 0.0, 0.0);
        blockB = BuildFrame("Block B", RPY, 15.0, 10.0, 5.0, 0.0, 0.0, 90.0);
        base = BuildFrame("Robot base", RPY, 40.0, 0.0, 20.0, 0.0, 0.0, 0.0);
        gripper = BuildFrame("Robot gripper",RPY,0.0,0.0,-5.0,180.0,0.0,0.0);
        grasp = BuildFrame("grasp at", RPY, 5.0, 5.0, 5.0, 180.0, 0.0, 0.0);
        stack = BuildFrame("stack at", RPY, 0.0, 0.0, 10.0, 0.0, 0.0, 0.0);

        /* Define object relations. */
        OpenRelation("inverse grasp");          /* easy way to get frame inverse */
                EquateFrame(LHS, grasp);
        igrasp = CloseRelation();
        OpenRelation("grasp B");
                EquateFrame(RHS, blockB);
                EquateFrame(RHS, grasp);
        graspB = CloseRelation();
        OpenRelation("stack A");
                EquateFrame(RHS, blockA);
                EquateFrame(RHS, stack);
        stackA = CloseRelation();
        OpenRelation("block B to stack A displacement");
                EquateFrame(LHS, conv);
                EquateFrame(LHS, blockB);
```

```
        EquateRelation(RHS, stackA);
    stackBA = CloseRelation();

    /* Define tasks of robot. */
    OpenRelation("robot depart with block B");
        EquateFrame(LHS, base);
        EquateFrame(RHS, conv);
        EquateRelation(RHS, graspB);
    depart = CloseRelation();
    OpenRelation("arm depart motion");
        EquateRelation(RHS, depart);
        EquateFrame(RHS, gripper);
    arm_depart = CloseRelation();
    OpenRelation("robot deposit block B on block A");
        EquateFrame(LHS, base);
        EquateRelation(RHS, stackA);
        EquateRelation(RHS, igrasp);
    deposit = CloseRelation();
    OpenRelation("arm deposit motion");
        EquateRelation(RHS, deposit);
        EquateFrame(RHS, gripper);
    arm_deposit = CloseRelation();

    /* Print results. */
    SetCoordSys(RPY);
    SetOutputMode(VECTOR);
    printf("INPUT TASK:\n");
    WriteFrame(FMT, 4, blockA, conv, blockB, base);
    WriteRelation(FMT, 3, stackA, graspB, stackBA);
    printf("\nROBOT SOLUTION:\n");
    WriteRelation(FMT, 4, depart, arm_depart, deposit, arm_deposit);
    printf("\nCONVEYOR BELT ADVANCES BLOCK B:\n");
    UpdateFrameRel(blockB, 0.0, 20.0, 0.0, 0.0, 0.0, 0.0);
    WriteRelation(FMT, 2, arm_depart, arm_deposit);
}
```

```
INPUT TASK:
Block A (RPY)
30.00  40.00  20.00   0.00   0.00   0.00
Conveyor (RPY)
80.00  10.00 -20.00   0.00   0.00   0.00
Block B (RPY)
15.00  10.00   5.00   0.00   0.00  90.00
Robot base (RPY)
40.00   0.00  20.00   0.00   0.00   0.00
stack A (RPY)
30.00  40.00  30.00   0.00   0.00   0.00
grasp B (RPY)
10.00  15.00  10.00 180.00   0.00  90.00
block B to stack A displacement (RPY)
20.00  65.00  45.00   0.00   0.00 -90.00

ROBOT SOLUTION:
robot depart with block B (RPY)
```

50.00 25.00 -30.00 180.00 0.00 90.00
arm depart motion (RPY)
50.00 25.00 -25.00 -0.00 0.00 90.00
robot deposit block B on block A (RPY)
-15.00 45.00 15.00 -180.00 0.00 0.00
arm deposit motion (RPY)
-15.00 45.00 20.00 0.00 0.00 0.00

CONVEYOR BELT ADVANCES BLOCK B:
arm depart motion (RPY)
50.00 45.00 -25.00 -0.00 0.00 90.00
arm deposit motion (RPY)
-15.00 45.00 20.00 0.00 0.00 0.00

7.2 Trajectory Examples

Examples of C programming language excerpts are highlighted in bold type. A full description of the RIGID function calls are given in the RIGID reference library in Appendix A.

Example 7.3: Simple PUMA 560 trajectory

To demonstrate the most simple use of the trajectory generation, an example from Lin, Chang and Luh [100] for a PUMA 560 robot is used. Figure 7.3 lists the joint displacements for each of the eight knots of the trajectory. The robot is initially at rest, and comes to a full stop at the end of the trajectory. The trajectory is specified to the trajectory generator, given in the program listing below. A summary of the maximum joint velocity and acceleration limits for the PUMA 560 are provided in the Appendix D. The overall time dilation factor was computed to be 4.56. A closer examination of the joint profiles reveal that:

$$\left|\dot{\theta}_1^{max}\right| = 238 \text{ deg/sec} \qquad \left|\ddot{\theta}_1^{max}\right| = 658 \text{ deg/sec}^2$$

$$\left|\dot{\theta}_2^{max}\right| = 132 \text{ deg/sec} \qquad \left|\ddot{\theta}_2^{max}\right| = 365 \text{ deg/sec}^2$$

$$\left|\dot{\theta}_3^{max}\right| = 252 \text{ deg/sec} \qquad \left|\ddot{\theta}_3^{max}\right| = 634 \text{ deg/sec}^2$$

$$\left|\theta_4^{max}\right| = 307 \text{ deg/sec} \qquad \left|\ddot{\theta}_4^{max}\right| = 1455 \text{ deg/sec}^2$$

$$\left|\theta_5^{max}\right| = 153 \text{ deg/sec} \qquad \left|\ddot{\theta}_5^{max}\right| = 742 \text{ deg/sec}^2$$

$$\left|\theta_6^{max}\right| = 112 \text{ deg/sec} \qquad \left|\ddot{\theta}_6^{max}\right| = 291 \text{ deg/sec}^2$$

The overall time dilation factor is then obtained from $\lambda = \max\{(2.38, 3.82), (1.38, 3.02), (2.52, 2.90), (2.04, 4.56), (1.17, 2.87), (1.01, 1.90)\} = 4.56$. It is obvious that the acceleration profile of joint #4 is primarily responsible for the magnitude of the overall time dilation factor λ. The total travelling time for the feasible trajectory of the robot is therefore 32 seconds. The complete trajectory for each of the joints of the PUMA 560 robot is given in figures 7.4 a-f.

The simplicity of example 7.3 is useful to demonstrate some B-spline curve characteristics. The derivative boundary constraints introduce tension in the B-spline curves forming the robot trajectory, as shown in figures 7.5, 6 for two robot joints. Figures 7.5, 6 are not time dilated so that a comparison of B-splines with and without constraints can be compared. In general, the constrained B-splines curves oscillate more than the unconstrained curves between knots. It is also evident from figures 7.5, 6 that the acceleration profile lags the velocity profile by a 90^0 degree phase difference, and that curve maxima/minima occur at uniformally spaced intervals.

The driver program for example 7.3 is given below.

```
#include "rigid.h"

#define FMT             "%6.2lf"

main()
{
        trajectory_t *path;

        /* Application setup. */
        SetSampleStep(0.1);
```

```
SetMotionMode(JOINT);
SetOutputMode(VECTOR);
SetBodyType(PUMA560);
SetVelConstraint(100.0, 95.0, 100.0, 150.0, 130.0, 110.0);
SetAccConstraint(45.0, 40.0, 75.0, 70.0, 90.0, 80.0);

/* Build trajectory. */
OpenTrajectory("Luh path");
        BodyVelocity(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
        BodyAcceleration(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
        BodyPosition(10.0, 15.0, 45.0, 5.0, 10.0, 6.0);
        BodyPosition(60.0, 25.0, 180.0, 20.0, 30.0, 40.0);
        BodyPosition(75.0, 30.0, 200.0, 60.0, -40.0, 80.0);
        BodyPosition(130.0, -45.0, 120.0, 110.0, -60.0, 70.0);
        BodyPosition(110.0, -55.0, 15.0, 20.0, 10.0, -10.0);
        BodyPosition(100.0, -70.0, -10.0, 60.0, 50.0, 10.0);
        BodyPosition(-10.0, -10.0, 100.0, -100.0, -40.0, 30.0);
        BodyVelocity(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
        BodyAcceleration(0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
        BodyPosition(-50.0, 10.0, 50.0, -30.0, 10.0, 20.0);
path = CloseTrajectory();

/* Print results. */
printf("TOTAL %lf\n", EnquireTrajectory(path));
WriteTrajectory("%6.2lf", 1, path);
```

Joint Knot	1	2	3	4	5	6
	(degrees)					
1	10	15	45	5	10	6
2	60	25	180	20	30	40
3	75	30	200	60	-40	80
4	130	-45	120	110	-60	70
5	110	-55	15	20	10	-10
6	100	-70	-10	60	50	16
7	-10	-10	100	-100	-40	30
8	-50	10	50	-30	10	20

Figure 7.3: Joint space example

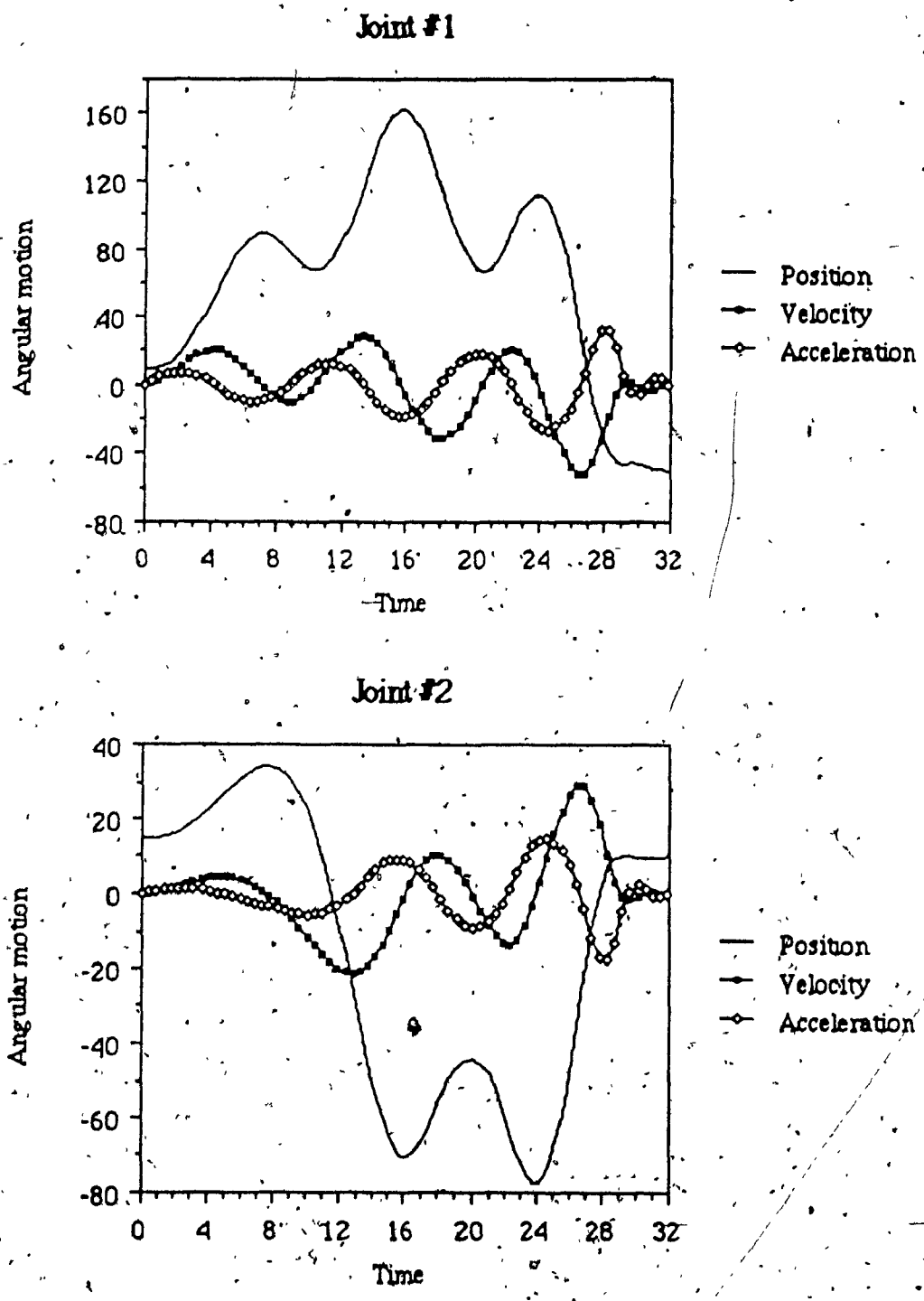


Figure 7.4a, b: Angular position, velocity and acceleration profiles.

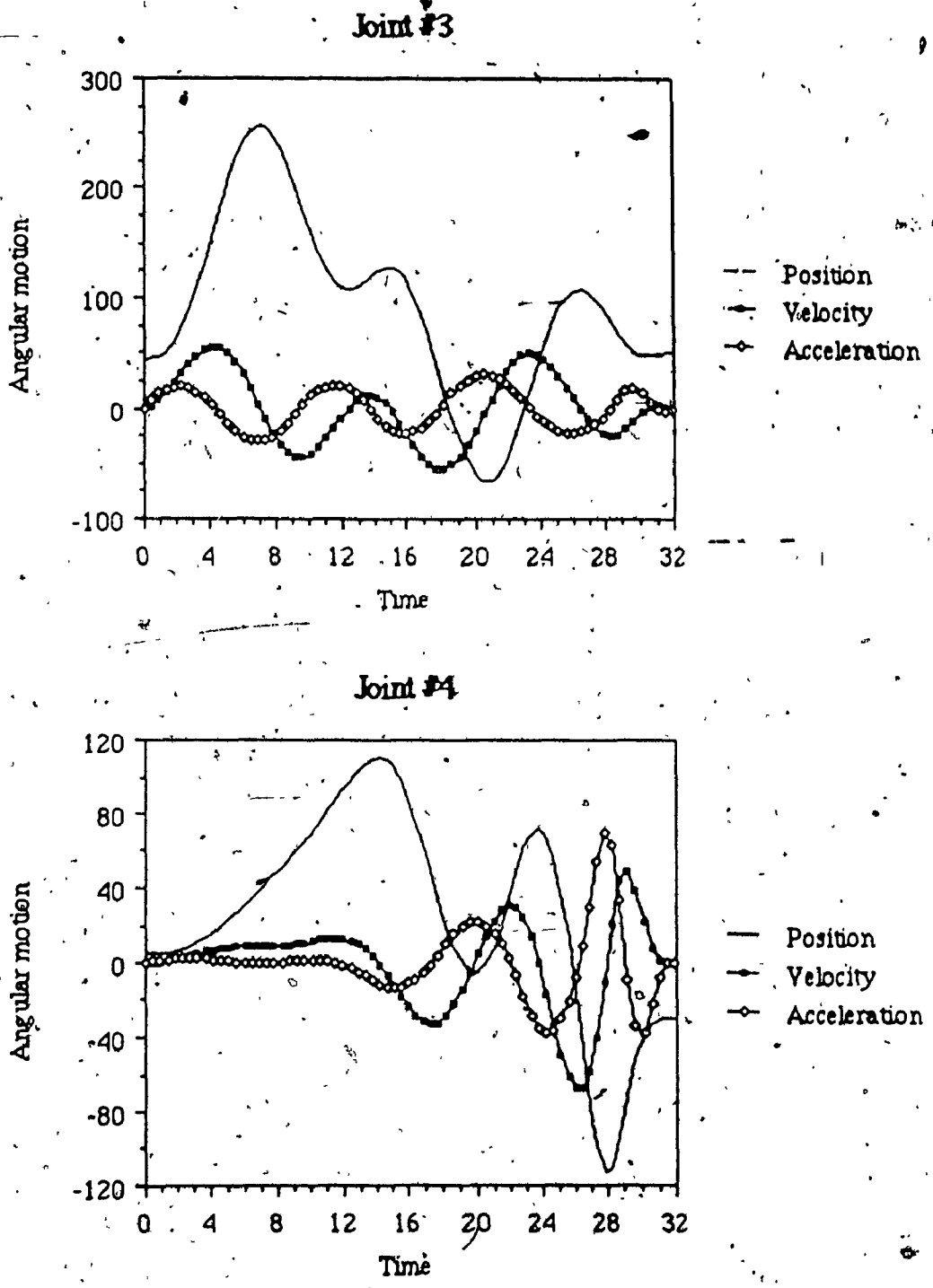


Figure 7.4c, d: Angular position, velocity, acceleration profiles.

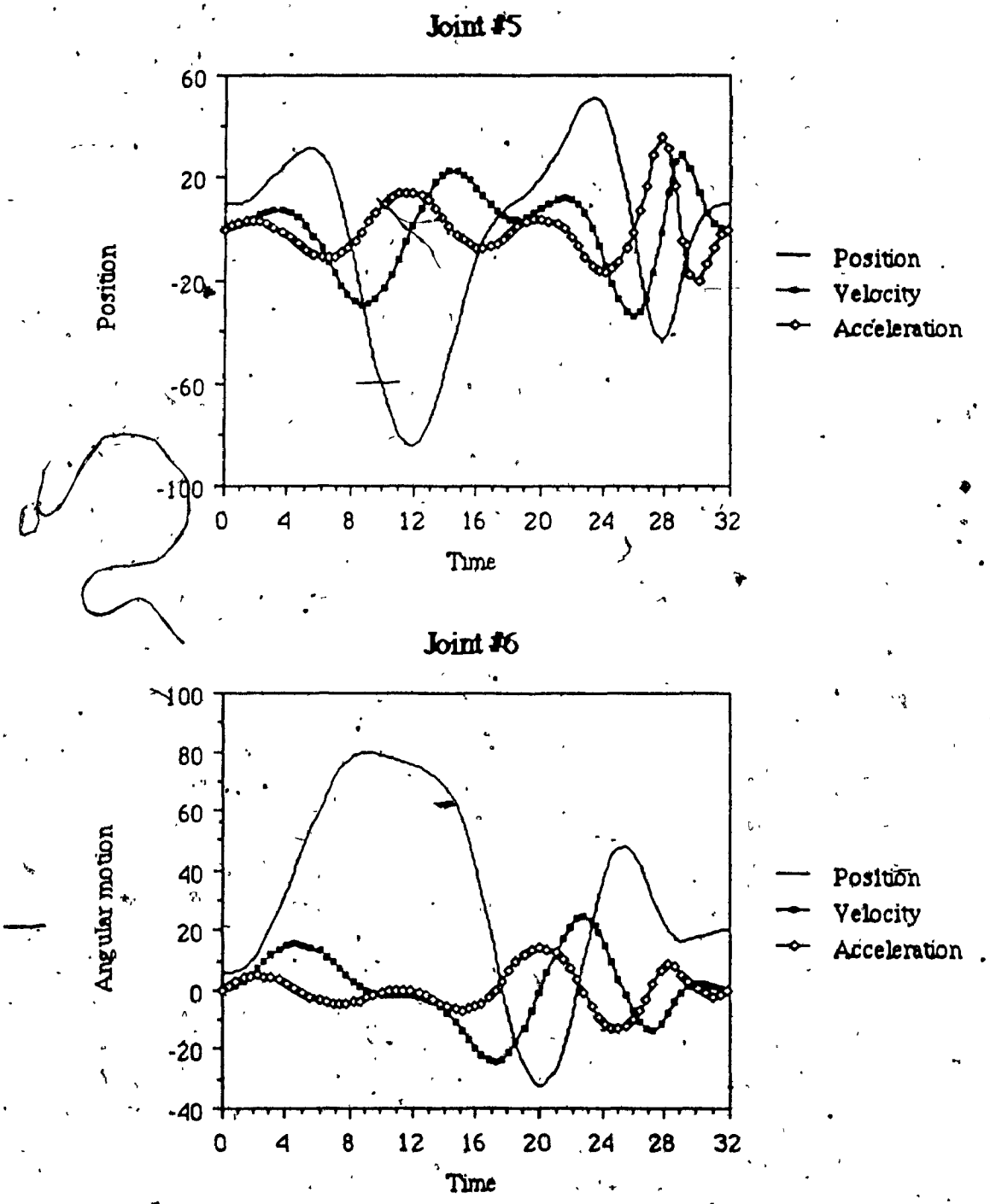


Figure 7.4e, f: Angular position, velocity, acceleration profiles.

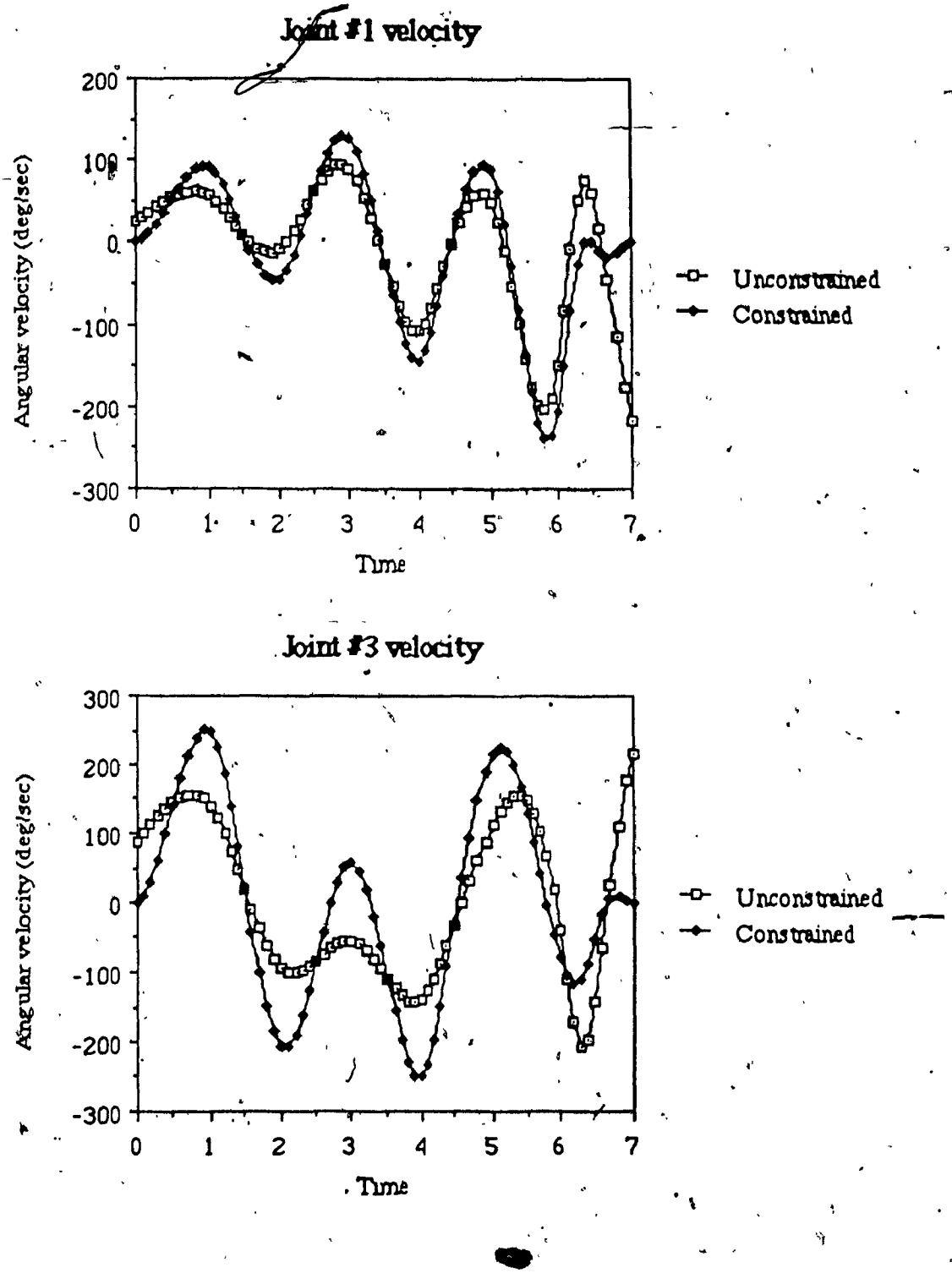


Figure 7.5: Constrained and unconstrained joint velocities.

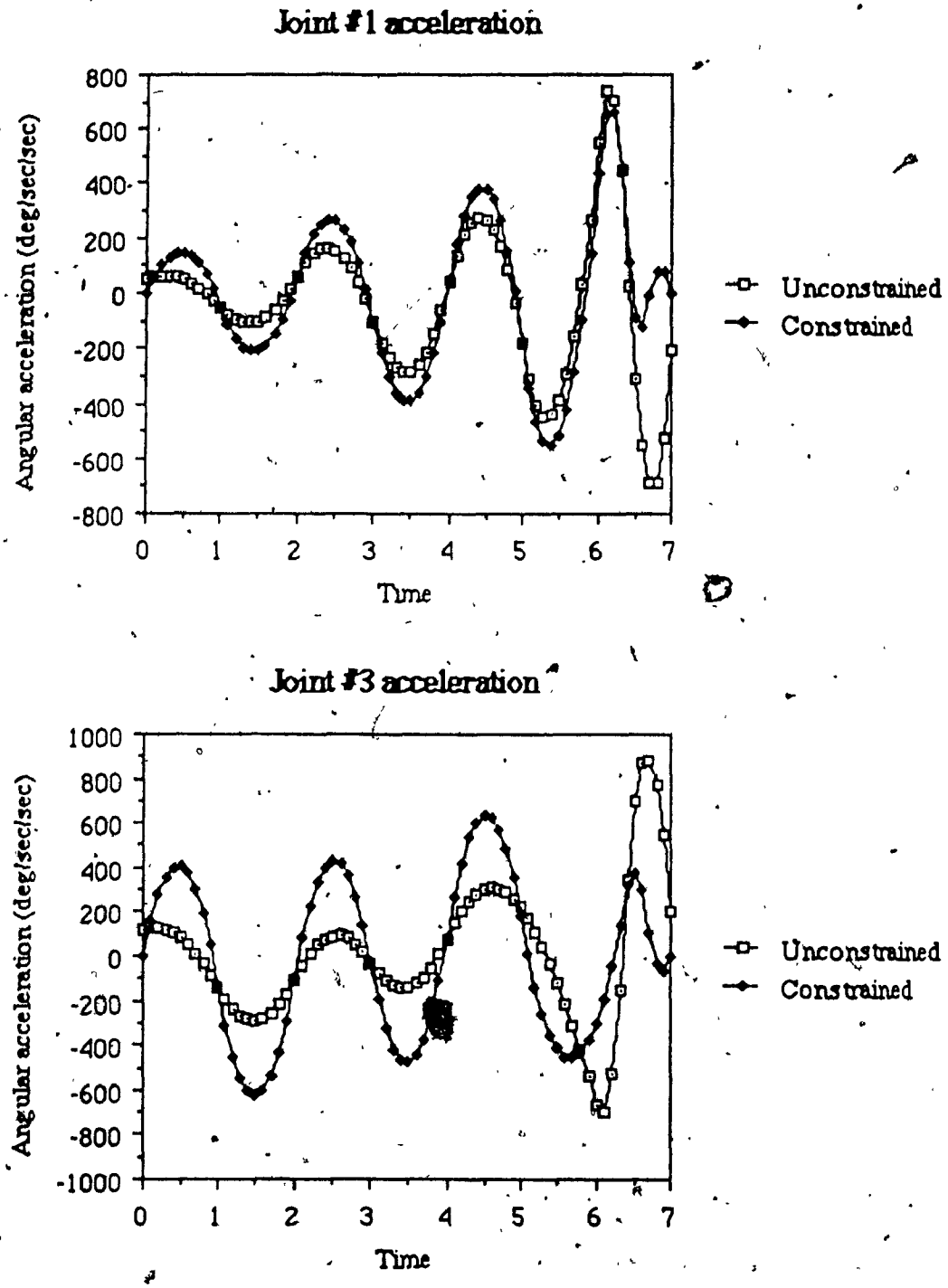


Figure 7.6: Constrained and unconstrained joint accelerations.

Example 7.4: Spray painting car body

This example simulates the motion of a PUMA 560 robot along the surface of a hypothetical car body, shown in figure 7.7. Eight points are selected, five forming a straight-line segment and three forming a curved segment with an initial constraint on velocity and acceleration. The desired robot configuration is a right shoulder, elbow up and flipped wrist. Time dilation is ignored in this example.

The eight frames are converted into joint displacements, then B-spline trajectories are generated for each joint of the PUMA 560 robot. When the joint space B-spline trajectories are evaluated, the B-spline derivatives (joint velocities and accelerations) are computed and plotted as shown in graphs in figure 7.8 e-f. To examine how well the robot inscribes the desired world space frames, the joint space trajectories are transformed back into Cartesian coordinates at a high sampling rate. The graphs in figure 7.9a,b depict the actual path of robot superimposed on the eight specified frame coordinates. Since the orientation vectors are maintained constant throughout the trajectory they are not shown. Figure 7.9a shows a ± 6.0 mm inscription error in the Y-axis. Figure 7.9b demonstrates a very small curve oscillation within the desired trajectory. The amount of inscription error is not very meaningful, since it is subject to robot configuration, initial conditions and frequency of knots within a particular trajectory.

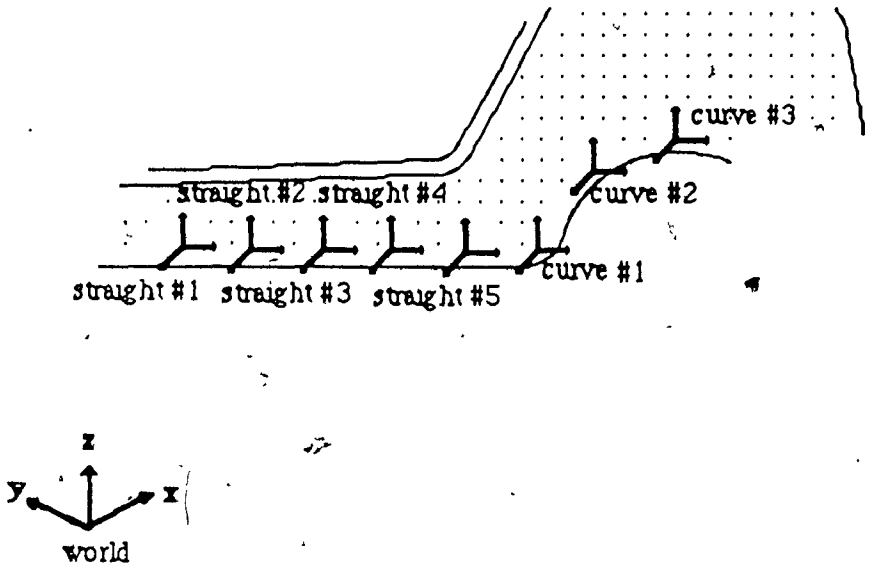


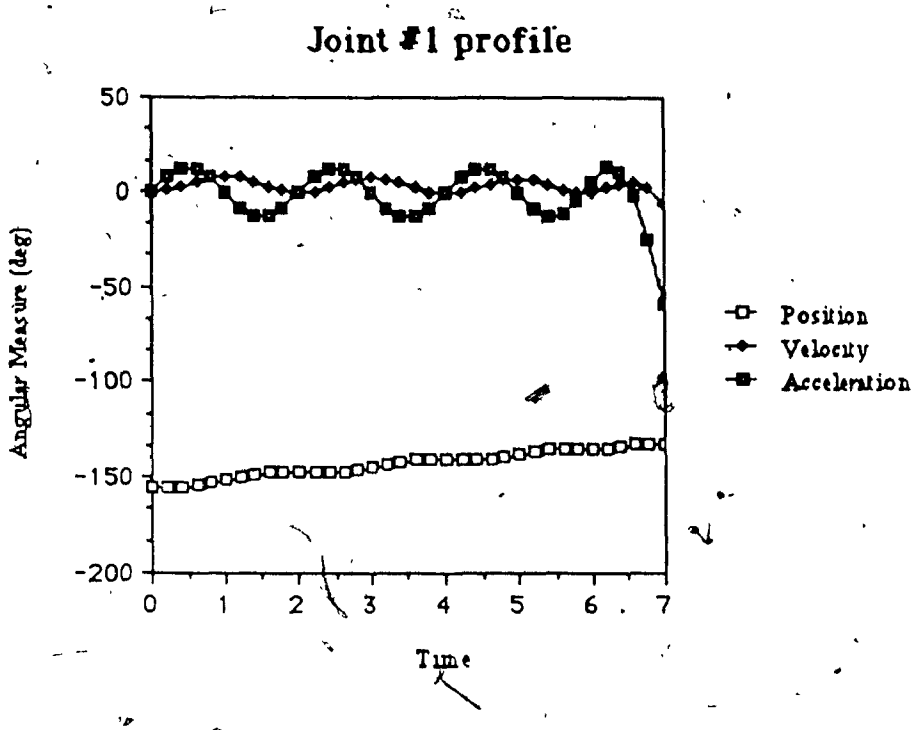
Figure 7.7. Spray paint car body.

The driver program and nongraphic results are given below.

```
#include "rigid.h"

#define         FMT    "%6.2lf"

void main(argc, argv)
int argc;
char *argv[];
{
        trajectory_t *paintPath;
        velocity_t *rest;
        frame_t *straight[5], *curve[3];
        int pt;
        double xt, yt, zt, inc;
        char name[SIZENAME];

        /* Application setup. */
        SetSampleStep(1.0);
        SetBodyType(PUMA560);
        SetMotionMode(JOINT);
        SetOutputMode(VECTOR);
        SetCoordSys(RPY);
        SetBodyConfig(-1, 1, -1);

        /* Generate straight-line part of path.*/
        xt = -150.0; yt = 600.0; zt = 85.0;
        for (pt = 0; pt < 5; pt++, xt -= 50.0) {
                sprintf(name, "straight #(%d)", pt + 1);
                straight[pt] = BuildFrame(name, RPY,xt,yt,zt,-50.0,50.0, 50.0);
        }

        /* Generate curved part of path. */
        xt = -400.0; yt = 600.0; zt = 85.0; inc = PI/4.0;
        for (pt = 0; pt < 3; pt++, xt -= 50.0) {
                sprintf(name, "curve #(%d)", pt + 1);
                curve[pt] = BuildFrame(name, RPY, xt, yt, zt + 50.0 *
                        sin(PI - pt * inc), -50.0, 50.0, 50.0);
        }
        rest = BuildVelocity("Robot at rest", RPY, 0.0,0.0,0.0,0.0,0.0,0.0);

        OpenTrajectory("Car Paint Path");
                SetVelocity(rest);
                for (pt = 0; pt < 5; pt++)
                        MoveToFrame(straight[pt]);
                for (pt = 0; pt < 3; pt++)
                        MoveToFrame(curve[pt]);
        paintPath = CloseTrajectory();

        /* Print results. */
        WriteFrame(FMT, 5, straight[0], straight[1], straight[2], straight[3],
```

```
        straight[4]);
    WriteFrame(FMT, 3, curve[0], curve[1], curve[2]);
    WriteTrajectory(FMT, 1, paintPath);
}
```

```
straight #(1) (RPY)
-150.00 600.00  85.00 -50.00  50.00  50.00
straight #(2) (RPY)
-200.00 600.00  85.00 -50.00  50.00  50.00
straight #(3) (RPY)
-250.00 600.00  85.00 -50.00  50.00  50.00
straight #(4) (RPY)
-300.00 600.00  85.00 -50.00  50.00  50.00
straight #(5) (RPY)
-350.00 600.00  85.00 -50.00  50.00  50.00
curve #(1) (RPY)
-400.00 600.00  85.00 -50.00  50.00  50.00
curve #(2) (RPY)
-450.00 600.00 120.36 -50.00  50.00  50.00
curve #(3) (RPY)
-500.00 600.00 135.00 -50.00  50.00  50.00
```



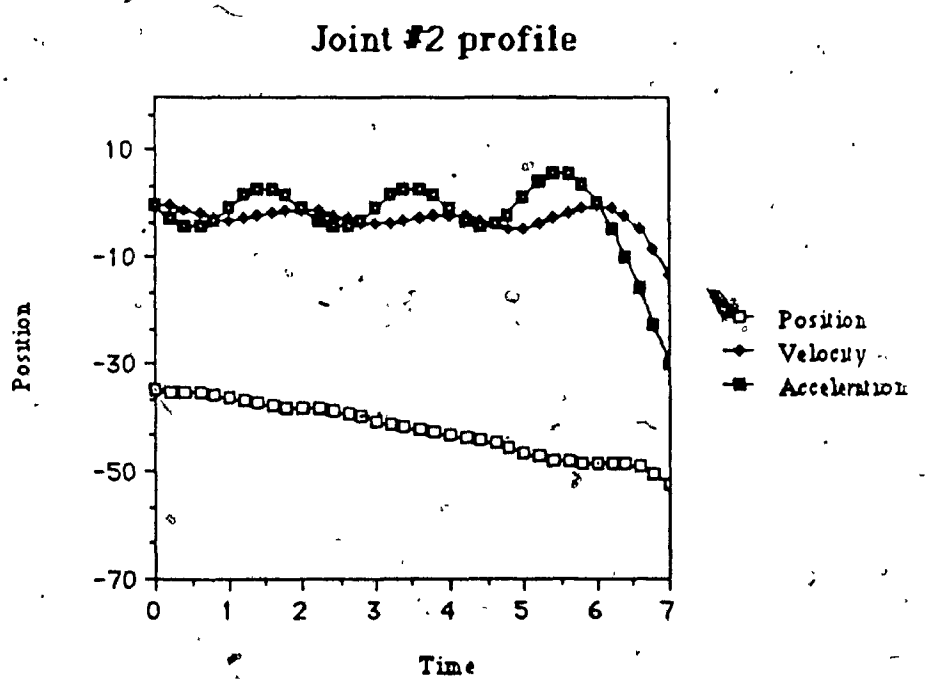
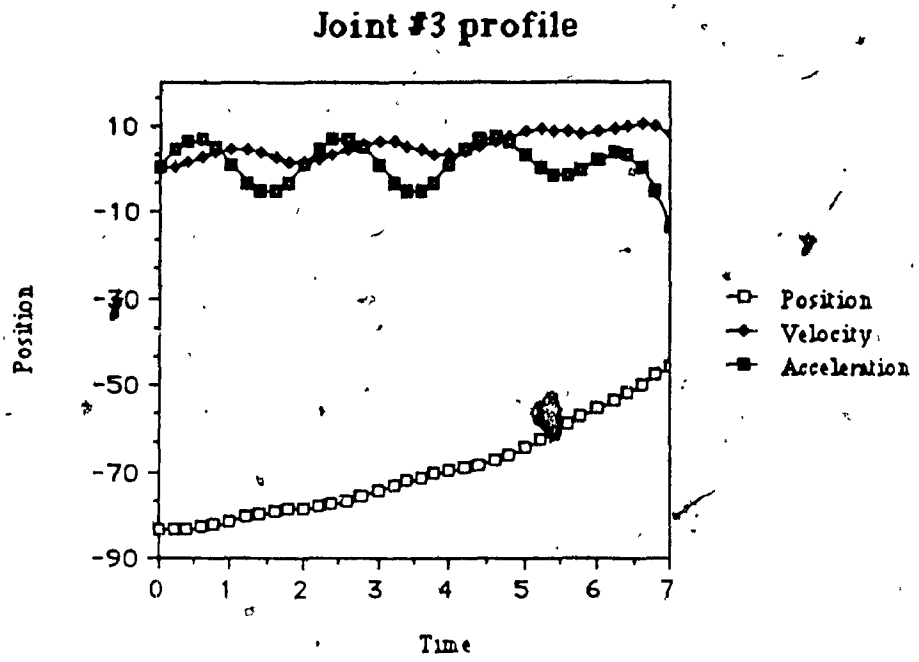


Figure 7.8 a, b. Angular position, velocity and acceleration profiles



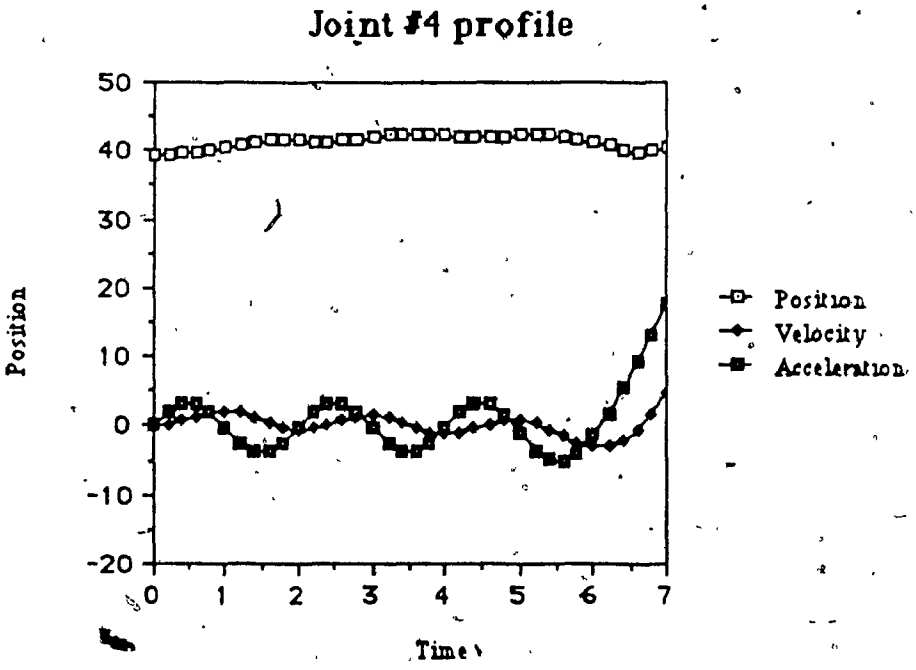
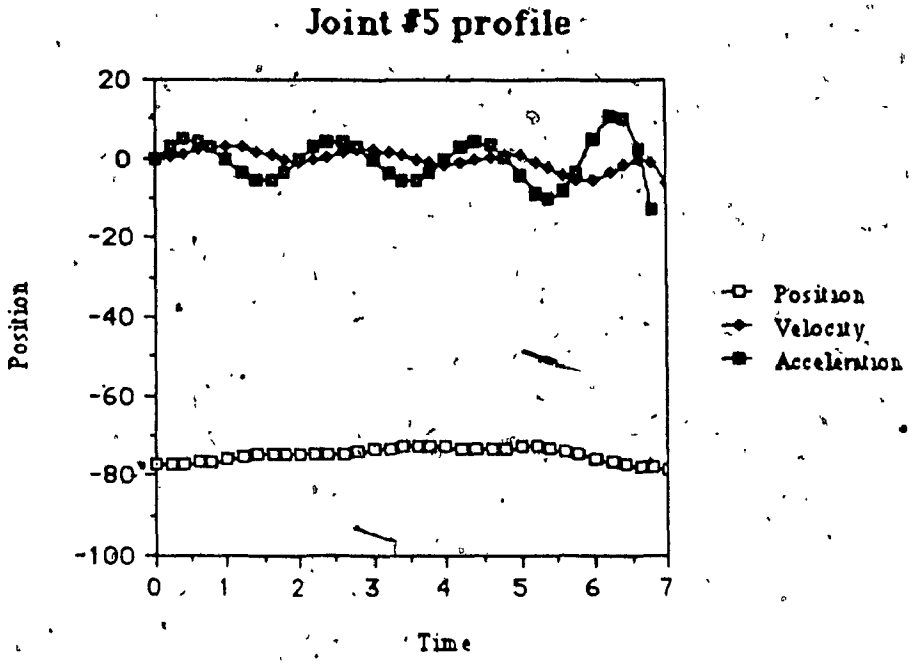


Figure 7.8 c, d: Angular position, velocity and acceleration profiles.



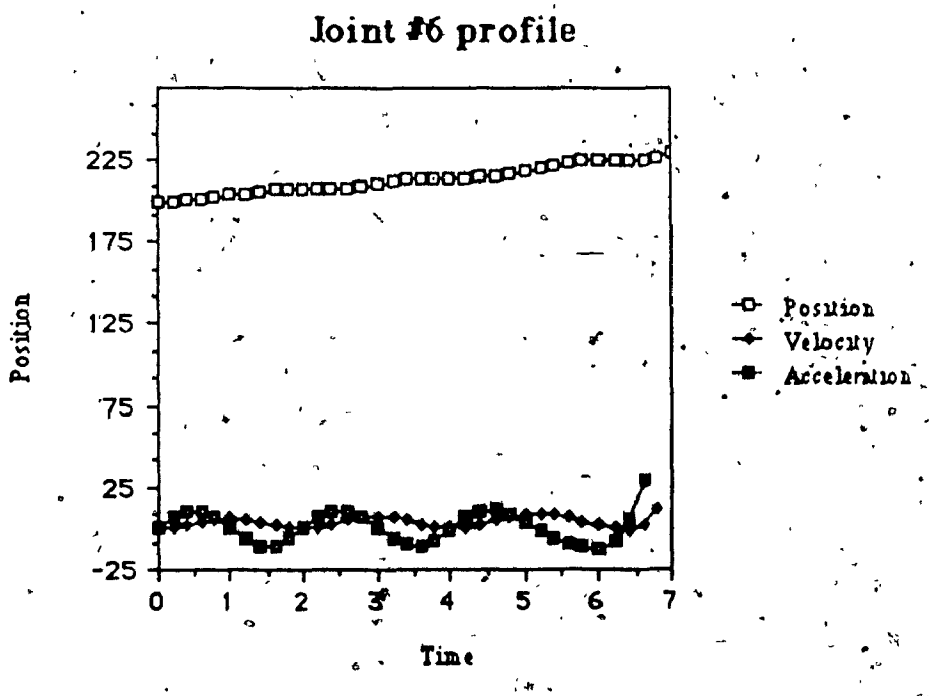
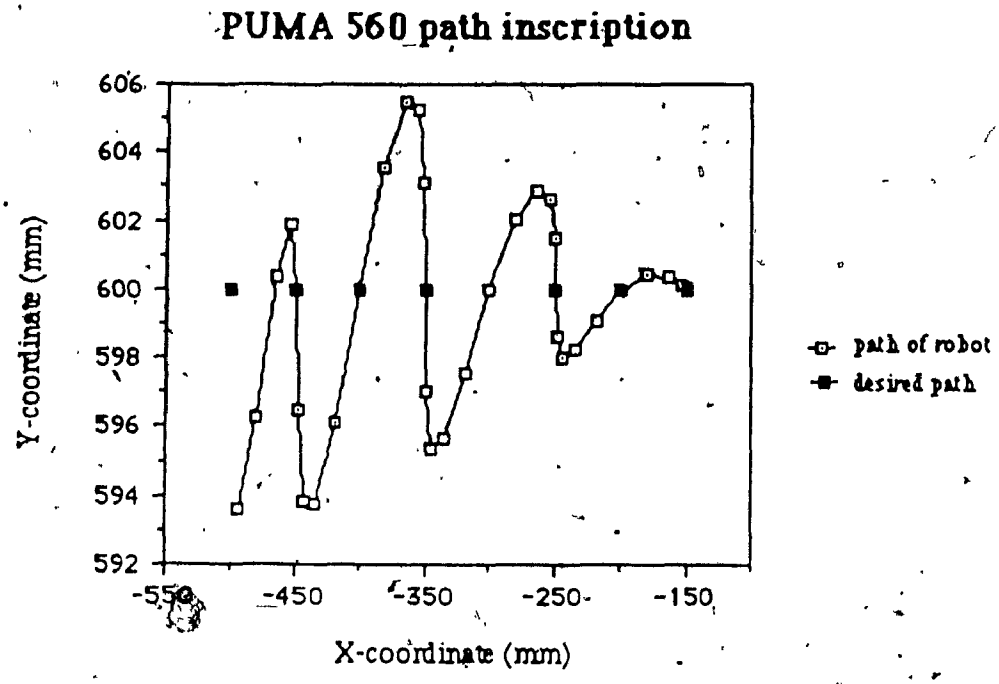


Figure 7.8 e, f: Angular position, velocity and acceleration profiles



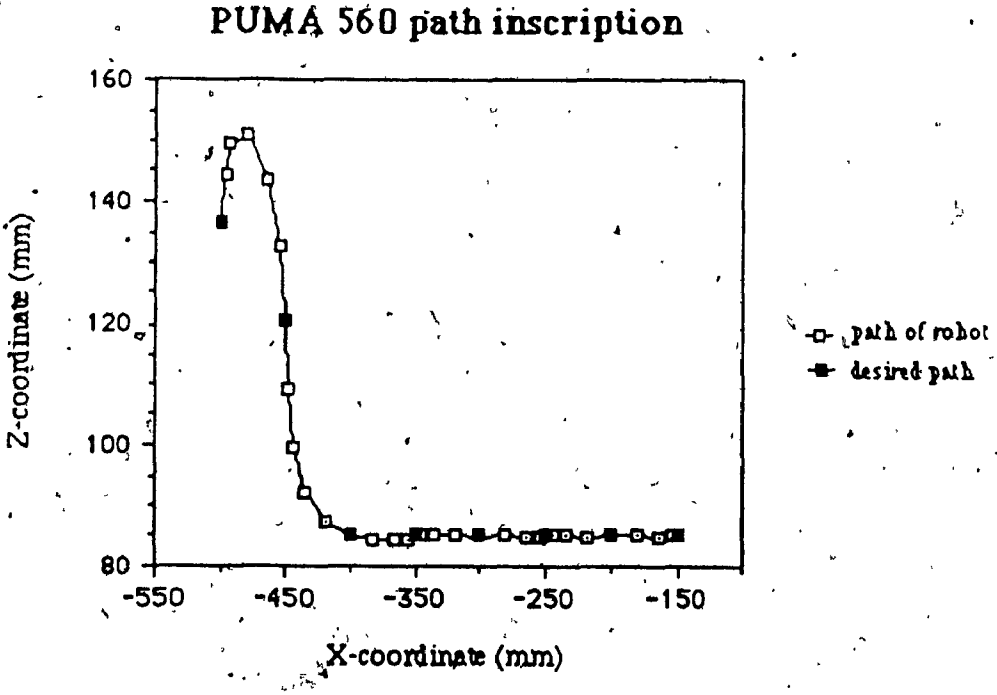


Figure 7.9 a, b: PUMA 560 path inscription error

Chapter 8 - Conclusions

Robot research encompasses a variety of areas including robot task planning, trajectory generation, kinematic and dynamic compensation in controller design, collision avoidance, multiple manipulator coordination and sensor feedback.

Robots must be able to perform independently and "intelligently" if they are to be better introduced to many sectors, such as manufacturing, medical technology, hazardous chemical experimentation, oceanography and space exploration. The accomplishment of such directives necessitates solutions to the many current research problems discussed in this thesis.

An attempt to bridge the gap between robot task planning and trajectory generation has been made in the programming environment called RIGID. This environment allows a user to describe a scene consisting of relations of coordinate frames using the RIGID language library and the standard C programming language. RIGID allows quite a variety of motion constraints and B-spline interpolation modes to be specified within each given trajectory. Both velocity and acceleration constraints can be applied to selected knots and may be specified at either the joint level, or in Cartesian space.

A mathematical development of a new formulation of B-splines is given which is suitable for constructing robot trajectories subject to certain boundary constraints and derivative extrema limits. These B-spline trajectory generation algorithms have been implemented in RIGID. It has been shown that extensions to the offline

trajectory generation techniques may be useful for online perturbations to perform compliant and collision avoidance manoeuvres.

Future possible extensions to RIGID include:

- support of more types of rigid-bodies,
- ability to specify tasks involving multiple and interacting manipulators,
- graphical representaion and/or interface using a library package such as GKS[128],
- an interpretor version using tools such as YACC [129], LEX [130] and compiler techniques [131] allowing higher levels of abstraction to specify robot tasks,
- a real time executive which communicates with some type of hardware, allowing external devices such as physical robots to be controlled.

References

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

[2] J. Denavit, R.S. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *ASME J. Appl. Mech.*, June 1955, pp. 215-221.

[3] S. Ahmad, "Second Order Nonlinear Kinematic Effects, and Their Compensation," *IEEE Int. Conf. Robotics and Automat.*, St. Louis, MI, March 1985.

[4] C.A. Balafoutis, P. Misra, R.V. Patel, "Recursive Evaluation of Linearized Dynamic Robot Models," *IEEE J. Robotics and Automat.*, vol. RA-2, no. 3, Sept. 1986, pp. 146-155.

[5] V. Hayward, R.P. Paul, "Robot Manipulator Control under UNIX RCCL: A Robot Control C Library," *Int. J. Robotics Res.*, vol. 5, no. 4, Winter 1986.

[6] R.A. Finkel, "Constructing and Debugging Manipulator Programs," Stanford University Computer Science Dept., Report no. STAN-CS-76-567, Stanford, CA, Aug. 1976.

[7] J.J. Craig, *Introduction to Robotics: Mechanics and Control*, Addison-Wesley, 1986.

[8] S.S. Leung, M.A. Shanblatt, "Real-Time DKS on a Single Chip," *IEEE J. Robotics Automat.*, vol. RA-3, no. 4, Aug. 1987, pp. 281-290.

[9] B.C. McInnis, C.K. Liu, "Kinematics and Dynamics in Robotics: A Tutorial Based Upon Classical Concepts of Vector Mechanics," *IEEE J. Robotics Automat.*, vol. RA-2, no. 4, Dec. 1986, pp. 181-187.

[10] *Unimate PUMA Mark II 500 Series, vol. I: Equipment Manual*, Unimation Inc., Shelter Rock Lane, Danbury, CT 06810, April 1984.

[11] S. Elgazzar, "Efficient Kinematic Transformations for the PUMA 560 Robot," *IEEE J. Robotics and Automat.*, vol. RA-1, no. 3, Sept. 1985, pp. 142-151.

[12] W.J. Crochetiere, "Locating the Wrist of an Elbow-Type Manipulator," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 3, May/June 1984, pp. 497-499.

[13] A.A. Goldenberg, B. Benhabib, R.G. Fenton, "A Complete Generalized Solution to the Inverse Kinematics of Robots," *IEEE J. Robotics and Automat.*, vol. RA-1, no. 1, March 1985, pp. 14-20.

[14] V.J. Lumelsky, "Iterative Coordinate Transformation Procedure for One Class of Robots," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 3, May/June 1984, pp. 500-505.

[15] L. Sciavicco, B. Siciliano, "Coordinate Transformation: A Solution Algorithm for One Class of Robots," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-16, no. 4, July/Aug. 1986, pp. 550-559.

[16] A. Bazerghi, A.A. Goldenberg, J. Apkarian, "An Exact Kinematic Model of PUMA 600 Manipulator," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 3, May/June 1984, pp. 483-487.

[17] R.P. Paul, B. Shimano, G.E. Mayer, "Kinematic Control Equations for Simple Manipulators," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-11, no. 6, June 1981, pp. 449-455.

[18] C.S.G. Lee, M. Ziegler, "A Geometric Approach in Solving the Inverse Kinematics of PUMA Robots," *13th ISIR/Robots 7 Proceedings*, August 1983, pp. 16.1-16.18.

[19] M. Shahinpoor, "The Exact Inverse Kinematics Solutions for the Rhino XR-2 Robot Manipulator," *Robotics Age*, vol. 7, no. 8, Aug. 1985, pp. 6-14.

[20] C.P. Neuman, J.J. Murray, "The Complete Dynamic Model and Customized Algorithms of the Puma Robot," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-17, no. 4, July/Aug 1987, pp. 635-644.

[21] A. Hemami, "Kinematics of Two-Arm Robots," *IEEE J. Robotics Automat.*, vol. RA-2, no. 4, Dec. 1987, pp. 225-228.

[22] W.P. Seering, "Robotics and Manufacturing - a Perspective," in *Robot Research: First International Symposium*, edited by M. Brady, R. Paul, MIT Press, 1984, pp. 973-982.

[23] T. Lozano-Perez, "A Simple Motion-Planning Algorithm for General Robot Manipulators," *IEEE J. Robotics Automat.*, vol. RA-3, no. 3, June 1987, pp. 224-238.

[24] T. Lozano-Perez, M.A. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Commun. ACM*, vol. 22, no. 10, Oct. 1979, pp. 560-570.

[25] T. Lozano-Perez, "Automatic Planning of Manipulator Transfer Movements," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-11, no. 10, Oct. 1981, pp. 681-689.

[26] T. Lozano-Perez, "Spatial Planning: A Configuration Space Approach ," *IEEE Trans. Computers*, vol. C-32, no. 2, Feb. 1983.

[27] R.A. Brooks, T. Lozano-Perez, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-15, no. 2, March/April 1985, pp. 224-233.

[28] S.M. Udupa, "Collision Detection and Avoidance in Computer Controlled Manipulators," *Proceedings of IJCAI-5*, Cambridge, MA: MIT Press, 1977, pp. 737-748.

[29] R.A. Brooks, "Solving the Find-Path Problem by Representing Free Space as Generalized Cones," *AI Memo #674*, AI Laboratory, MIT, Cambridge, MA, May 1982.

[30] J.Y.S. Luh, C.E. Campbell, "Collision-Free Path Planning for Industrial Robots," *Proc. 21st IEEE Conf. Decision Contr.*, 1982, pp. 84-88.

[31] D.C. Pieper, "The Kinematics of Manipulators Under Computer Control," *ARPA order #957*, Stanford University, Stanford, CA, 1968.

[32] C. Widdoes, "A Heuristic Collision Avoidance for the Stanford Robot Arm," *C.S. memo #227*, Stanford University, Stanford, CA, June 1974.

[33] H. Ozaki, A. Mohri, M. Takata, "On the Collision Free Movement of a Manipulator," in *Advanced Software in Robotics*, ed. A. Danthine and M. Geradin, Liege, Belgium, May 1983, pp. 189-200.

[34] Hans P. Moravec, "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover," PhD dissertation, Stanford, CA, *AIM-340*, Sept. 1980.

[35] Hans P. Moravec, *Robot Rover Visual Navigation*, Ann Arbor, MI; UMI Research Press, 1981.

[36] J. O'Rourke, N. Badler, "Decomposition of Three-Dimensional Objects into Spheres," *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. PAMI-1, no. 3, July 1979, pp. 295-305.

[37] B.I. Soroka, R.K. Bajcsy, "A Program for Describing Complex Three-Dimensional Objects Using Generalized Cyclinders as Primitives," *Proc. IEEE Conf. Patt. Recognition Image Processing*, 1978.

[38] G.T. Toussaint, "Some Collision Avoidance Problems Between Spheres," *Proc. IEEE Int. Conf. Cybernetics and Society*, Tucson, AZ, Nov. 1985, pp. 291-295.

[39] J.E. Hopcroft, J.T. Schwartz, M. Sharir, "Efficient Detection of Intersections among Spheres," *Int. J. Robotics Res.*, vol. 2, no. 4, Winter 1983, pp. 77-80.

[40] Y.C. Kim, J.K. Aggarwal, "Rectangular Parallelepiped Coding: A Volumetric Representation of Three-Dimensional Objects," *IEEE J. Robotics Automat.*, vol. RA-2, no. 3, Sept. 1986, pp. 127-134.

[41] L. Gouzenes, "Strategies for Solving Collision-Free Trajectories Problems for Mobile and Manipulator Robots," *Int. J. Robotics Res.*, vol. 3, no. 4, Winter 1984, pp. 51-65.

[42] R.A. Brooks, "Planning Collision-Free Motions for Pick-and-Place Operations," *Int. J. Robotics Res.*, vol. 2, no. 4, Winter 1983, pp. 19-44.

[43] N. Hogan, "Impedance Control: An Approach to Manipulation," *American Control Conference*, San Diego, Aug. 1984.

[44] N. Hogan, "Impedance Control: An Approach to Manipulation. Part I: Control of Mechanical Interaction. Part II: Control of End-Point Impedance," *ASME J. Dyn. Syst., Meas. and Contr.*, 1983.

[45] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *IEEE Int. Conf. Robotics and Automat.*, St. Louis, March 1985.

[46] K. Kant, S.W. Zucker, "Trajectory Planning in Time-Varying Environments I: TPP = PPP + VPP," *TR-84-7R*, Computer Vision and Robotics Laboratory, McGill University, Montreal, 1984.

[47] J.T. Schwartz, Micha Sharir, "On the Piano Movers Problem I: The Case of a Two Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers," *Communications on Pure and Applied Mathematics*, vol. xxxvi, 1983, pp. 345-398.

[48] J.T. Schwartz, M. Sharir, "On the Piano Movers Problem II: General Properties for Computing Topological Properties of Real Algebraic Manifolds," *Rept. 41: New York University Department of Computer Science*, Courant Institute of Mathematical Sciences, N.Y., 1982.

[49] J.T. Schwartz, M. Sharir, "On the Piano Movers Problem III: Coordinating the Motion of Several Independent Bodies: The Special Case of Circular Bodies Moving Among Polygonal Barriers," *Tech. Report New York University Department of Computer Science*, Courant Institute of Mathematical Science, N.Y., 1983.

[50] E.K. Wong, K.S. Fu, "A Hierarchical Orthogonal Space Approach to Collision-Free Path Planning," *Proc. IEEE Int. Conf. Robotics*, March 1985.

[51] B. Faverjon, "Obstacle Avoidance Using Octree in the Configuration Space of a Manipulator," *Proc. Int. Conf. Robotics*, March 1984.

[52] S. Kambhampati, L.S. Davis, "Multipresolution Path Planning for Mobile Robots," *IEEE J. Robotics Automat.*, vol. RA-2, no. 3, Sept. 1986, pp. 135-145.

[53] E.K. Wong, K.S. Fu, "A Hierarchical Orthogonal Space Approach to Three-Dimensional Path Planning, " *IEEE J. Robotics Automat.*, vol, RA-2, no. 1, March 1986, pp. 42-53.

[54] M.A. Peshkin, A.C. Sanderson, "Reachable Grasps on a Polygon: The Convex Rope Algorithm," *IEEE J. Robotics Automat.*, vol. RA-2, no. 1, March 1986, pp. 53-58.

[55] J.D. Wolter, R.A. Volz, A.C. Woo, "Automatic Generation of Gripping Positions," *IEEE Trans. Syst., Man, Cybern.*, vol SMC-15, no. 2, March/April 1985, pp. 204-213.

[56] T. Lozano-Perez, "Task Planning," in *Robot Motion: Planning and Control*, Cambridge, MA, MIT Press, 1982, pp. 490-493.

[57] J.M. Abel, W. Holtzman, J.M. McCarthy, "On Grasping Planar Objects With Two Articulated Fingers," *IEEE Trans. Robotics Automat.*, vol. RA-1, no. 4, Dec. 1985, pp. 211-214.

[58] J. Barber, R.A. Volz, et. al., "Automatic Evaluation of Two-Fingered Grips," *IEEE J. Robotics Automat.*, vol. RA-3, no. 4, Aug. 1987, pp. 356-361.

[59] R.S. Fearing, "Simplified Grasping and Manipulation with Dextrous Robot Hands," *IEEE J. Robotics Automat.*, vol. RA-2, no. 4, Dec. 1986, pp. 188-195.

[60] J.W. Jameson, L.J. Leifer, "Automatied Grasping: An Optimal Approach," *IEEE JTrans. Syst., Man, Cybern.*, vol. SMC-17, no. 5, Sept/Oct. 1987, pp. 806-814.

[61] D.E. Whitney, "Quasi-Static Assembly of Compliantly Supported Ridig Parts," in *Robot Motion: Planning and Control*, edited by M. Brady, et. al., MIT Press, Cambradge, MA, 1982, pp. 439-471.

[62] B. Dufay, J.C. Latombe, "An Approach to Automatic Robot Programming Based on Inductive Learning," in *Robot Research: Irst International Symposium*, edited by M. Brady, R. Paul, MIT Press, 1984, pp. 97-115.

[63] T. Lozano-Perez, M.T. Mason, R.H. Taylor, "Automatic Synthesis of Fine-Motion Strategies for Robots," in *Robot Research: Irst International Symposium*, edited by M. Brady, R. Paul, MIT Press, 1984, pp. 65-96.

[64] M.J. Dunne, "An Advanced Assembly Robot," in *Industrial Robots, vol. 2: Applications*, edited W.R. Tanner, Society of Manufacturing Engineers, Dearborn, MI, 1979, pp. 249-262.

[65] E. Freund, H. Hoyer, "Collision Avoidance for Industrial Robots with Arbitrary Motion," *J. Robotic Sys.*, vol. 1, no. 4, 1984, pp. 317-329.

[66] B.H. Lee, C.S.G. Lee, "Collision-Free Planning of Two Robots," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-17, no. 1, Jan/Feb. 1987, pp. 21-32.

[67] Y.F. Zheng, "Collision Effects on Two Coordinating Robots in Assembly and the Effect Minimization," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-17, no. 1, Jan/Feb. 1987, pp. 108-116.

[68] S. Mujtaba, R. Goldman, "AL Users' Manual," Third Edition, Stanford Dept. of Computer Science, Report No. STAN-CS-81-889, Dec. 1981.

[69] R. Goldman, *Design of an Interactive Manipulator Programming Environment*, UMI Research Press, Ann Arbor, MI, 1985.

[70] User's Guide to VAL II, *Part 1: Control from the System Terminal, Part 2: Communications with a Supervisory Computer, Part 3: Real-Time Path Control,* version X2, Unimation Inc., Danbury, CT, Apr. 1983.

[71] J.J. Craig, "JARS: JPL Autonomous Robot System, Robotics and Teleoperators Group, Jet Propulsion Laboratory, Pasadena, CA, 1980.

[72] R.A. Volz, T.N. Mudge, D.A. Gal, "Using Ada as a Programming Language for Robot-Based Manufacturing Cells," IEEE Trans. Syst., Man, Cybern., vol. SMC-14, no. 6, Nov/Dec. 1984, pp. 863-878.

[73] R.J. Popplestone, A.P. Ambler, I.M. Bellos, "An Interpreter for a Language for Describing Assemblies," in *Robot Motion: Planning and Control*, MIT Press, Cambridge, MA, 1982.

[74] T.C. Henderson, W.S. Fai, C. Hansen, "MKS: A Multisensor Kernel System," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 5, Sept./Oct. 1984; pp. 784-791.

[75] *Proceedings of the NATO Advanced Research Workshop on Languages for Sensor Based Control in Robotics*, Ciacco, Castelvecchio Pasco/Italy, Sept. 1-5, 1986 also published as *Languages for Sensor-Based Control in Robotics*, Springer-Verlag, Computer and Systems Science, vol. 29, 1986

[76] A. Zwarico, "Robot Programming Languages: Issues of Concurrency and Real-Time," *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Tucson, AZ, Nov. 1985, pp. 291-295.

[77] K.G. Shin, S.B. Malin, "A Structured Framework for the Control of Industrial Manipulators," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-15, no. 1, Jan/Feb. 1985, pp. 78-90.

[78] W.A. Gruver, B.I. Soroka, J.J. Craig, T.L. Turner, "Industrial Robor Programming Languages: A Comparitive Evaluation," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 4, July/Aug. 1984, pp. 565-570.

[79] T. Lozano-Perez, "Robot Programming," *Proc. IEEE*, vol. 7, July 1983, pp. 821-840.

[80] R.L. Burden, J.D. Faires, A.C. Reynolds, *Numerical Analysis*, Prindle; Weber & Schmidt, MA, 1981.

[81] I.D. Faux, M.J. Pratt, *Computational Geometry for Design and Manufacture,* John Wiley & Sons, 1979.

[82] B.A. Barsky, "A Description and Evaluation of Various 3-D Models," *IEEE Computer Graphics and Applications*, vol. 4, no. 1, January 1984, pp. 38-52.

[83] C. De Boor, *A Practical Guide to Splines*, Applied Mathematical Sciences, vol. 27, Springer-Verlag, New York, 1978.

[84] R.T. Farouki, J.K. Hinds, "A Hierarchy of Geometric Forms," *IEEE Computer Graphics and Applications*, vol. 5, no. 5, May 1985, pp. 51-78.

[85] B.A. Barsky, J.C. Beatty, "Local Control of Bias and Tension," *Computer Graphics*, vol. 17, no. 3, July 1983, pp. 193-218.

[86] G.M. Nielson, "A Locally Controllable Spline with Tension for Interactive Curve Design," *Computer-Aided Design*, vol. 14, no. 3, 1984, pp. 199-205.

[87] T.N.T. Goodman, K. Unsworth, "Manipulating Shape and Producing Geometric Continuity in β-Spline Curves," *IEEE Trans. Computer Graphics and Applications*, vol. 6, no. 2, Feb. 1986, pp. 50-56.

[88] G.M. Nielson, "Computation of v-Splines," *technical report NR 044-433-11*, Dept. Mathematics, Arizona State University, Tempe, Arizona, 1974.

[89] G.M. Nielson, "Rectangular v-Splines," *IEEE Trans. Computer Graphics and Applications*, vol. 6, no.2, Feb. 1986, pp. 35-40.

[90] M. Brady, et al., "Trajectory Planning," in *Robot Motion: Planning and Control*, MIT Press, MA, 1982.

[91] M.S. Mujtaba, "Discussion of Trajectory Calculation Methods," in *Exploratory Study of Computer Integrated Assembly Systems*, T. O. Binford, et. al.; Stanford University, Artificial Intelligence Laboratory, AIM 285.4, 1977.

[92] D.E. Witney, "Resolved Motion Rate Control of Manipulators and Human Prostheses," *IEEE Trans. Man-Machine Systs.*, vol. 10, no. 2, June 1969, pp. 47-53.

[93] D.E. Witney, "The Mathematics of Coordinated Control of Prostheses and Manipulators," *ASME J. Dynamic Syst., Meas., Contr.*, Dec. 1972, pp. 303-309.

[94] R.P. Paul, "Manipulator Cartesian Path Control," *IEEE Trans. Syst. Man, Cybern.*, vol. SMC -9, no. 11, Nov. 1979, pp. 702-711.

[95] R.H. Taylor, "Planning and Execution of Straight Line Manipulator Trajectories," *IBM J. Res., Develop.*, vol. 23, no. 4, July 1979, pp. 425-436.

[96] W. Khalil, "Generation of Straight Line Motion for Robots," *2nd IASTED Int. Symp. Robotics and Automation*, Lugano Switzerland, June 1983, pp. 133-135.

[97] W. Khalil, "Trajectories Calculations in the Joint Space of Robots," in *Advanced Software in Robotics*, Liège, Belgium, May 1983, pp. 177-187.

[98] R.H. Castain, R.P. Paul, "An On-Line Dynamic Trajectory Generator," *Int. J. Robotics Res.*, vol. 3, no. 1, Spring 1984, pp. 68-72.

[99] J.Y.S. Luh, C.S. Lin, "Optimum Path Planning for Mechanical Manipulators," *ASME J. Dynamic Syst., Meas., Contr.*, vol. 102, June 1981, pp. 142-151.

[100] C.S. Lin, P.R. Chang, J.Y.S. Luh, "Formulation and Optimization of Cubic Polynomial Joint Trajectories for Industrial Robots," *IEEE Trans. Automat. Contr.*, vol. AC-28, no. 12, Dec. 1983, pp. 1066-1074.

[101] C.S. Lin, P.R. Chang, "Approximate Optimum Paths of Robot Manipulators under Realistic Physical Constraints," *IEEE Int. Conf. Robotics and Automation*, St. Louis, Missouri, March 1985, pp. 737-742.

[102] C.S. Lin, P.R. Chang, "Joint Trajectory of Mechanical Manipulators for Cartesian Path Approximation," *IEEE Trans. Syst. Man, Cybern.*, vol SMC-13, no. 6, Nov/Dec. 1983, pp. 1094-1102.

[103] J.Y.S. Luh, C.S. Lin, "Approximate Joint Trajectories for Control of Industrial Robots along Cartesian Paths," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 3, May/June 1984, pp. 444-450.

[104] S. Chand, K.L. Doty, "On-Line Polynomial Trajectories for Robot Manipulators," *Int. J. Robotics Res.*, vol. 4, no. 2, Summer 1985, pp. 38-48.

[105] M. Vukobratovic, M. Kircanski, "A Dynamic Approach to Nominal Trajectory Synthesis for Redundant Manipulators," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-14, no. 4, July/August 1984, pp. 580-586.

[106] B.K. Kim, K.G. Shin, "Minimum-Time Path Planning for Robot Arms and Their Dynamics," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-15, no. 2, March/April 1985, pp. 213-223.

[107] K.G. Shin, N.D. McKay, "Minimum-Time Control of Robotic Manipulators with Geometric Path Constraints," *IEEE Trans. Automat. Contr.*, vol. AC-30, no. 6, June 1985, pp. 531-541.

[108] J.E. Bobrow, S. Dubowsky, J.S. Gibson, "On the Optimal Control of Robotic Manipulators with Actuator Constraints," *Proc. 1983 American Contr. Conf.*, San Francisco, CA, June 1983, pp. 782-787.

[109] C.S.G. Lee, B.H. Lee, "Planning of Straight Line Manipulator Trajectory in Cartesian Space with Torque Constraints," *Proc. 23rd Conf. Decision Contr.*, Las Vegas, NV, Dec. 1984, pp. 1603-1609.

[110] G.H. Seeger, R.P. Paul, "Optimizing Robot Motion Along a Predefined Path," *IEEE Int. Conf. of Robotics and Automation*, St. Louis, Missouri, March 1985, pp. 765-770.

[111] K.G. Shin, N.D. McKay, "A Dynamic Programming Approach to Trajectory Planning of Robotic Manipulators," *IEEE Trans. Automat. Contr.*, vol. AC-31, no. 6, June 1986, pp. 491-500.

[112] K.G. Shin, N.D. McKay, "Selection of Near-Minimum Time Geometric Paths for Robotic Manipulators," *IEEE Trans. Automat. Contr.*, vol. AC-31, no. 6, June 1986, pp. 501-511.

[113] S.E. Thompson, R.V. Patel, "Formulation of Joint Trajectories for Industrial Robots Using B-splines," *IEEE Trans. Ind. Electron.*, vol. IE-34, no. 2, May 1987, pp. 192-199.

[114] S.E. Thompson, R.V. Patel, "Generation of Collision-Free Trajectories for Industrial Robots," *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Tucson, AZ, Nov. 1985, pp. 300-304.

[115] W.J. Gordon, R.F. Riesenfeld, "B-Spline Curves and Surfaces," in *Computer-Aided Geometric Design*, eds. R.E. Barnhill and R.F. Riesenfeld, Academic Press, New York, 1974.

[116] C. De Boor, "On Calculating with B-splines," *J. Approx. Theory*, vol. 6, 1972, pp. 50-62.

[117] M.G. Cox, "The Numerical Evaluation of B-splines," *J. Inst. Math. Applic.*, vol. 10, 1972, pp. 134-149.

[118] B.A. Barsky, "End Conditions and Boundary Conditions for Uniform B-spline Curve and Surface Representations," *Computers in Industry*, vol. 3, 1982, pp. 17-29.

[119] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[120] Microsoft Corporation, *Microsoft C Compiler*, Redmond, WA, 1986.

[121] Think Technology, *Lightspeed C User's Manual*, Bedford, MA, 1987.

[122] M. Waite, S. Prata, D. Martin, *C Primer Plus*, Howard W. Sams & Co., Indianapolis, IN, 1984.

[123] G.E. Sobelman, D.E. Krekelberg, *Advanced C Techniques & Applications*, Que Corporation, Indianapolis, IN, 1985.

[124] S.P. Harbison, G.L. Steele Jr., *C Reference Manual*, Prentice-Hall Software Series, 1987.

[125] Augie Hansen, *Proficient C*, Microsoft Press, Redmond, WA, 1987.

[126] A.M. Tenenbaum, M.J. Augenstein, *Data structures using Pascal*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[127] Robert Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

[128] ISO, *Graphic Kernel System (GKS) - Function Description*, Draft International Standard ISO/DIS 7942, 1982.

[129] S.C. Johnson, "Yacc: Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, New Jersey 07974.

[130] M.E. Lesk, E. Schmidt, "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murray Hill, New Jersey 07974.

[131] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986.

[132] Convergent Technology Inc., "The C Programming Language," *CTIX Programmers Manual*, 1986, pp. 19-34-38.

Appendix A - RIGID Reference Library

Constants

Some common constants used throughout RIGID have been predeclared in the rigid.h header file, which is normally included as the first line in the user's program, i.e. the **#include "rigid.h"** statement. The rigid.h header file is listed at the end of this appendix. The following is a list of the constants included:

PI	value of pi (π), set to 3.14159
DEGTORAD	degree to radian measure conversion, set to π / 180°
RADTODEG	radian to degree measure conversion, set to 180° / π
TWOPI, PIBY2	value of 2 π, value of π / 2
TRIGERR	trigonometric tolerance limit, set to 0.001
TRUE, FALSE	boolean data type values
NULL	character string terminator, set to ascii 0
NIL	pointer terminator, set to **0L**
SIZENAME	maximum number of characters in a name, set to 80
RPY, EUL, OAT	coordinate transform functions
MATRIX, VECTOR	parameters for **SetOutputMode()**
JOINT, CARTESIAN	parameters for **SetMotionMode()**
RHS, LHS	parameters for **EquateFrame()**, **EquateRelation()**
NOBODY, PUMA560	parameters for **SetBodyType()**

Data types

The following data types have their data structure layouts given in Chapter 6. All other data types which occur in the following pages are considered standard C.

boolean	makeshift boolean data type (**typedef int boolean**)
coord_t	coordinate function type
frame_t	coordinate frame type
velocity_t	velocity constraint type
relation_t	kinematic relation type
trajectory_t	trajectory specification type

The following data types are private, or internal to RIGID, but should be noted to preserve uniqueness of data structure names.

errtrap_t	error trap type
spline_t	spline function type
body_t	rigid body type
part_t	relation part type
knot_t	trajectory knots type
segment_t	trajectory segment type
fifo_t	generic queue header type
node_t	generic queue data type

Functions

The functions which comprise the RIGID language library are described in the following pages. These functions are partitioned into four categories: world primitive operators (WP), kinematic relationships (KR), trajectory generation (TG) and reporting functions (RF).

Functions pertaining to world primitive operators allow coordinate frames of objects and locations to be transformed between vector and matrix representations. Velocity constraint specification functions are also included in this category. The category on kinematic relations provides the functions used to construct and evaluate kinematic relations between frames. The trajectory generation category encompases all the functions required to generate trajectories using frames, relations and velocity constraints. Functions used for selecting and configuring the modes of motion for the rigid body types are also considered part of the trajectory generation category. Reporting functions allow convenient output of results directly from various data types in RIGID. These functions incorporate the format specification of the C language standard input/output (stdio) library.

RIGID Function Summary

World Primitive Operators

frame_t *BuildFrame(char *, coord_t,double,double,double, double, double, double)
void UnBuildFrame(frame_t *, double*,double*,double*,double*,double*,double*)
void UpdateFrameRel(frame_t *, double,double,double,double,double,double)
void UpdateFrameAbs(frame_t *, double, double, double, double, double, double)
velocity_t *BuildVelocity(char *, coord_t,double,double,double,double,double,double)
void UnBuildVelocity(velocity_t *, double*,double*,double*,double*,double*,double*)

Kinematic Relationships

boolean OpenRelation(char *)
void EquateFrame(int, frame_t *)
void EquateRelation(int, relation_t *)
relation_t *CloseRelation(void)
void EvalRelation(double *, relation_t *)

Trajectory Generation

boolean OpenTrajectory(char *)
void MoveToFrame(frame_t *)
void MoveToRelation(relation_t *)
void MoveToTrajectory(trajectory_t *)
void BodyPosition(double, ...)
void SetVelocity(velocity_t *)
void BodyVelocity(double, ...)
void BodyAcceleration(double, ...)
trajectory_t *CloseTrajectory(void)
void EvalTrajectory(trajectory_t *)
double EnquireTrajectory(trajectory_t *)
void SetCoordSys(coord_t)
void SetSampleStep(double)
void SetMotionMode(int)
void SetBodyType(void (*)())
void SetBodyConfig(int, ...)
void GetBodyConfig(int *, ...)
void SetVelConstraints(double, ...)
void SetAccConstraints(double, ...)

Reporting Functions

void SetOutputMode(int)
void WriteMatrix(char *, double *)
void WriteVector(char *, double *)
void WriteFrame(char *, int, frame_t *, ...)
void WriteVelocity(char *, int, velocity_t *, ...)
void WriteRelation(char *, int, relation_t *, ...)
void WriteTrajectory(char *, int, trajectory_t *, ...)

BuildFrame (WP)

Summary

```
#include "rigid.h"

frame_t *BuildFrame(fname, ffun, x, y, z, a, b, c)
char *fname;
coord_t ffun;
double x, y, z, a, b, c;
```

Description

The **BuildFrame()** function creates a homogeneous transform, which is made up of a translation vector x, y, z, and a rotation vector of angles a, b, c (in degrees). The units of position throughout the world primitive operators are ultimately interpreted by the rigid-body transformation functions which are mm for **PUMA560**, see **SetBodyType()** for more details. The geometry of the rotations depend upon the type of transformation selected by **ffun**. The **ffun** argument is an array of pointers to predefined routines which convert the translation and rotation argument vectors into a homogeneous transform matrix. The **ffun** argument may be set to one of the following predefined arrays of functions:

- **RPY** roll-pitch-yaw rotations and standard translation,
- **EUL** Euler angle rotations and standard translation,
- **OAT** Unimation rotations and standard translation.

The **fname** argument is a character string used by some output functions for identification and for tracing program execution. The **fname** may be at most **SIZENAME** characters. The inverse transform is also computed at this time using an orthogonal matrix inversion technique which avoids the numerical instability problem caused by some types of matrices.

The user may provide some other type of transformation function for frame coordinate specification by using the following steps. The example supposes that functions **FwdNEW()**, **InvNEW()** are the forward and inverse kinematic transformation functions, respectively. RIGID expects the description name of the transform function as the first parameter (**NEW.cname**), the forward transform function pointer second (**NEW.fun[0]**) and the inverse transform function pointer (**NEW.fun[1]**). Declare these before the driver program declaration (**main()**) so that the driver program would then refer to just **NEW** as the new **ffun** argument.

```
void FwdNEW(double*,double,double,double, double,double,double);
void InvNEW(double*, double*, double*, double*, double*,
   double*, double*);
static coord_t NEW = {"NEW angles", FwdNEW, InvNEW};
```

where these are the actual functions:

```
void FwdNEW(m, x, y, z, a, b, c)
double *m;
double x, y, z, a, b, c;
{
```

```
    a *= DEGTORAD;
    :
    *(m + 3) = x;
    *(m + 7) = y;
    :
}

void InvNEW(m, x, y, z, a, b, c)
double *m;
double *x, *y, *z, *a, *b, *c;
{
    :
    *x = *(m + 3);
    *y = *(m + 7);
    a *= RADTODEG;
    :
}
```

Return Value

A pointer to the frame_t data structure is returned. A NIL pointer value is returned and an error message generated if the ffun argument is undefined.

See Also

UnBuildFrame(), SetBodyType()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;

    rest = BuildFrame("Resting",RPY,50.0,30.0,0.0,90.0,0.0,-45.0);
    :
}
```

UnBuildFrame (WP)

Summary

```
#include "rigid.h"

void UnBuildFrame(f, x, y, z, a, b, c)
frame_t *f;
double *x, *y, *z, *a, *b, *c;
```

Description

The **UnBuildFrame()** function performs the inverse of **BuildFrame()**. The f argument contains a homogeneous transform, which gets decomposed into a translation vector x, y, z, and a rotation vector of angles a, b, c (in degrees). The translation/rotation values are returned in the same coordinate system as they were specified in by the call to **BuildFrame()**.

If the f argument is undefined, the call to **UnBuildFrame()** is ignored.

Return Value

A return value of FALSE indicates that the f argument is undefined.

See Also

BuildFrame()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;
    double x, y, z, a, b, c;

    :
    UnBuildFrame(rest, &x, &y, &z, &a, &b, &c);
    :
}
```

UpdateFrameRel (WP)

Summary

```
#include "rigid.h"

void UpdateFrameRel(f, dx, dy, dz, da, db, dc)
frame_t *f;
double dx, dy, dz, da, db, dc;
```

Description

During execution the transformation of a previously defined frame may be updated to a new set of coordinates using the **UpdateFrameRel()** function. This function updates the frame **f** relative to its previous coordinates. Since RIGID consists entirely of dynamic data structures, the updated frame affects every occurance of itself among all relations and trajectories. The translation/rotation values are updated in the same coordinate system as they were specified in by the call to **BuildFrame()**, resulting in the coordinates (x+dx, y+dy, z+dz, a+da, b+db, c+dc).

If the **f** argument is undefined, the call to **UpdateFrameRel()** is ignored.

See Also

BuildFrame(), UpdateFrameAbs()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;

    :
    rest = BuildFrame("Resting",RPY,50.0,30.0,0.0,90.0,0.0,-50.0);
    UpdateFrameRel(rest, 10.0, 0.0, 0.0, 90.0, 0.0, 45.0);
    :
}
```

UpdateFrameAbs (WP)

Summary

```
#include "rigid.h"

void UpdateFrameAbs(f, x, y, z, a, b, c)
frame_t *f;
double x, y, z, a, b, c;
```

Description

During execution the transformation of a previously defined frame may be updated to a new set of coordinates using the **UpdateFrameAbs()** function. This function updates the frame f to an absolute location with respect to world coordinates. Since RIGID consists entirely of dynamic data structures, the updated frame affects every occurance of itself among all relations and trajectories. The translation/rotation values are updated in the same coordinate system as they were specified in by the call to **BuildFrame()**, resulting in the coordinates (x, y, z, a, b, c).

If the f argument is undefined, the call to **UpdateFrameAbs()** is ignored.

See Also

BuildFrame(), **UpdateFrameRel()**

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;

    :
    rest = BuildFrame("Resting",RPY,50.0,30.0,0.0,90.0,0.0,-50.0);
    UpdateFrameAbs(rest, 120.0, -20.0, 0.0, 90.0, 0.0, 45.0);
    :
}
```

BuildVelocity (WP)

Summary

```
#include "rigid.h"

velocity_t *BuildVelocity(vname, vfun, x, y, z, a, b, c)
char *vname;
coord_t vfun;
double x, y, z, a, b, c;
```

Description

The BuildVelocity() function creates a velocity constraint specification, which is made up of a linear velocity vector with elements x, y, z, and an angular velocity vector with elements a, b, c (in degrees/sec). The geometry of the rotations depends upon the type of transformation selected by vfun. The vfun argument is an array of pointers to predefined routines which are capable of converting the specified rotational information to a matrix representation

The vname argument is a character string used by some output functions for identification and for tracing program execution. The vname may be at most SIZENAME characters.

Refer to BuildFrame() for providing user-defined transformation functions for velocity coordinate specification.

Return Value

A pointer to the velocity_t data structure is returned. A NIL pointer value is returned if the vfun argument is undefined.

See Also

BuildFrame()

Example

```
#include "rigid.h"

main()
{
    velocity_t *slow;

    slow = BuildVelocity("Approach",RPY,10.0,10.0,0.0,0.0,0.0,0.0);
    :
}
```

UnBuildVelocity (WP)

Summary

```
#include "rigid.h"

void UnBuildVelocity(v, x, y, z, a, b, c)
velocity_t *v;
double *x, *y, *z, *a, *b, *c;
```

Description

The **UnBuildVelocity()** function performs the inverse of **BuildVelocity()**. The **v** argument contains a matrix transform, which gets decomposed into a linear velocity vector with elements **x**, **y**, **z**, and an angular velocity vector with elements **a**, **b**, **c** (in degrees/sec). The translation/rotation values are returned in the same coordinate system as they were specified in by the call to **BuildFrame()**.

If the **v** argument is undefined, the call to **UnBuildFrame()** is ignored.

Return Value

A return value of FALSE indicates that the **v** argument is undefined.

See Also

BuildVelocity()

Example

```
#include "rigid.h"

main()
{
   velocity_t  *slow;
   double x, y, z, a, b, c;

   .
   .
   UnBuildVelocity(slow, &x, &y, &z, &a, &b, &c);
   .
}
```

OpenRelation (KR)

Summary

```
#include "rigid.h"

boolean OpenRelation(rname)
char *rname;
```

Description

A call to **OpenRelation()** opens a new kinematic relation to be built by subsequent calls to either **EquateFrame()** or **EquateRelation()**. The **rname** argument is a character string used by some output functions for identification and for tracing program execution. The **rname** may be at most **SIZENAME** characters. The relation is not completely linked into the internal data structures until the terminating call to **CloseRelation()**, usually after all the frames in the kinematic equation have been specified.

Only one relation at a time may be open while frames are being equated to the kinematic relation so that the system can discriminate which relation is currently open

The number of relations possible per application depends upon the amount of main memory available for dynamic allocation

Return Value

A return value of **FALSE** indicates that another relation has not been closed

See Also

CloseRelation(), EquateFrame(), EquateRelation()

Example

```
#include "rigid.h"

main()
{
    :
    OpenRelation("Find Unknown");
    :
}
```

EquateFrame (KR)

Summary

```
#include "rigid.h"

void EquateFrame(side, f)
int side;
frame_t *f;
```

Description

The EquateFrame() function equates the f argument to the currently open relation. The frame must be previously created using the BuildFrame() function. The side argument indicates which side of the relation equation the frame should appear on by setting it to one of the following constants:

RHS right hand side of kinematic equation.
LHS left hand side of kinematic equation.

In general, frames equated on the left hand side (LHS) have their inverse transformation matrix premultiplied, while right hand side (RHS) frames have their forward transformation matrix postmultiplied.

The number of frames which may be part of a relation depends upon the amount of main memory available for dynamic allocation.

If no relation is curently open when a call to EquateFrame() is made, the call is ignored. If the f argument is undefined, the call to EquateFrame() is ignored.

See Also

BuildFrame(), OpenRelation()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;

    :
    OpenRelation("Find Unknown");
        EquateFrame(RHS, rest);
    :
}
```

EquateRelation (KR)

Summary

```
#include "rigid.h"

void EquateRelation(side, r)
int side;
relation_t *r;
```

Description

The **EquateRelation()** function equates the r argument to the currently open relation. The relation r must be previously created using the **CloseRelation()** function. The side argument indicates which side of the equation the relation solution should appear on by setting it to one of the following constants:

RHS right hand side of kinematic equation.
LHS left hand side of kinematic equation.

In general, relations equated on the left hand side (**LHS**) have their inverse transformation matrix premultiplied, while right hand side (**RHS**) relations have their forward transformation matrix postmultiplied. When relations are evaluated, the matrix solution is internally substituted as a frame in the kinematic equation represented by a relation. The number of relations which may be included within a relation depends upon the amount of main memory available for dynamic allocation. Due to the recursive structure induced by this function, when evaluating the kinematic equations, it may be necessary to increase the stack size at link time to prevent stack overflow during program execution. Again, this is hardware dependent.

If no relation is curently open when a call to **EquateRelation()** is made, the call is ignored. If the r argument is undefined, the call to **EquateRelation()** is ignored.

See Also

CloseRelation()

Example

```
#include "rigid.h"

main()
{
    relation_t *soln;

    :
    soln = CloseRelation();
    OpenRelation("Find Unknown");
        EquateRelation(soln);
    :
}
```

CloseRelation (KR)

Summary

```
#include "rigid.h"

relation_t *CloseRelation(void)
```

Description

A call to CloseRelation() terminates the definition of the currently open relation.

If no relation has been opened prior to CloseRelation() an error condition is generated and a NIL pointer is returned.

Return Value

A pointer to the newly formed relation is returned. A NIL pointer value is returned if no relation is currently open.

See Also

OpenRelation()

Example

```
#include "rigid.h"

main()
{
   relation_t *unknown;

   OpenRelation("Find Unknown");
   :
   unknown = CloseRelation();
}
```

EvalRelation (KR)

Summary

```
#include "rigid.h"

void EvalRelation(m, r)
double *m;
relation_t *r;
```

Description

A call to EvalRelation() solves the kinematic equation represented by the r argument. The solution matrix is returned in the m argument. Normally the user writes the solution matrix to a file (or screen) using the WriteRelation() function, but EvalRelation() allows the user to access the solution matrix data from within a RIGID program. Refer to BuildFrame() for more details on the transformation function options.

If the r argument is undefined, the call to EvalRelation() is ignored.

See Also

OpenRelation()

Example

```
#include "rigid.h"

main()
{
   relation_t *unknown;
   double res[4][4];

   :
   unknown = CloseRelation();
   EvalRelation((double *) res, unknown);
   :
}
```

OpenTrajectory (TG)

Summary

```
#include "rigid.h"

boolean OpenTrajectory(tname)
char *tname;
```

Description

A call to OpenTrajectory() opens a new trajectory to be built by subsequent calls to either MoveToFrame(), MoveToRelation(), MoveToTrajectory(), SetVelocity(); BodyPosition(), BodyVelocity() and BodyAcceleration(). The tname argument is a character string used by some output functions for identification and for tracing program execution. The tname may be at most SIZENAME characters. The trajectory is not completely linked into the internal data structures until the terminating call to CloseTrajectory(), usually after all the frames and velocity constraints have been specified.

Only one trajectory at a time may be open while frames and velocities are being specified in the trajectory so that the system can descriminate which trajectory is currently open.

The number of trajectories possible per application depends upon the amount of main memory available for dynamic allocation.

Return Value

A return value of FALSE indicates that another trajectory has not been closed.

See Also

CloseTrajectory(), SetVelocity(), MoveToFrame(), MoveToRelation(), MoveToTrajectory(), BodyPosition(), BodyVelocity(), BodyAcceleration()

Example

```
#include "rigid.h"

main()
{
    :
    OpenTrajectory("Good Move");
    :
}
```

MoveToFrame (TG)

Summary

```
#include "rigid.h"

void MoveToFrame(f)
frame_t *f;
```

Description

The MoveToFrame() function equates the f argument as the next frame in the currently open trajectory. The frame f must be previously created using the **BuildFrame()** function.

The number of frames which may be part of a trajectory depends upon the amount of main memory available for dynamic allocation.

If no trajectory is currently open when a call to **MoveToFrame()** is made, the call is ignored. If the f argument is undefined, the call to **MoveToFrame()** is ignored.

See Also

BuildFrame(), OpenTrajectory()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;
    :
    OpenTrajectory("Good Move");
        MoveToFrame(rest);
    :
}
```

MoveToRelation (TG)

Summary

```
#include "rigid.h"

void MoveToRelation(r)
relation_t *r;
```

Description

The **MoveToRelation()** function equates the solution to the kinematic equation represented by the **r** argument as the next frame in the currently open trajectory. The relation **r** must be previously created using the **CloseRelation()** function.

The number of relations which may be part of a trajectory depends upon the amount of main memory available for dynamic allocation. Due to the recursive structure induced by this function, when evaluating the kinematic equations, it may be necessary to increase the stack size at link time to prevent stack overflow during program execution. Again, this is hardware dependent.

If no trajectory is currently open when a call to **MoveToRelation()** is made, the call is ignored. If the **r** argument is undefined, the call to **MoveToRelation()** is ignored.

See Also

CloseRelation(), OpenTrajectory()

Example

```
#include "rigid.h"

main()
{
    relation_t *soln;

    :
    soln = CloseRelation();
    OpenTrajectory("Good Move");
        MoveToRelation(soln);
    :
}
```

MoveToTrajectory (TG)

Summary

```
#include "rigid.h"

void MoveToTrajectory(t)
trajectory_t *t;
```

Description

The MoveToTrajectory() function equates the solution to the kinematic equation represented by the t argument as the next frame in the currently open trajectory. The trajectory t must be previously created using the CloseTrajectory() function.

The number of trajectories which may be part of a trajectory depends upon the amount of main memory available for dynamic allocation. Due to the recursive structure induced by this function, when generating trajectories, it may be necessary to increase the stack size at link time to prevent stack overflow during program execution. Again, this is hardware dependent.

If no trajectory is currently open when a call to MoveToTrajectory() is made, the call is ignored. If the t argument is undefined, the call to MoveToTrajectory() is ignored.

See Also

CloseTrajectory()

Example

```
#include "rigid.h"

main()
{
    trajectory_t *soln;

    :
    soln = CloseTrajectory();
    OpenTrajectory("Good Move");
        MoveToTrajectory(soln);
    :
}
```

BodyPosition (TG)

Summary

```
#include "rigid.h"

void BodyPosition(p, ...)
double p;
```

Description

It is possible to construct trajectories for rigid-body mechanisms by directly specifiying the individual joint displacements of the joints of the rigid-body using the **BodyPosition()** function. A variable number of **p** arguments is expected, equal to the number of degrees of freedom of the rigid-body type selected using **SetBodyType()**.

Since the **BodyPosition()** function directly indicates joint displacements of a rigid-body through its argument list, it may be combined only with the **BodyVelocity()** and **BodyAcceleration()** function calls within a given trajectory. More specifically this function should not be used with any of the following functions **MoveToFrame()**, **MoveToRelation()**, **MoveToTrajectory()** or **SetVelocity()**.

If no trajectory is currently open when a call is made to **BodyPosition()**, the call is ignored. If no rigid body has been selected using **SetBodyType()**, the call is ignored.

See Also

SetBodyType(), **OpenTrajectory()**, **BodyVelocity()**, **BodyAcceleration()**

Example

```
#include "rigid.h"

main()
{
   trajectory_t *soln;

   :
   SetBodyType(PUMA560);
   OpenTrajectory("Good Move");
      BodyPosition(10.0, 15.0, 45.0, 5.0, 10.0, 6.0);
   :
}
```

SetVelocity (TG)

Summary

```
#include "rigid.h"

void SetVelocity(v)
velocity_t *v;
```

Description

The SetVelocity() function constrains the next knot with the velocity given by the v argument in the currently open trajectory. The velocity v must be previously created using the BuildVelocity() function. The velocity constraint does not take effect until the next frame in the trajectory is defined either by MoveToFrame(), MoveToRelation() or MoveToTrajectory(). A call to SetVelocity() without a proceeding call to one of the previously mentioned three functions results in a fatal execution error.

The number of velocity constraints which may be imposed on a trajectory depends upon the amount of main memory available for dynamic allocation.

If no trajectory is currently open when a call to SetVelocity() is made, the call is ignored. If the v argument is undefined, the call to SetVelocity() is ignored.

See Also

BuildVelocity(), OpenTrajectory()

Example

```
#include "rigid.h"

main()
{
    frame_t *rest;
    velocity_t *slow;

    :
    OpenTrajectory("Good Move");
        SetVelocity(slow);
        MoveToFrame(rest);
    :
}
```

BodyVelocity (TG)

Summary

```
#include "rigid.h"

void BodyVelocity(v, ...)
double v;
```

Description

The BodyVelocity() function constrains the next knot with the joint velocities given by the v arguments in the currently open trajectory. A variable number of v arguments is expected, equal to the number of degrees of freedom of the rigid-body type selected using SetBodyType(). A call to BodyVelocity() without a proceeding call to BodyPosition() results in a fatal execution error; although one call to BodyAcceleration() is permitted for each call to BodyVelocity().

Since the BodyVelocity() function directly indicates joint rate displacements of a rigid-body through its argument list, it may be combined only with the BodyPosition() and BodyAcceleration() function calls within a given trajectory. More specifically this function should not be used with any of the following functions MoveToFrame(), MoveToRelation(), MoveToTrajectory() or SetVelocity().

If no trajectory is currently open when a call is made to BodyVelocity(), the call is ignored. If no rigid body has been selected using SetBodyType(), the call is ignored.

See Also

SetBodyType(), OpenTrajectory(), BodyPosition()

Example

```
#include "rigid.h"

main()
{
   trajectory_t *soln;

   :
   SetBodyType(PUMA560);
   OpenTrajectory("Good Move");
      BodyVelocity(0.0, 10.0, 0.0, 0.0, 0.0, 0.0);
   :
}
```

BodyAcceleration (TG)

Summary

```
#include "rigid.h"

void BodyAcceleration(a, ...)
double a;
```

Description

The BodyAcceleration() function constrains the next knot with the joint accelerations given by the a arguments in the currently open trajectory. A variable number of a arguments is expected, equal to the number of degrees of freedom of the rigid-body type selected using SetBodyType(). A call to BodyAcceleration() without a proceeding call to BodyPosition() results in a fatal execution error; although one call to BodyVelocity() is permitted for each call to BodyAcceleration().

Since the BodyAcceleration() function directly indicates joint velocities of a rigid-body through its argument list, it may be combined only with the BodyPosition() and BodyVelocity() function calls within a given trajectory. More specifically this function should not be used with any of the following functions MoveToFrame(), MoveToRelation(), MoveToTrajectory() or SetVelocity().

If no trajectory is currently open when a call is made to BodyAcceleration(), the call is ignored. If no rigid body has been selected using SetBodyType(), the call is ignored.

See Also

SetBodyType(), OpenTrajectory(), BodyPosition(), BodyAcceleration()

Example

```
#include "rigid.h"

main()
{
   trajectory_t *soln;

   :
   SetBodyType(PUMA560);
   OpenTrajectory("Good Move");
      BodyVelocity(0.0, 10.0, 0.0, 0.0, 0.0, 0.0);
      BodyAcceleration(0.0, 30.0, 10.0, 0.0, 0.0, 0.0);
   :
}
```

CloseTrajectory (TG)

Summary

```
#include "rigid.h"

trajectory_t *CloseTrajectory(void)
```

Description

A call to CloseRelation() terminates the definition of the currently open trajectory.

If no trajectory has been opened prior to CloseTrajectory() an error condition is generated and a NIL pointer is returned.

Return Value

A pointer to the newly formed trajectory is returned. A NIL pointer value is returned if no trajectory is currently open.

See Also

OpenTrajectory()

Example

```
#include "rigid.h"

main()
{
   trajectory_t *path;

   OpenTrajectory("Good Move");
   :
   path = CloseTrajectory();
}
```

EvalTrajectory (TG)

Summary

```
#include "rigid.h"

void EvalTrajectory(t)
trajectory_t *t;
```

Description

EvalTrajectory() splines the trajectory specified by the t argument using a constrained B-spline formulation. The trajectory is evaluated at a rate set by the SetSampleSize() function. The trajectory is formed in either CARTESIAN or JOINT-interpolated mode depending on the SetMotionMode() function parameters. The output is reported in the coordinate system selected by the SetCoordSys() function. Finally, the SelectBody() and SetBodyConfig() functions determine the geometry the trajectory posesses. Each SetVelocity() is internally used to split the total trajectory into segments of knots. Each MoveToFrame(), MoveToRelation() or MoveToTrajectory() creates a knot in the B-spline function.

If the t argument is undefined, the call to EvalTrajectory() is ignored.

See Also

OpenTrajectory()

Example

```
#include "rigid.h"

main()
{
    trajectory_t *path;

    :
    path = CloseTrajectory();
    EvalTrajectory(path);
    :
}
```

EnquireTrajectory (TG)

Summary

```
#include "rigid.h"

double EnquireTrajectory(t)
trajectory_t *t;
```

Description

EnquireTrajectory() makes a pass through the B-spline trajectory specified by the t argument to make an estimate of the time dilation factor required to satisfy the velocity and acceleration constraints of the rigid body specified by the SetVelConstraint() and SetAccConstraint() functions. The computed time dilation factor value is returned by the function call. This function also serves as a switch to indicate to WriteTrajectory() that time dilation should be activated when generating output results and plots.

If the t argument is undefined, the call to EnquireTrajectory() is ignored.

See Also

OpenTrajectory(), SetVelConstraint(), SetAccConstraint()

Example

```
#include "rigid.h"

main()
{
   trajectory_t *path;

   :
   path = CloseTrajectory();
   printf("Dilation: %lf\n", EnquireTrajectory(path));
   :
}
```

SetCoordSys (TG)

Summary

```
#include "rigid.h"

void SetCoordSys(fun)
coord_t fun;
```

Description

A call to SetCoordSys() selects the coordinate system to be used in reporting functions such as WriteRelation() and WriteTrajectory(). Since the input frames may be of mixed coordinate system types from BuildFrame(), it is desirable to view the output in one consistent coordinate system type. Refer to BuildFrame() for more details on the transformation function options for the **fun** argument, and procedures for providing user-defined transformation procedures.

If the **fun** argument is undefined, the default coordinate system (RPY) is maintained and the call to SetCoordSys() is ignored.

See Also

EvalTrajectory(), BuildFrame()

Example

```
#include "rigid.h"

main()
{
    :
    SetCoordSys(EUL);
    :
}
```

SetSampleStep (TG)

Summary

```
#include "rigid.h"

void SetSampleStep(step)
double step;
```

Description

A call to **SetSampleStep()** selects the frame frequency for which the EvalTrajectory() function reports motion along the splined trajectory. The valid range of the step argument occupies the range 0.0 to 1.0. The default step, 1.0, reports each frame in the trajectory; smaller step values allow reporting inbetween frames.

If the step argument is beyond the allowable range, it is maintained at the default of 1.0 and the call to **SetSampleStep()** is ignored.

See Also

EvalTrajectory()

Example

```
#include "rigid.h"

main()
{
    SetSampleStep(0.25);
    :
}
```

SetMotionMode (TG)

Summary

```
#include "rigid.h"

void SetMotionMode(mode)
int *mode;
```

Description

A call to SetMotionMode() determines whether to spline the trajectory in straight-line mode or joint-interpolated mode. The mode argument indicates the trajectory splining mode by setting it to one of the following constants:

CARTESIAN	straight-line interpolation
JOINT	joint-interpolation

If the mode argument is undefined, the default mode (CARTESIAN) is maintained and the call to SetMotionMode() is ignored.

See Also

EvalTrajectory()

Example

```
#include "rigid.h"

main()
{
    :
  SetMotionMode(JOINT);
    :
}
```

SetBodyType (TG)

Summary

```
#include "rigid.h"

void SetBodyType(bfun)
void (*bfun)();
```

Description

The geometry of the rigid body RIGID generates a trajectory for must be selected by SetBodyType(). The bfun argument may be set to one of the following predefined bodies:

NOBODY	the default one-to-one correspondence
PUMA560	the PUMA 560 robot model (mm)

The user may provide some other type of kinematic rigid body model by using the following steps. Define the forward and inverse kinematic model functions of the rigid body. Define the macro which selects the functions and sets the degrees of freedom and the number of configuration parameters used by the inverse kinematic function. Refer to the rigid.h header file for examples.

The PUMA560 body accepts measurement in mm, NOBODY uses dimensionless units.

If the bfun argument is undefined, the call to SetBodyType() is ignored.

See Also

EvalTrajectory(), SetBodyConfig()

Example

```
#include "rigid.h"

main()
{
    :
    SetBodyType(PUMA560);
    :
}
```

SetBodyConfig (TG)

Summary

```
#include "rigid.h"

void SetBodyConfig(p, ...)
int p;
```

Description

For those rigid bodies that may contain many multiple joint-articulated configurations to any given point in Cartesian space, a call to SetBodyConfig() specifically selects one initial rigid body configuration. A variable number of p arguments is possible. Refer to SetBodyType() for details on the allowable configurations and number of configuration parameters for each of the rigid body types.

For user-defined rigid body kinematic models, the inverse model supplied must have the necessary provisions for the configuration parameters specified in the argument list for the SetBodyConfig() function to have any releyance.

See Also

SetBodyType(), GetBodyConfig()

Example

```
#include "rigid.h"

main()
{
    :
    SetBodyType(PUMA560);
    SetBodyConfig(-1, 1, -1);
    :
}
```

GetBodyConfig (TG)

Summary

```
#include "rigid.h"

void GetBodyConfig(p, ...).
int *p;
```

Description

For those rigid bodies that may contain many multiple joint-articulated configurations to any given point in Cartesian space, a call to **GetBodyConfig()** specifically selects one initial rigid body configuration. A variable number of p arguments is possible. Refer to **GetBodyType()** for details on the allowable configurations and number of configuration parameters for each of the rigid body types.

For user-defined rigid body kinematic models, the inverse model supplied must have the necessary provisions for the configuration parameters specified in the argument list for the **GetBodyConfig()** function to have any relevance.

See Also

SetBodyType(), SetBodyConfig()

Example

```
#include "rigid.h"

main()
{
    int k1, k2, k3;
    :
    SetBodyType(PUMA560);
    GetBodyConfig(&k1, &k2, &k3);
    :
}
```

SetVelConstraint (TG)

Summary

```
#include "rigid.h"

void SetVelConstraint(v, ...)
int v;
```

Description

The velocity constraints for each joint of a rigid body may be specified with the SetVelConstraint() function. The velocity argument list v accepts as many values as there are degrees of freedom (DOF) in the rigid body.

It is assumed that a rigid body has no physical velocity constraints unless each call to SetBodyType() is followed by a call to SetVelConstraint().

See Also

SetBodyType()

Example

```
#include "rigid.h"

main()
{
    :
    SetBodyType(PUMA560);
    SetVelConstraint(100.0, 100.0, 100.0, 80.0, 80.0, 80.0);
    :
}
```

SetAccConstraint (TG)

Summary

```
#include "rigid.h"

void SetAccConstraint(a, ...)
int a;
```

Description

The acceleration constraints for each joint of a rigid body may be specified with the SetAccConstraint() function. The acceleration argument list a accepts as many values as there are degrees of freedom (DOF) in the rigid body.

It is assumed that a rigid body has no physical acceleration constraints unless each call to SetBodyType() is followed by a call to SetAccConstraint().

See Also

SetBodyType()

Example

```
#include "rigid.h"

main()
{
    :
    SetBodyType(PUMA560);
    SetAccConstraint(60.0, 60.0, 60.0, 40.0, 40.0, 40.0);
    :
}
```

SetOutputMode (RF)

Summary

```
#include "rigid.h"

void SetOutputMode(omode)
int omode;
```

Description

The style of output is determined by SetOutputMode(). The omode argument may be set to one of the following modes:

MATRIX	output in matrix style
VECTOR	output in vector style

After a call to SetOutputMode(), all subsequent calls to either WriteRelation() will print results from RIGID in matrix form, or vector form depending on the status of omode.

The default output mode is MATRIX.

See Also

WriteRelation(), WriteTrajectory()

Example

```
#include "rigid.h"

main()
{
    :
  SetOutputMode(VECTOR);
    :
}
```

WriteMatrix (RF)

Summary

```
#include "rigid.h"

void WriteMatrix(fmt, m)
char *fmt;
double *m;
```

Description

The **WriteMatrix()** function is similar to the **printf()** function but is designed sspecifically for printing a 4x4 transformation matrix. It is derived from the stdio (C language standard input/output) library, and prints to the standard output (**stdout**). For the matrix specified by the **m** argument, the proper 4x4 memory space of type **double** must be allocated.

The **fmt** argument is a character string that contains the output format specification for the data in the frame f. The first character of fmt should be %, followed by:

a) an optional flag: - left justify field,
+ always print numeric sign,
# always print decimal point,
0 zero-padd numeric on left,

b) an optional minimum width field width integer

c) an optional precision integer specifying the number of digits after the decimal (including 0) this integer must be preceded by a decimal point

d) conversion type: lf double precision style [-]ddd.ddd, default precision 6,
le double precision style [-]d.dddde±dd, default precision 6

More details on all of the possible formats for **printf()** can be found in C programming texts [120, 123, 125, 130].

In general, errors in the specification of the **fmt** argument are not gracefully handled by the stdio library.

Example

```
#include "rigid.h"

main()
{
    double test[4][4];
    :
    EvalRelation((double *) test, unknown);
    WriteMatrix("%6.3lf", (double *) test);
    :
}
```

WriteVector (RF)

Summary

```
#include "rigid.h"

void WriteVector(fmt, v)
char *fmt;
double *v;
```

Description

The **WriteVector()** function is similar to the **printf()** function but is designed for printing a 6 element positional/rotational vector.

In general, errors in the specification of the **fmt** argument are not gracefully handled by the stdio library.

See Also

WriteMatrix()

Example

```
#include "rigid.h"

main()
{
   double p[6];

   :
   UnBuildFrame(rest,RPY,&p[0],&p[1],&p[2],&p[3],&p[4],&p[5]);
   WriteVector("%6.2lf", (double *) p);
   :
}
```

WriteFrame (RF)

Summary

```
#include "rigid.h"

void WriteFrame(fmt, nf, f, ...)
char *fmt;
int nf;
frame_t *f;
```

Description

The WriteFrame() function is similar to the printf() function but is designed for printing frames. It is derived from the stdio library, and prints to the standard output (stdout). A variable number of f arguments is possible, the exact number of frames is specified in the nf argument. For each frame specified in the f argument space the complete frame data is printed, including name, forward and inverse transformation matrices. Details on the fmt argument are described in WriteMatrix()

In general, errors in the specification of the fmt argument are not gracefully handled by the stdio library.

See Also

WriteMatrix(), BuildFrame()

Example

```
#include "rigid.h"

main()
{
   frame_t *rest, *run;

   rest=BuildFrame("Resting",RPY,50.0,30.0,0.0,90,0,0.0,-45.0);
   run=BuildFrame("Run",  RPY,10.0,-30.0,0.0,0.0,0.0,-45.0);
   WriteFrame("%6.2lf", 2, rest, run);
   :
}
```

WriteVelocity (RF)

Summary

```
#include "rigid.h"

void WriteVelocity(fmt, nv, v, ...)
char *fmt;
int nf;
velocity_t *v;
```

Description

The **WriteVelocity()** function is similar to the **printf()** function but is designed for printing velocity constraints. It is derived from the stdio library, and prints to the standard output (**stdout**). A variable number of **v** arguments is possible, the exact number of velocities is specified in the **nv** argument. For each velocity constraint specified in the **v** argument space the complete constraint data is printed, including name and linear/angular velocity vectors. Details on the **fmt** argument are described in **WriteMatrix()**.

In general, errors in the specification of the **fmt** argument are not gracefully handled by the stdio library.

See Also

WriteMatrix(), **BuildVelocity()**

Example

```
#include "rigid.h"

main()
{
   velocity_t *slow, *fast;

   slow=BuildVelocity("Slow   pace",RPY,50.0,30.0,0.0,90.0,0.0,45.0);
   fast=BuildVelocity("Fast   pace",RPY,10.0,-30.0,0.0,0.0,0.0,-45.0);
   WriteVelocity("%6.2lf", 2, slow, fast);
   :
}
```

WriteRelation (RF)

Summary

```
#include "rigid.h"

void WriteRelation(fmt, nr, r, ...)
char *fmt;
int nr;
relation_t *r;
```

Description

The WriteRelation() function is similar to the printf() function but is designed for printing relations. It is derived from the stdio library, and prints to the standard output (stdout). A variable number of r arguments is possible, the exact number of relations is specified in the nr argument. For each relation specified in the r argument space the complete relation data is printed, including name and transformation solution matrix. Details on the fmt argument are described in WriteMatrix().

In general, errors in the specification of the fmt argument are not gracefully handled by the stdio library.

See Also

WriteMatrix(), SetOutputMode(), SetCoordSys(), CloseRelation()

Example

```
#include "rigid.h"

main()
{
   relation_t *unknown;

   :
   unkown = CloseRelation();
   WriteRelation("%6.2lf", 1, unknown);
}
```

WriteTrajectory (RF)

Summary

```
#include "rigid.h"

void WriteTrajectory(fmt, nt, t, ...)
char *fmt;
int nt;
trajectory_t *t;
```

Description

The WriteTrajectory() function is similar to the printf() function but is designed for printing trajectories. It is derived from the stdio library, and prints to the standard output (stdout). A variable number of t arguments is possible, the exact number of trajectories is specified in the nt argument. For each trajectory specified in the t argument space the complete trajectory data is printed; including name and joint motions. Details on the fmt argument are described in WriteMatrix().

In general, errors in the specification of the fmt argument are not gracefully handled by the stdio library.

See Also

WriteMatrix(), SetOutputMode(), CloseTrajectory()

Example

```
#include "rigid.h"

main()
{
    trajectory_t *path;

    :
    path = CloseTrajectory();
    WriteTrajectory("%6.2lf", 1, path);
}
```

```
/*
** File:        rigid.h
** Date:        March 1, 1988
** Author:      Stuart E. Thompson
** Purpose:     RIGID data types and function declarations
*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <malloc.h>

/* RIGID keywords. */
#define RHS             1       /* side of relation */
#define LHS             2       /* side of relation */
#define MATRIX          1       /* reporting mode */
#define VECTOR          2       /* reporting mode */
#define CARTESIAN       1       /* interpolation mode */
#define JOINT           2       /* interpolation mode */

/* Rigid-body types. */
#define NOBODY          1       /* NULL rigid-body */
#define PUMA560         2       /* PUMA 560 robot */

/* Data structure identifier types. */
#define FRMTYPE         1       /* frame type indicator */
#define BDYTYPE         2       /* body type indicator */
#define RELTYPE         3       /* relation type indicator */
#define TRAJTYPE        4       /* trajectory type indicator */
#define NIL             0L      /* linked-list terminator */
#define SIZENAME        81      /* length of name variables */
#define SIZEMSG         81      /* length of message variables */

/* Common mathematical constants. */
#define PI              3.14159         /* constant PI */
#define TWOPI           2.0*PI          /* constant 2PI */
#define PIBY2           PI/2.0          /* constant PI/2 */
#define DEGTORAD        PI/180.0        /* degree to radian */
#define RADTODEG        180.0/PI        /* radian to degree */
#define TRIGERR         0.001           /* trigonometric tolerance */

/* Simulate boolean data type. */
typedef int boolean;
#define TRUE            1
#define FALSE           0

/* Fake data types. */
#define private
#define public

/* B-spline of order (degree + 1) five. */
#define ORDER           5
#define ORDERP1         (ORDER + 1)
```

```
/* RIGID data structures. */
typedef struct coord {
        char *cname;                    /* coordinate function name */
        void (*fun[2])();               /* coordinate specification function */
} coord_t;
#define SIZECOORD       sizeof(coord_t)

typedef struct frame {
        char fname[SIZENAME];           /* frame name */
        double vec[6];                  /* translation/rotation angles */
        double fwd[4][4];               /* forward transformation matrix of frame */
        double inv[4][4];               /* inverse transformation matrix of frame */
        struct coord *ffun;             /* coordinate specification function */
} frame_t;
#define SIZEFRAME       sizeof(frame_t)

typedef struct velocity {
        char vname[SIZENAME];           /* velocity name */
        double vel[6];                  /* linear/angular velocity */
        struct coord *vfun;             /* coordinate specification function */
} velocity_t;
#define SIZEVELOCITY    sizeof(velocity_t)

typedef struct part {
        int side;                       /* data appears on LHS or RHS */
        int ptype;                      /* pptr type indicator for union */
        union {
                struct frame *frm;      /* pointer to a frame */
                struct relation *rel;   /* pointer to another relation */
        } pptr;
        struct part *nextpart;          /* pointer to next part in part-list */
} part_t;
#define SIZEPART        sizeof(part_t)

typedef struct relation {
        char rname[SIZENAME];           /* relation name */
        struct part *rdata;             /* pointer to first part in part-list */
        struct part *reol;              /* pointer to last part in part-list */
        struct relation *nextrel;       /* pointer to next relation in rel-list */
} relation_t;
#define SIZEREL         sizeof(relation_t)

typedef struct knot {
        int ktype;                      /* kptr type indicator for union */
        union {
                double *bpos;           /* pointer to a body vector */
                struct frame *frm;      /* pointer to a frame */
                struct relation *rel;   /* pointer to another relation */
                struct trajectory *traj; /* pointer to a trajectory */
        } kptr;
        struct knot *nextknot;          /* pointer to next knot in knot-list */
} knot_t;
#define SIZEKNOT        sizeof(knot_t)
```

```
typedef struct spline {
        int numKnots;                   /* number of extended knots in curve */
        int dof;                        /* number of curve families */
        int numPts;                     /* number of points along basis vector */
        double *knots;                  /* knot locations along the basis vector */
        double *coef;                   /* the B-spline coefficients */
        double *basis;                  /* the B-spline basis vector */
} spline_t;
#define SIZESPLINE          sizeof(spline_t)

typedef struct segment {
        int numKnots;                   /* number of knots in knot-list */
        boolean isvel;                  /* TRUE if velocity set */
        boolean isacc;                  /* TRUE if acceleration set */
        struct spline *path;            /* the knots in splined form */
        int stype;                      /* sptr type indicator for union */
        union {
                double *bvel;           /* pointer to body velocity */
                struct velocity *vel;   /* velocity of first knot in knot-list */
        } sptr;
        double *bacc;                   /* pointer to body acceleration */
        struct knot *sdata;             /* pointer to first knot in knot-list */
        struct knot *seol;              /* pointer to last knot in knot-list */
        struct segment *nextseg;        /* pointer to next segment in seg-list */
} segment_t;
#define SIZESEG             sizeof(segment_t)

typedef struct trajectory {
        char tname[SIZENAME];           /* trajectory name */
        int numSegs;                    /* length of seg-list */
        int mode;                       /* CARTESIAN or JOINT */
        double dilation;                /* overall time dilation factor */
        struct segment *tdata;          /* pointer to first segment in seg-list */
        struct segment *teol;           /* pointer to last segment in seg-list */
        struct trajectory *nextraj;     /* pointer to next trajectory in traj-list */
} trajectory_t;
#define SIZETRAJ            sizeof(trajectory_t)

typedef struct node {
        char *item;                     /* data to be queued */
        struct node *nextnode;          /* pointer to next node in queue */
} node_t;
#define SIZENODE            sizeof(node_t)

typedef struct fifo {
        int length;                     /* length of queue */
        struct node *front;             /* front of queue (dequeue) */
        struct node *rear;              /* rear of queue (enqueue) */
} fifo_t;
#define SIZEFIFO            sizeof(fifo_t)

typedef struct body {
        int dof;                        /* degrees of freedom */
        int numConfig;                  /* number of configuration parameters */
```

```
        int *config;                    /* configuration parameters */
        double p[6];                    /* Cartesian position/orientation vector */
        double v[6];                    /* Cartesian linear/angular vel. vector */
        double a[6];                    /* Cartesian linear/angular acc. vector */
        double *qp;                     /* joint displacement vector */
        double *qv;                     /* joint velocity vector */
        double *qa;                     /* joint acceleration vector */
        struct coord bfun;              /* motion transform functions */
        boolean isMaxVel;               /* velocity constraints set flag */
        double *maxVel;                 /* velocity constraints */
        boolean isMaxAcc;               /* acceleration constraints set flag */
        double *maxAcc;                 /* acceleration constraints */
} body_t;
#define SIZEBODY        sizeof(body_t)

typedef struct errtrap {
        char errmsg[SIZEMSG];           /* error description */
        int numErrors;                  /* error count */
        boolean fatal;                  /* fatal error flag */
} errtrap_t;

/* Function definitions on a per file basis. */
/* rel.c */
public boolean          OpenRelation();
public relation_t       *CloseRelation();
public void             EquateFrame();
public void             EquateRelation();
public void             EvalRelation();
/* options.c */
public void             SetCoordSys();
public void             SetSampleStep();
public void             SetMotionMode();
public void             SetBodyType();
public void             SetBodyConfig();
public void             GetBodyConfig();
public void             SetVelConstraints();
public void             SetAccConstraints();
private void            SetError();
public void             ReportError();
public void             ClearError();
/* traj.c */
public boolean OpenTrajectory();
public trajectory_t     *CloseTrajectory();
public void             SetVelocity();
public void             BodyVelocity();
public void             BodyAcceleration();
public void             MoveToFrame();
public void             MoveToRelation();
public void             MoveToTrajectory();
public void             BodyPosition();
private void            FormTrajectory();
private void            SplineTrajectory();
private void            ExtractPos();
public void             EvalTrajectory();
```

```
public double       EnquireTrajectory();
/* queue.c */
private boolean     EnQueue();
private char        *DeQueue();
private void        ClearQueue();
private fifo_t      *InstQueue();
private node_t      *InstNode();
/* prim.c */
public velocity_t   *BuildVelocity();
public void         UnBuildVelocity();
public frame_t      *BuildFrame();
public void         UnBuildFrame();
public void         UpdateFrameAbs();
public void         UpdateFrameRel();
/* xforms.c */
private void        FwdRPY();
private void        InvRPY();
private void        FwdEUL();
private void        InvEUL();
private void        FwdOAT();
private void        InvOAT();
private void        HMMul();
private void        HMInv();
private void        HMCpy();
private void        HMAdd();
private void        HMEye();
/* report.c */
public void         SetOutputMode();
public void         WriteVector();
public void         WriteMatrix();
public void         WriteFrame();
public void         WriteVelocity();
public void         WriteRelation();
public void         WriteTrajectory();
/* evaluate.c */
private void        Evaluate();
/* fillrow.c */
private int         FillRow();
/* formset.c */
private void        FormSet();
/* formsys.c */
private void        FormSys();
/* solvesys.c */
private void        SolveSys();
/* spline.c */
private void        FormSpline();
private void        UnCnstrTop();
private void        UnCnstrBot();
private double      DilateSpline();
public void         EvalSpline();
/* instance.c */
private coord_t     *InstCoord();
private relation_t  *InstRelation();
private part_t      *InstPart();
```

```
private frame_t      *InstFrame();
private velocity_t   *InstVelocity();
private trajectory_t *InstTrajectory();
private segment_t    *InstSegment();
private knot_t       *InstKnot();
private body_t       *InstBody();
private spline_t     *InstSpline();
private double       *InstDArray();
private int          *InstIArray();
/* utils.c */
private int          imax();
private int          imin();
private double       dmax();
private double       dmin();
private void         OpenPlotFiles();
private void         WritePlotFiles();
private void         ClosePlotFiles();
/* bodies.c */
private void         FwdNULL();
private void         InvNULL();
private void         FwdPUMA();
private void         InvPUMA();

/* Initialize some predeclared data structures. */
static coord_t       RPY = {"RPY", FwdRPY, InvRPY};
static coord_t       EUL = {"EUL", FwdEUL, InvEUL};
static coord_t       OAT = {"OAT", FwdOAT, InvOAT};

/* Set global variables for RIGID. */
int          motionMode;                  /* CARTESIAN or JOINT */
int          outputMode;                  /* MATRIX or VECTOR */
double       motionStep;                  /* range 0.0 to 1.0 */
coord_t      motionXfun;                  /* RPY, EUL or OAT */
body_t       bodyType;                    /* NOBODY or PUMA560 */
static errtrap_t errigid={NULL,0,FALSE};  /* error handler */
FILE         *fp, *fv, *fa;               /* plotting file pointers */
```

Appendix B - C Programming Language Syntax Summary

This syntax summary [132] is intended for aiding comprehension of the syntax of the C programming language, it is not a tutorial by any means. C language keywords are highlighted in bold.

Expressions

The basic expressions are:

expression:
- primary
- *expression
- &lvalue
- -expression
- !expression
- ~expression
- ++lvalue
- --lvalue
- lvalue++
- lvalue--
- **sizeof** expression
- (type-name) expression
- expression binop expression
- expression ? expression : expression
- lvalue asgnop expression
- expression, expression

primary:
- identifier
- constant
- string
- (expression)
- primary (expression-list$_{opt}$)
- primary [expression]
- primary.identifier
- primary->identifier

lvalue:
- identifier
- primary [expression]
- lvalue.identifier
- primary->identifier
- *expression
- (lvalue)

The primary-expression operators have highest priority and group left to right.

() [] . ->

The unary operators have priority below the primary operators but higher than any binary operator and group right to left.

* & - ! ~ ++ -- **sizeof** (type-name)

Binary operators group left to right; they have priority decreasing as indicated below. The conditional operator groups right to left.

binop:

```
*    /    %
+    -
>>   <<
<    >    <=   >=
==   !=
&
^
|
&&
||
?:
```

Assignment operators all have the same priority and all group right to left

asgnop:

```
=    +=   -=   *=   /=   %=   >>=  <<=
&=   ^=   !=
```

The comma operator has the lowest priority and groups left to right.

Declarations

declaration:

decl-specifiers init-declarator-list$_{opt}$;

decl-specifiers:

type-specifier decl-specifiers$_{opt}$

sc-specifier decl-specifiers$_{opt}$

sc-specifier:

auto
static
extern
register
typedef

type-specifier:
char
short
int
long
unsigned
float
double
void
struct-or-union-specifier

typedef-name enum-specifier
enum-specifier:
enum {enum-list}
enum identifier {enum-list}
enum identifier

enum-list:
enumerator
enum-list, enumerator

enumerator:
identifier
identifier = constant-expression

init-declarator-list
init-declarator
init-declarator, init-declarator-list

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
*declarator
declarator()
declarator [constant-expression$_{opt}$]

struct-or-union-specifier:
struct {struct-decl-list}
struct identifier {struct-decl-list}
struct identifier
union {struct-decl-list}
union identifier {struct-decl-list}
union identifier

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specier struct-declarator-list;

struct-declarator-list:
struct-declarator
struct-declarator, struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= expression
= {initializer-list}
= {initializer-list,}

initializer-list:
expression
initializer-list, initjalizer-list
{initializer-list}

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
*abstract-declarator
abstract-declarator()
abstract-declarator [constant-expression$_{opt}$]

typedef-name:
identifier

Statements

compound-statement:
{declaration-list$_{opt}$ statement-list$_{opt}$}

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression;
if (expression) statement

if (expression) statement **else** statement
while (expression) statement
do statement **while** (expression);
for (expression-1$_{opt}$; expression-2$_{opt}$; expression-3$_{opt}$) statement
switch (expression) statement
case constant-expression: statement
default : statement
break;
continue;
return;
return expression;
goto identifier;
identifier: statement
;

External Definitions

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition
type-specifier$_{opt}$ function-declarator function-body

function-declarator:
declarator (parameter-list$_{opt}$)

parameter-list:
identifier
identifier, parameter-list

function-body:
declaration-list compound-statement

data-definition:
extern$_{opt}$ type-specifier$_{opt}$ init-declarator-list$_{opt}$;
static$_{opt}$ type-specifier$_{opt}$ init-declarator-list$_{opt}$;

Preprocessor

#define identifier token-string
#define identifier (identifier, ..., identifier) token-string

```
#undef identifier
#include "filename"
#include <filename>
#if constant-expression
#ifdef identifier
#ifndef identifier
#else
#endif
#line constant "filename"
```

Appendix C - Frame Transformations

A summary of the equations for the frame transformations are given. More specifically, the forward and inverse kinematic equations for Roll-Pitch-Yaw (RPY), Euler (EUL) and Unimation PUMA (OAT) angles are given. More elaborate explanations of the transforms are provided in [1, 7].

Roll-Pitch-Yaw

Each of the RPY rotations takes place about a fixed reference frame {A}. The rotation about $\hat{X}_A$ by angle γ is called roll, rotation about $\hat{Y}_A$ by β called pitch and about $\hat{Z}_A$ by α called yaw. See figure C.1. Therefore,

$$R(\gamma,\beta,\alpha) = \begin{pmatrix} C\alpha C\beta & C\alpha S\beta S\gamma - S\alpha C\gamma & C\alpha S\beta C\gamma + S\alpha S\gamma \\ S\alpha C\beta & S\alpha S\beta S\gamma + C\alpha C\gamma & S\alpha S\beta C\gamma - C\alpha S\gamma \\ -S\beta & C\beta S\gamma & C\beta C\gamma \end{pmatrix}$$

$$\gamma = \text{atan2}(r_{32}, r_{33})$$

$$\beta = \text{atan2}(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2})$$

$$\alpha = \text{atan2}(r_{21}, r_{11})$$

If $\beta = 90.0$ degrees, then $\gamma = \text{atan2}(r_{12}, r_{22})$, $\beta = 90.0$, $\alpha = 0.0$. Conversely, if $\beta = -90.0$ degrees, then $\gamma = -\text{atan2}(r_{12}, r_{22})$, $\beta = -90.0$, $\alpha = 0.0$.

Euler Angles

Each of the EUL rotations takes place about a moving reference frame {B}. The rotations are consecutively made about about $\hat{Z}_B$ by α, rotation about $\hat{Y}_B$ by β and about $\hat{Z}_B$ by γ. See figure C.2. Therefore,

$$R(\alpha,\beta,\gamma) = \begin{pmatrix} C\alpha C\beta C\gamma - S\alpha S\gamma & -C\alpha C\beta S\gamma - S\alpha C\gamma & C\alpha S\beta \\ S\alpha C\beta C\gamma + C\alpha S\gamma & -S\alpha C\beta S\gamma + C\alpha S\gamma & S\alpha S\beta \\ -S\beta C\gamma & S\beta S\gamma & C\beta \end{pmatrix}$$

$$\alpha = \text{atan2}(r_{23}, r_{13})$$

$$\beta = \text{atan2}(\sqrt{r_{31}^2 + r_{32}^2}, r_{33})$$

$$\gamma = \text{atan2}(r_{32}, -r_{31})$$

If $\beta = 0.0$ degrees, then $\gamma = \text{atan2}(-r_{12}, r_{11})$, $\beta = 0.0$, $\alpha = 0.0$.

If $\beta = 180.0$ degrees, then $\gamma = -\text{atan2}(r_{12}, -r_{11})$, $\beta = 180.0$, $\alpha = 0.0$.

Unimation Angles

Each of the Unimation angles are closely related to the EUL angles. The OAT angular measure for the PUMA 560 robot are defined in [10]. They are defined as

$O = \alpha + 90.0$ degrees

$A = \beta - 90.0$ degrees

$T = \gamma$

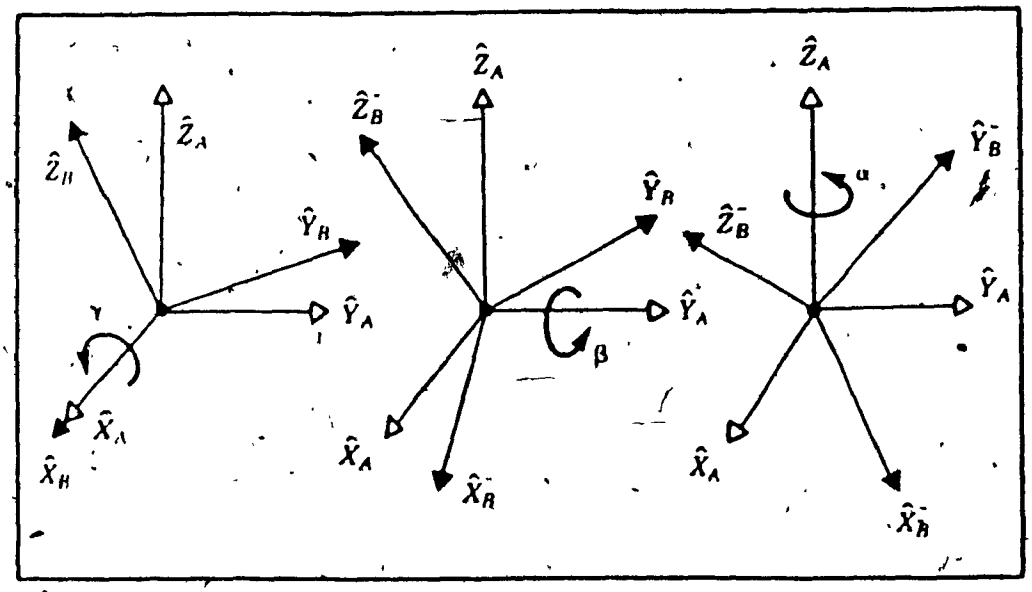


Figure C.1: RPY Angles.

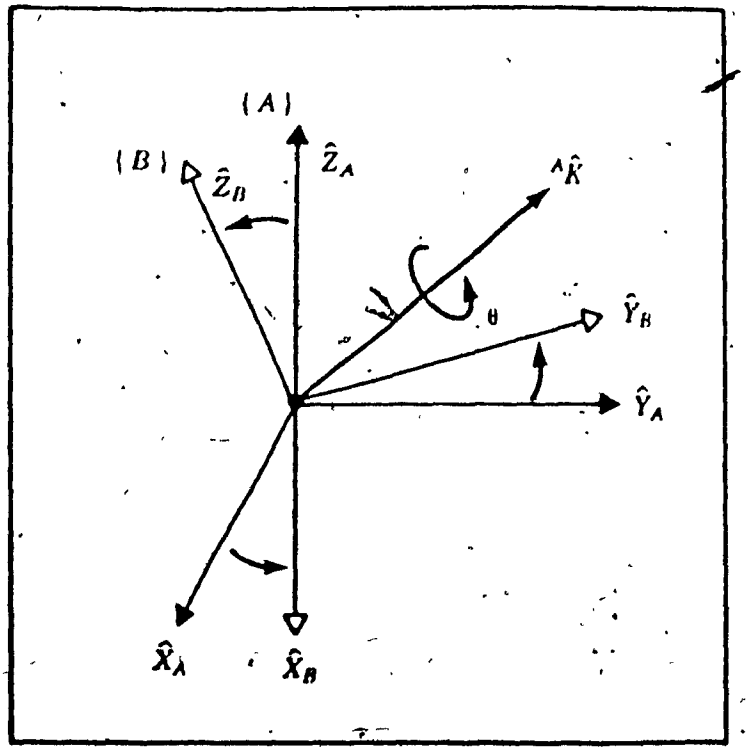


Figure C.2: EUL Angles.

Appendix D - Puma 560 Robot Model

A complete summary of the closed-form equations of motion for the PUMA 560 robot manipulator are given in a paper by Elgazzar [11]. These results are too lengthy to include in this thesis. The physical characteristics of the PUMA 560 robot is given.

The PUMA 560 robot has six revolute joints. Joints 1 through 3 form the arm of the robot, joints 4 through 6 form a spherical wrist. The shoulder may be used in a left or right shoulder configuration. The elbow can have two configurations: elbow up (elbow above shoulder) and elbow down (elbow below shoulder). The wrist may also assume two solutions: the flip wrist (joint 5 angle positive) and flip wrist (joint 5 angle negative).

The configuration control parameters are defined as:

$$k_1 = \begin{cases} +1 \text{ left shoulder} \\ -1 \text{ right shoulder} \end{cases}$$

$$k_2 = \begin{cases} +1 \text{ elbow up} \\ -1 \text{ elbow down} \end{cases}$$

$$k_3 = \begin{cases} +1 \text{ no-flip wrist} \\ -1 \text{ flip wrist} \end{cases}$$

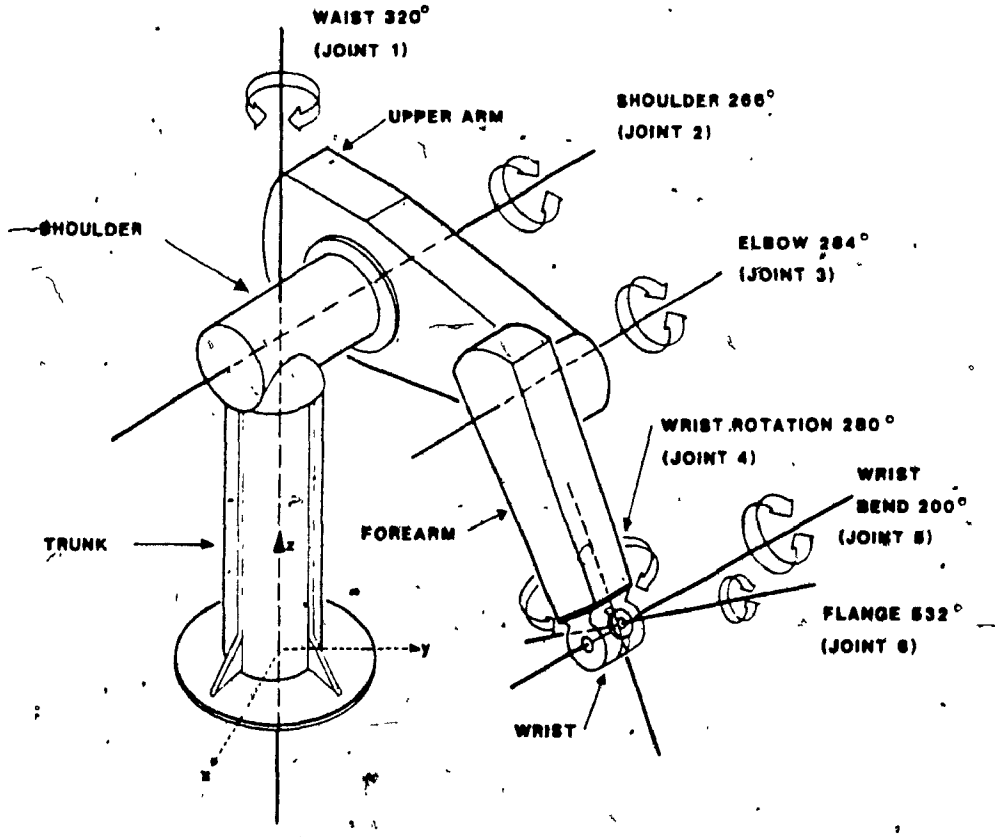


Figure D.1: PUMA 560 Robot.

These parameters are passed to SetBodyConfig(), in respective order, when calculating the inverse kinematic solution for manipulator joint angles. A summary of forward and inverse solution operations is given below. The dynamics operation count assumes that the kinematics has been calculated.

	kinematics	dynamics	
		velocity	acceleration
forward	19a + 25m + 17f	21a + 30m	44a + 52m
inverse	23a + 31m + 16f	23a + 35m + 6f	45a + 59m

(a:additions, m:multiplications, f:transcendental functions)

The link constants of figure A.2 are given below:

l_1	672 mm
l_2	432 mm
l_3	433 mm
l_4	56 mm
$d = d_1 - d_2$	149.1 mm
δ	2.72°.

The ranges for joint angles are given below. Motion requests for robot solutions which voilate any of these limits are trapped by the error handler during trajectory generation.

$-250^\circ \le \theta_1 \le 70^\circ$	(waist joint)
$-133^\circ \le \theta_2 \le 133^\circ$	(shoulder joint)
$-142^\circ \le \theta_3 \le 142^\circ$	(elbow joint)
$-110^\circ \le \theta_4 \le 170^\circ$	(wrist rotation joint)
$-100^\circ \le \theta_5 \le 100^\circ$	(wrist bend joint)
$-176^\circ \le \theta_6 \le 356^\circ$	(flange joint)

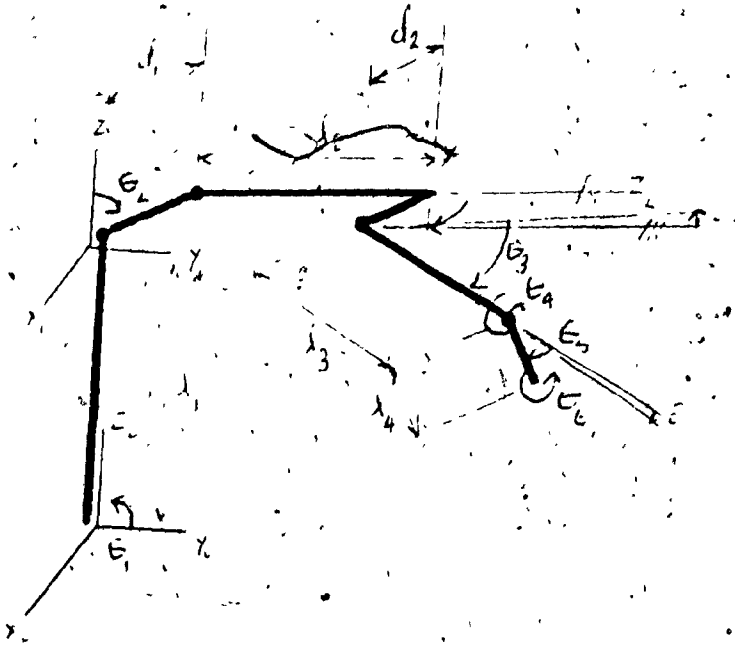


Figure D.2: Link representation of PUMA 560.

Wrist optimization

If θ_4 and / or θ_6 pass beyond their joint limit, the wrist configuration parameter k_3 changes to allow the joint angles to stay within limits. If upon flipping the wrist in the alternate configuration, the joint limit for θ_4 and / or θ_6 is still exceeded then the solution is definitely out of range. Two solutions may exist for θ_6, 360° apart in the region $-176° \leq \theta_6 \leq -4°$ or $184° \leq \theta_6 \leq 356°$, which must be decided based on the joint's previous value.