7-1-1984

Advanced Industrial Robot Control Systems

Richard P. Paul
Purdue University

J. Y. S. Luh
Purdue University

S. Y. Nof
Purdue University

Y. Hayward
Purdue University

Advanced Industrial Robot Control Systems

Richard P. Paul
J.Y.S. Luh
S.Y. Nof
V. Hayward

TR-EE 84-25
July 1984

School of Industrial Engineering
and
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Tenth Report
Covering Period March 1, 1983 to September 1, 1984

ADVANCED INDUSTRIAL ROBOT CONTROL SYSTEMS

Richard P. Paul, J. Y. S. Luh, S. Y. Nof,
and V. Hayward
School of Electrical & Industrial Engineering
Purdue University

TR-84-25

July 1984

Prepared for:

NATIONAL SCIENCE FOUNDATION
WASHINGTON, D. C. 20550

Attention: DR. WILLIAM M. SPURGEON

GRANT: MEA-8119884

This research was supported by the National Science Foundation under grant No. DAR (APR) 77-14533. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

ADVANCED INDUSTRIAL ROBOT CONTROL SYSTEMS

Richard P. Paul, J. Y. S. Luh, S. Y. Nof,
and V. Hayward
School of Electrical & Industrial Engineering
Purdue University

I. PROGRAM OBJECTIVE

The objective of this research is to extend the flexibility and usefulness of current industrial robots by the integration of robot motion control directly into a general purpose programming language, the development of force feedback and its integration into the language, the formulation of a high-level task description language RTM, and by the investigation of both off-line collision-free path planning and on-line collision avoidance.

II. PROGRAM ACHIEVEMENT

Major accomplishments for the first five years of this grant, through March 1983 are:

A. Motion in Joint Coordinates - The initial theoretical development work and simulation of a language system, known as PAL. The major components of PAL included an editor/scanner that allows a user to create, edit, check and store motion procedures; a teach module simulation for single stepping through procedures and for spatial position correction; and a execution module.

B. Minimum Motion Time - By eliminating the need to stop at the end of each path segments and by ensuring that the manipulator moves at maximum velocity and acceleration, the motion time can be reduced. This was accomplished by two optimization methods: An algorithm for direct approximate programming of motions subject to given physical constraints; by dynamically identifying the slowest joint, which then defines the coordination for driving of the other joints.

C. Newton-Euler Formulation of Dynamics and Resolved-Acceleration Control for Manipulators - A new approach to the problem was developed by adopting the idea of the "inverse problem" and extending the results of "resolved-motion rate control". This approach differed in applying all feedback control at the robot hand level, and in its Newton-Euler Formulation of motion equations.

D. RTM, A Technique for Analyzing and Specifying Work for Robots - We have developed a higher level, user oriented technique called RTM (Robot Time and Motion), to systematically specify a work method for a robot in a simple, straightforward manner. RTM can be used to evaluate and compare alternative robot work methods before having to program the robot motions in detail.

E. Experimentation on Joint Torque Sensing - A simple, high gain, wide bandwidth joint torque servo system has been developed to provide a fast response without extensive computation or differential approximations.

F. Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator - A method of "variable" branch-and-bound has been developed which schedules the computation of tasks by distributing the load in a sequential order among the CPU's under the series-parallel prescedence constraints.

G. Resolved Motion Force Control - In Resolved Motion Force Control (RMFC) Cartesian forces are determined instead of joint positions and torques.

III. RESEARCH RESULTS SINCE MARCH 1983 REPORT

A. ROBOT MANIPULATOR CONTROL UNDER UNIX

1. Objective - The objectives of this research is to improve the capabilities of current industrial robots. We propose a new solution to the problem by integrating the robot control into an existing high level language. The robot manipulator is integrated in such a manner that conventional programming techniques can be used to solve the special requirements of manipulator control. We use the 'C' language and run the manipulator under the UNIX operating system. The robot manipulator is integrated into the language in the same manner as is input/output. That is, integration into the language is handled by a small set of functions included in a library. The robot program thus becomes a conventional 'C' program. The implementation language of the library is also written in 'C,' which provides a "user transparent" system, allowing complete freedom in the mode of controlling the manipulator. Concurrency is provided within the operating system. An optimizing computer is available for both the user and as implementation language. There are no special data types as the entire system is represented in terms of standard language features. We have included the manipulator into the 'C' programming language in the form of a library, RCCL the Robot 'C' Control Library (see Appendix 1.)

2. Introduction - RCCL is not a language but a set of system calls suitable for the control of robot manipulators. Manipulator programs become ordinary computer programs, and the manipulator is considered as a peripheral device. Since manipulator control primitives are defined at the system level, a program written in any language which is able to provide the proper list of arguments can use the manipulator primitives.

Instead of designing another robot programming language, we use the 'C' language to write manipulator programs. The RCCL system is itself written in the 'C' language. 'C' is a high level structured language suitable for projects of any size, and which also allows us to deal with low level implementation details. Programs are easily portable, and yet can be efficiently implemented. Two criticisms are often made of compiled language based systems. First, the compilation time increases the edit-test cycle time; secondly, if a program fails, because it is wrong from either the manipulation or the programming point of view, the whole task has to be stopped. Our practice has shown that these limitations are largely offset by the gain in flexibility and generality of a powerful operating system. If for some applications an interpreted language is needed, the interpreter of a general purpose or a dedicated language could also make use of RCCL system calls. The RCCL design approach has advantages in modularity, flexibility, and hardware independence.

1. Overview

1.1. Manipulator Task Description

The location of an object is described by its position and orientation with respect to some reference coordinate frame. In the following the word 'location' will implicitly mean 'position and orientation'. Tasks are described in terms of locations to be reached in space in order to grasp, displace, or exert forces on objects located in the robot work space. Tasks are also described by the sequence and the type of motions necessary to carry out the work. Location descriptions require special data structures, and sequential operations of a robot also require special primitives. Both can, however, be implemented with the tools provided by high level languages namely, data structures, functions, and structured flow of control. (The 'C' language does not know anything about a file, for example. Users wishing to manipulate files in their programs have to include a system file called "stdio.h". This file contains a description of the necessary data structures. Files can be manipulated by system primitive functions like *read, write, filbusf, or, flsbuf* [1]).

1.1.1. Structured Location Description

RCCL handles what is referred to as structured location description [2]. The basic construct is the homogeneous transformation which is a mathematical construct describing the location of coordinate frames. A homogeneous transformation can either be interpreted as the description of the location of a coordinate frame with respect to another, or as a transformation performed on the first coordinate frame. One RCCL system call directly constructs location equations in terms of dynamic data structures. The locations can be modified at the level of the move statement in terms of small translations and rotations described with respect to the *tool* frame. This provides a convenient shorthand for specifying approach and deproach locations, or for specifying motions which purposely overshoot the described location when the manipulator is to perform guarded motions [21].

1.1.2. Motion Description

A task is made up of a number of *path segments* between successive locations. There are many ways to generate trajectories for a manipulator[4][5]. RCCL provides two types of motions. The first, called *joint mode,* consists of computing the set of joint values for each path segment end and generating all intermediate values by linear interpolation. The second type, which we will call *Cartesian mode,* requires the system to solve a modified location equation each sample interval and to compute the corresponding joint coordinates. The location equation is internally modified in such a way that one frame, called the *tool frame*, moves along straight lines and rotates around a fixed axis. These motion types are discussed elsewhere [3][6]. Here, we will assume that we are dealing with a manipulator for which an analytical solution exists, relating a Cartesian location to a set of joints coordinates [7][8][9][10]. In the current implementation, manipulator motions are obtained by specifying a sequence of desired joint values to the servo processes controlling the manipulator joints. However, most of what follows does not assume a particular control method.

When the manipulator is to move while exerting forces or torques on objects, the manipulator must be controlled in a such a way that forces and torques are controlled directly in place of locations. The manipulator is then said to be controlled in a *comply* mode. Several methods [11][12][13][14] are proposed for such control. RCCL implements a variation of Shimano's joint matching method [22]. RCCL provides for compliance specifications in the *tool* coordinate frame which is defined in the location equation. Compliance is specified in terms of forces along, and torques around, the principal axes of the *tool* frame. The manipulator loses one degree of freedom for each direction along or around which it is complying, in forces or torque respectively. The trajectory is then constrained by the geometrical features of the objects in contact. A more complete discussion of this subject can be found in [15].

1.2. Sensor Integration; Update World Representation;

One of the man goals of RCCL is to facilitate the integration of sensors [16]. Sensors are used to modify the behavior of the manipulator according to information acquired from the manipulator or from its environment. Sensor information can be classified in many different ways: according to the data type necessary to represent it, booleans, scalars, vectors, arrays, tensors, etc.; by meaning: touch limit, distance, location, temperature, vibration, force, etc.; by the order of magnitude of the acquisition time, whether minutes, seconds, milliseconds, or microseconds; by accuracy; and so on. Considering this variety, the RCCL approach is deliberately to ignore, when possible, the type of information we may have to deal with, but, on the other hand, to provide means for an efficient utilization of this information.

1.2.1. Modifying Trajectories

Fast sensors can provide for direct synchronous sensory feedback. This corresponds to the class of *functionally* defined transformations. In this case, a transformation is attached to a function that will be evaluated each sample period. the purpose of the function is to calculate the value of the transformation as a function of sensor readings. The location equation in section 2.1.1. makes use of such a functionally defined transform to describe a location with respect to a conveyor belt. If the motion is performed in *Cartesian* mode, the tracking is perfectly accurate, since the location equation is evaluated at sample time intervals. When the motion is performed in *joint* mode, the system estimates the expected location at the end of the segment by linear extrapolation. If the functionally defined transform is computed as a function of time, we can obtain mathematically described motions (circles, ellipses etc...).

The transitions to or from path segments involving moving coordinate frames must deal with unpredictable velocity changes. Smooth transitions are obtained by adding a modifying third order polynomial trajectory during the transition time. The manipulator is stopped by repeating a move to the same location. When the location involves moving coordinate frames the manipulator comes to rest relative to the moving frame. If a stop in absolute coordinates is required, a move to a fixed location must be performed before specifying the stop. The system internally maintains a location equation which always reflects the current location of the manipulator. It is possible to have the manipulator stop at an arbitrary instant at the location it currently occupies. Functionally described transformations can be used anywhere in a location equation. Trajectories can be modified with respect to any coordinate frame which provides unlimited applications.

2. The RCCL Implementation

When a manipulator is under RCCL control, four processes are concurrently running. At the lower level, a *servo process* controls the location or the torque of each manipulator joint. The *setpoint process,* running at interrupt level, computes the Cartesian trajectories and determines the corresponding joint parameters. A real time communication channel swaps information between the *servo process* and the *setpoint process. The user process* running under time sharing is the user program and makes the RCCL system calls. The *setpoint process* communicates with the *user* process via a motion request queue containing all the necessary information.

3. Tools

3.1. Trajectory Planning

There exists a version of the RCCL library which, instead of computing the trajectories in real time, computes them off-line. This is achieved by calling the setpoint function in a loop instead of activating it upon interrupt. The same manipulator programs, provided that they do not depend on external events and information, can be run in this fashion. Some debugging tools are then provided. The system can be asked to keep a trace of the motion requests, to store the sequence of setpoints on file in order to replay them afterwards, or to plot them.

3.2 Teaching

A manual control program is included within RCCL. It consists of a very simple command line language interpreter enabling an operator to move the manipulator interactively in Cartesian coordinates. Motions can be specified in world or tool coordinates. Locations can be recorded via the *update* primitive. The manual control program is implemented entirely in terms of RCCL primitives.

3.3 Transformation Data Base

A simple data base system has also been developed. Transformation values can be recorded and read on-line in manipulator programs. The values can be displayed and modified off-line for maintenance.

4. Conclusion

The main goal of this project was to show that manipulator control could be developed in a more general context than within the framework of a stand-along robot controller with its own language. The current RCCL implementation does not yet offer the convenience of dedicated robot controllers because it requires a large machine. However, as microprocessor based computers become more powerful and can run operating systems like UNIX, the RCCL approach exhibits many advantages over conventional robot controller designs. The conclusion we wish to draw is that robot control can be viewed as an addition to an already existing, tested, and standardized system, rather than the design from scratch of a system which provides only for robot control. The RCCC software has been distributed to approximately 20 research institutions world wide.

B. COLLISION-FREE PATH PLANNING FOR ROBOTS WITH A PRISMATIC JOINT

Industrial robots are computer-controlled mechanical manipulators which perform tasks for industrial applications. One of the essential operations in all the assigned tasks involves the physical motion of the manipulator whose end effector moves from a known initial position and orientation to a specified goal position and orientation. In reality, the workspace of the robot is not free from obstacles such as fixtures, mechanical parts, etc., so that a collision may result if the robot moves freely without any guidance. If, however, the positions and orientations of all the obstacles are known for the entire time interval of operation, it is possible to plan a collision-free path, if one exists, for the robot to move along while performing its task.

The subject of collision-free path planning is relatively new. Within the past five years, only a handful of people have been actively working on this subject. Among them are Pieper [27] and Widdoes [28] who used planes, cylinders, and spheres to represent obstacles (objects). The use of spheres has an advantage of avoiding the orientation problem. However, the free space that is occupied by parts of the spheres is wasted for planning purposes. In addition, the intersection functions are often nonlinear involving square roots or transcendental functions. Udupa [29], Lozano-Perez and Wesley [30], and Lozano-Perez [31,32], and Brooks [33] adopted the polyhedra as the models which result in linear intersection functions. But the orientation problem must be handled with care. Udupa discretized the space into cells which were labelled free if not occupied by obstacles and objects. Lists of free cells are joined together to form a collision-free path. To allow for arbitrary orientation, the obstacles' expansions over-compensate, which reduce the number and/or size of the free cells available for path planning. Lozano-Perez described linked polyhedra using swept volumes. The rotation range is then divided into a finite number of slices. Brooks adopts the idea of generalized cones [34] which are equivalent to swept volumes. Free space is then represented as overlapping generalized cones.

In the methods described above, some determine the free space inside which the point robot may move freely without collisions with obstacles, while other determine the forbidden region so that a collision-free path may be traced along the boundaries of the region. This paper adopts the second approach to the problem which involves objects and obstacles that interact with a robot which has a prismatic link, such as the Stanford manipulator [35]. The prismatic joint, however, creates additional problems. As usual, the objects and obstacles are approximated by enclosing polyhedra. The manipulator is represented by a point; in particular, the point at the tip of the end effector. Its real body width is compensated for by expanding the polyhedral obstacles [29-32]. Methods of constructing the expanded polyhedra are given in these references. If the point robot enters into the expanded polyhedra, a collision will then occur. Now since the prismatic joint of the manipulator has a long boom, it creates two pseudo obstacles: one by the restriction that the front of the boom remain free of collision and the other by any confinement of the rear of the boom due to obstacles. The pseudo obstacle is not a physical object but a region of shadow in the workspace. However, when the point robot enters into the pseudo obstacle, a collision between the boom and a polyhedral obstacle occurs somewhere along its length. Thus the pseudo obstacles together with the expanded polyhedra from the forbidden regions that the point robot must stay away to avoid collisions.

It was shown that for robots with a prismtic joint, such as joint 3 of the Stanford manipulator, the boom's length may be compensated for by two pseudo obstacles for every edge of the objects when the robot is, in the usual sense, represented by a point. One of the pseudo-obstacles is due to the front end of the boom, and the other is due to the rear end. An algorithm has been developed for the computation of the shortest feasible collision-free path for the robot for the case of stationary obstacles. The algorithm converges in at most (N-2)(N-1)/2 iterations where N (see Appendix 2).

C. REAL-TIME 3-D VISION SYSTEM WITH MULTI-CAMERA FOR COLLISION-AVOIDANCE

In the usual robot tasks, practically all involve some manipulation requiring the motion of the end effectors from their initial positions and orientations to the specified goal positions and orientations. However there are fixtures, mechanical parts, etc. in the work-space of the root. Thus collisions between the robot and the obstacles may occur unless some guidance for motion is provided.

A three-dimensional vision system for on-line operation that aids a collision avoidance system for an industrial robot is developed. Because of the real-time requirement, the process that locates and describes the obstacles must be fast. To satisfy the safety requirement, the obstacle model should always contain the physical obstacle entirely. This condition leads to the bounding box description of the obstacle, which is simple for the computer to process.

The image processing is performed by a Machine Intelligence Corporation VS-100 machine vision system. The control and object perception is performed by the developed software on a host Digital Equipment Corporation VAX 11/780 Computer. Also, the communication with the robot collision avoidance program occurs on the VAX 11/780.

The resultant system outputs a file of the locations and bounding descriptions for each object found. When the system is properly calibrated, the bounding descriptions always completely envelop the obstacle. The response time is data-dependent. When using two cameras and processed on UNIX time sharing mode, the average response time will be less than two seconds if eight or fewer objects are present. When using all three cameras, the average response time will be less than four seconds if eight or fewer objects are present. However, the total clapsed time is data-dependent. The program could return in one second if no objects are present (see Appendix 3).

The use of three cameras is preferred since otherwise non-existing objects may be found by the program. However, the perception error of detecting objects that do not exist is more favorable than not to detect objects that do exist, for the purpose of collision avoidance. The bounding description will often waste the space surrounding an object. But, for the same purpose, the inclusion of extra space in the boundary is favorable to not including a part of an object. Also, the user of the output must be aware that the output descriptions may overlap in the three-dimensional space.

Again, the accuracy of the scheme is dependent on the accuracy of the initializations performed and the resolution of the sensor. The user affects the accuracy of the scheme by the accuracy of the lens models used, the orthogonality of the camera set-up, the accuracy of the distance measurements, and the accuracy of the cursor positions chosen during the system initialization. As for any vision system, choosing the correct threshold for each camera and properly adjusting the lighting are also important.

D. DUAL-NUMBER TRANSFORMATION IN DYNAMICS FOR SIMPLIFIED COMPUTATION

The industrial robots have serial link mechanisms whose dynamic behavior can be described by equations in Lagrangian formulation as [36,37]:

$$\tau_i = \sum_{j=1}^{n} D_{ij}\ddot{q}_j + J_{ai}\ddot{q}_i + \sum_{j=1}^{n} D_{ijj}(\dot{q}_j)^2 + \sum_{\substack{j=1 \\ j\neq k}}^{n} \sum_{k=1}^{n} D_{ijk}\dot{q}_j\dot{q}_k + D_i \qquad (1)$$

where

τ_i = input generalized force for joint i for i = 1,2,...,n; and

q_k = generalized coordinate (i.e., joint displacement).

Whether equation (1) is utilized to solve forward dynamics problem for analysis and simulation (i.e., solve for q_j's and their time-derivatives for given τ_i's), or to solve inverse dynamics problem for control of robots (i.e., solve for τ_i's for desired q_j's and their derivatives), one must compute the coefficients D_{ij}, D_{ijk} and D_i. The computation of these terms is, unfortunately, very complicated and time consuming. It involves an evaluation of thousands of trigonometrical terms [38]. Obviously it is not a simple computational task especially when the position-dependent and orientation-dependent parameters change as the robot moves. Therefore it warrants the effort of searching for methods of simplifying the computation.

Efficient algorithms for computing τ_i have been developed by various authors during the past three years. Luh, Walker and Paul [39] computed the joint forces/torques based on the Newton-Euler formulation. Walker and Orin [40] extended the approach to compute the joint accelerations which were then used in the simulation of the robot control scheme. Hollerbach [41] developed recursive algorithms based on the Lagrangian formulation which were shown to be equivalent to the Newton-Euler method [42]. Recently Kane and Levinson [43] used specialized formulation for specific robots, while Featherstone [44] approached the problem differently by using articulated-body inertias.

All the methods mentioned above are very efficient in producing numerical solutions. However, they will yield very little insight views of the dynamical behavior of the robot. To analyze the dynamics of the robot for full understanding and aiding in designing new robots, it is desirable to simplify the computation of the coefficients D_{ij}, D_{ijk} and D_i and then deal with the differential equation (1) directly.

There are three known approaches of simplification, viz. geometric/numeric, composite, and differential transformation. Bejczy's geometric/numeric evaluation [45,46] deals with the nature of joints whether it is revolute or prismatic. Thus the 4 by 4 homogeneous transformation matrices T_j^k in the coefficients can be simplified in advance. Since many elements in the matrices are zeros, the resulting expressions for D_i, D_{ij} and D_{ijk} are less complicated [19,20]. The composite technique by Luh and Lin [47] involves the comparison of all the terms in Newton-Euler formulation of the dynamic equation [39] in a computer. Some of the terms may be eliminated under various criteria. The remaining terms are then rearranged in a Lagrangian formulation. The upshot is a computer output of a simplified equation in symbolic form. Paul's differential transformation [37] which converts $\partial T_o^p/\partial q_j$, the partial derivative of the homogeneous transformation matrices, into the matrix product of the transformation and a differential matrix which reduces D_{ij} to a much simpler form. However, the term D_{ijk} contains a second order partial derivative $\partial^2 T_o^p/(\partial q_j\, \partial q_k)$ which was not simplified until recently by Bejczy and Lee [48]. Their approach is to apply the differential operator used by Paul, successively at the appropriate link-to-link coordinate transformations. An alternative approach is to adopt the dual-number algebra and screw calculus in the analysis instead of the homogeneous transformation.

In screw calculus [49,50], a vector may be represented by either six real numbers, or thee dual numbers. The associated coordinate transformation matrices perform line transformations, which is different from the point transformation by homogeneous transformation. In robotics, this approach has been investigated by Pennock and Yang [51], and Featherstone [44]. As shown by Rooney [52], the dual-number representation is most concise, while the real 6 by 6 matrix representation contains redundant components since not all conditions that form the matrix are independent. The size of the 6 by 6 matrix gives an intuitive impression of excessive computational burden. Yet the dynamical analyses are done by the real 6 by 6 matrix representation in [44] and [51] because it is not feasible to express the inertia directly in dual-numbers.

This paper (see Appendix 4) presents a method of expressing the kinetic energy of the system in terms of dual-number transformations so that the analysis of the dynamics using dual-number algebra is possible. The method is different from the momentum approach by Yang [53]. Because of the property of line transformation, the dual-number transformation may deal with dual-velocity vectors. Thus the differential transformation in the kinetic energy term yields only the first order partial derivatives in D_{ijk} so that Paul's simplification approach [37] applies. Although there is no first order partial derivatives in D_{ij} in the dual-number representation, the computation of D_{ij} is still simpler than that by Paul's simplified representation [37]. The computational efficiency of the dual-number representation is exhibited by comparing the numbers of required multiplications and additions for computing the joint torques/forces τ_i for all n joints, with those numbers required when the direct homogeneous transformation [36], and Paul's simplified homogeneous transformation [37] methods are applied.

Table 1 summarizes the computational complexity of the three methods or comparison. It is seen that the dual-number approach require less computations.

Table 1. Comparison of Computational Complexity

METHOD	NUMBER OF MULTIPLICATIONS*	NUMBER OF ADDITIONS*
Homogeneous Transformation	$28\frac{1}{2}n^4+91n^3+79n^2+19n$	$19\frac{1}{2}n^4+62\frac{1}{6}n^3+54\frac{3}{4}n^2+11\frac{5}{6}n$
Differential Simplification	$28\frac{1}{2}n^4+71n^3+49\frac{1}{2}n^2+7n$	$19\frac{1}{4}n^4+48\frac{1}{6}n^3+33\frac{3}{4}n^2+4\frac{5}{6}n$
Dual-Number	$22\frac{1}{2}n^4+56n^3+39n^2+5\frac{1}{2}n$	$17\frac{3}{4}n^4+43\frac{1}{6}n^3+29\frac{1}{4}n^2+3\frac{5}{6}n$

*Computing all D_{ij} and D_{ijk} for n joints.

E. RTM-ROBOT TIME AND MOTION METHOD

The RTM system is constructed around a list of basic elements that are divided into four major groups: movement elements; sensing elements; gripper or tool elements; process delay elements. Initially, RTM performance models have been developed for the Stanford Arm and for the T^3 robots. Experiments have also been carried out with performance models for Unimate, PUMA, Minimover, and IBM RS/1 robots. A number of work element modeling approaches have been tried, including: look-up tables based on mean performance time values; regression equations based on experimental laboratory data; velocity control models, which depend on the precise method by which the robot is designed to move; path geometry, which presently requires relatively detailed specification and motion parameters.

1. RTM Software - A user can specify a work method for a particular robot by using RTM statements. The statements are of two main types: for robot operations, each containing a standard RTM element and its parameters; and control statements that include general information about the tasks, robot type, output detail and control logic. The logic structure provides capabilities of REPEAT blocks, PARALLEL blocks (for multiple robots or robots and machines), and conditional branching based on simulated conditions of status signals, such as sensory input. A summary of the RTM software statements is shown in Table 1 and in Appendix 5.

Table 2
Summary of RTM system's statements

Statement Type	Statement Structure	
1. Sub-task title	SUBT, (no.), (title), (comment)	
2. REPEAT control card	REP (no. of first operation), TO, (no. of last operation) (no. times to repeat*), (comment)	
3. PARALLEL control card	PAR, no. of first, TO, no. of last, (comment)	
4. Conditional branching	IF, (condition name.condition.value*), GOTO, operation no. or subtask number	
5. Control transfer	GOTO, operation no., subtask no., (comment)	
6. Movement elements (Rn,Mn,ORn)		
a. Position initialization	(joints parameters)	
b. By end-point of segments	(operation no.) (R. T. M. symbol), (comment) (velocity), (joints parameters)	
or: c. By displacement	(operation no.), (R. T. M. symbol), (velocity), (displacement*)	A-Angular D-Linear
7. All other R. T. M. elements	(operation no.), (R. T. M. symbol), (operation parameter), (comment)	
8. END Card	END	
9. CONDITION initialization	COND (condition name), (set of initial values*) END	

*can be generated randomly

2. Performance Prediction Accuracy - Extensive laboratory experimentation and analysis of realistic robot tasks have established that an important advantage of applying the RTM method is in previewing robot work methods before programming them in detail. The prediction accuracy of the system has been found to be only a few percent away from the actual performance time. Specific results for predicting performance time: for the Stanford Arm with the detailed path geometry approach deviations were within -2% to +12%; with the table look-up approach within ±5%; for the T^3 with the velocity control models, within -2% to +3%; with the table look-up approach within ±15%. Analyses have also been performed to study the relationship between accuracy and task element variety and length. For example, the relative inaccuracy of the table look-up approach for the T^3 is evident mainly in multi-segment motions. In less detailed tables several interpolations are required for such motions and consequently, the resulting error increases. On the other hand, an analysis of generic task elements in manufacturing has led to the successful development of simplified RTM models for "point operations" (e.g., spot welding, drilling).

3. Models for Sensory Elements - Modeling work and experiments have been started in order to develop performance models of sensory elements in robot work, mainly with the IBM RS/1 and with a T^3 instrumented with touch and photo sensors on the gripper. The models address several types of sensory work, such as "monitor" elements and "sense input/output" elements. Additionally, several vision systems are studied in order to understand the parameters of their performance time. It was found that in most cases there is a strong dependency on the equipment used. Some of the sensory models require information about dynamic properties of the work environment as well as the expected feedback.

4. Probabilistic RTM Functions - RTM has been expanded to allow sampling from user specified probability distributions of several input parameters.

a. Random elements

All time values specified for Time Delay or Process Time Delay, as well as motion elements, can be supplied with random variable. For example,

```
        M1        10,              D3
```

means, that element M1 is specified with expected motion length of 10 units, distributed according to a given distribution D3.

b. Random Repeat blocks

A block of RTM elements can be repeated a number of times using the REPEAT instruction. The number of repetitions can now be specified as a random variable.

c. Random conditions

Conditional values can be used to evaluate a robot task in a realistic situation, where malfunctions, sensory input, or variable requirements occur. This can be analyzed by the RTM conditional branching. The particular conditions can now be sampled automatically from a random distribution.

4. Interface of RTM to Robot Control Language - One of our research objectives is to interface and integrate the RTM software to the robot control language. As part of the control language, RTM models could be called by an engineer in order to hierarchically evaluate and compare alternative work methods before programming one in detail. Later, it will be possible to use some of the data supplied in the RTM statements directly for the statements of the control language. The environment of the C language provides simple means of delivering this objective.

This objective has been studied relative to the RCCL, the Robot Control C Library. It was found that with the more common, simplified RTM input (without use of world coordinates), effective translation from RTM to RCCL cannot be accomplished. However, with the RTM input using world coordinate specifications of motions, translation is not only feasible, but also quite direct. Several example programs have been studied and translated. Although the resulting RCCL program may be non-optimal at the current stage, little additional data is needed from the user beyond the RTM data for typical robot programs.

IV. REFERENCES

[1] Kernighan, B. K., "The C Programming Language," Prentice-Hall, 1978.

[2] Paul, R. P., "Manipulator Language," Workshop On The Research Needed to Advance The State Of Knowledge In Robotics, April 15-17, 1980, organized by J. Birk and R. Kelley, supported by N.S.F.

[3] Paul, R. P., "Robot Manipulators: Mathematics, Programming, and Control," MIT Press 1981.

[4] Derby, S., "Simulating Motion Elements of General-Purpose Robot Arms," International Journal of Robotic Research, Vol. 2, No. 1, Spring 1983.

[5] Castain, R. H., Paul, R. P., "Polynomial Robotic Trajectories: A New Approach," TR-EE 82-37, Dec. 1982.

[6] Hayward, V., Paul, R. P., "Robot Manipulator Control Using the C Language Under UNIX," IEEE Workshop on Languages for Automation, Chicago, Nov. 183.

[7] Shimano, B. E., "The Kinematic Design and Force Control of Computer Controlled Manipulators," Stanford Artificial Laboratory, Stanford University, AIM 313, 1978.

[8] Paul, R. P., Stevenson, C. N., "Kinematics of Robot Wrists," International Journal of Robotic Research, Vol. 2, No. 1, Spring 1983.

[9] Paul, R. P., Shimano, B. E., Mayer, E. G., "Kinematic Control Equations for Simple Manipulator," IEEE Transactions on Systems, Man, and Cybernetics, Vol SMC-11, No 6, June 1981.

[10] Fisher, W. D., Private communication.

[11] Inoue, H., "Force Feedback in Precise Assembly Tasks," MIT Artificial Intelligence Laboratory, Memo 308, Aug. 1974.

[12] Raiberg, M. H., Craig, J. J., "Hybrid Position/Force Control of Manipulators," Journal of Energy Resources Technology, Vol. 103, June 1981.

[13] Salisbury, J. K., "Active Stiffness Control of a Manipulator in Cartesian Coordinates," 19th IEEE Conference on Decision and Control, Dec. 1980, Albuquerque, New Mexico.

[14] Geschke, C. C., "A System for Programming and Controlling Sensor-Based Robot Manipulators," IEEE Transactions on Pattern Matching and Machine Intelligence, Vol. PAMI-5, No. 1, Jan. 1983.

[15] Mason, M. T., "Compliance and Force Control for Computer Controlled Manipulators," MIT TR-515, April 1979.

[16] Rosen, C. A.,Nitzan, D., " Use of Sensors In Programmable Automation", Computer Magazine, December 1977.

[17] Paul, R. P., "Computational Requirements of Third Generation Manipulators"

[18] Fisher, W. D., "The Modification of a Robotic Manipulator and Digital Controller to Incorporate Both Force and Position Control," MSE Thesis, Purdue University, May 1981.

[19] Luh, J. Y. S., Fisher, W. G., Paul, R. P., "Joint Torque Control by Direct Feedback for Industrial Robots," IEEE Transaction on Automatic Control, Vol. AC-28, No. 2, February 1983.

[20] Zhang, H., Paul, R. P., "Determination of Simplified Dynamics of Puma Manipulator," Purdue University.

[21] Will, P. M., Grossman, D. D., "An Experimental System for Computer Controlled Mechanical Assembly," IEEE Trans. Computers C-24 9, 1975, 879-888.

[22] Shimano, B. E., "The Kinematic Design and Force Control of Computer Controlled Manipulators," Ph.D. Dissertation, Memo AIM-313, 1978, Stanford University.

[23] Ernst, H. A. A., "A Computer Operated Mechanical Hand," Sc. D. Thesis, Masachusetts Institute of Technology, 1961.

[24] Paul, R. P., "WAVE: A Model-Based Language for Manipulator Control," The Industrial Robot **4** 1 (March 1977), 10-17.

[25] Finkel, R., et al. "An Overview of Al, A Programming Language for Automation," Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, 1975, 758-765.

[26] Taylor, R. H., Summers, P. D., Meyer, J. M., "AML: A Manufacturing Language", International Journal of Robotics Research, **1**, 3, Fall 1982, 19-41.

[27] Pieper, D. C., *The Kinematics of Manipulators Under Computer Control,* ARPA Order No. 957, Stanford University, 1968.

[28] Widdoes, C., *A Heuristic Collision Avoider for the Stanford Robot Arm,* C.S. Memo 227, Stanford University, 1974.

[29] Udupa, S. M., *Collision Detection and Avoidance in Computer Controlled Manipulators,* Ph.D. Thesis, California Institute of Technology, 1977.

[30] Lozano-Perez, T. and M. A. Wesley, *An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles,* Communications of the ACM, Vol. 22, No. 10,

[31] Lozano-Perez, T., "Automatic Planning of Manipulator Transfer Movements," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 11, No. 10, October 1981, pp. 681-698.

[32] ----, *Spatial Planning: A Configuration Space Approach,* IEEE Transactions on Computers, Vol. 32, No. 2, February 1983, pp. 108-120.

[32] Brooks, R. A., *Solving the Find-Path Problem by Good Representation of Free Space, Proc. AAAI 2nd Annual National Conference* on Artificial Intelligence, August 18-20, 182, Pittsburgh, Penn., pp. 381-386.

[34] Binford, T. O., "Visual Perception by Computer," Presented at the IEEE Systems Science and Cybernetics Conference, December 1971, Miami, Florida.

[35] Scheinman, V. D., *Design of a Computer Controlled Manipulator,* AI Memo No. 92, Artificial Intelligence Laboratory, Stanford University, June 1969.

[36] Bejczy, A. K., *Robot Arm Dynamics and Control,* Technical Memorandum 33-669, Jet Propulsion Laboratory, February 1974.

[37] Paul, R. P., "Robot Manipulators: Mathematics, Programming, and Control," MIT Press 1981.

[38] Luh, J. Y. S., "Conventional Controller Design for Industrial Robots - A Tutorial," IEEE Transactions on Systems, Man and Cybernetics, Vol. 13, No. 3, May/June 1983, pp. 298-316.

[39] Luh, Y. Y. S., M. W. Walker and R. P. C. Paul, "On-Line Computational Scheme for Mechanical Manipulators," ASME Transactions, Journal of Dynamic Systems, Measurement and Control, Vol. 102, No. 2, June 1980, pp. 69-76.

[40] Waker, M. W. and D. E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," ibid, Vol. 104, No. 3, September 1982, pp. 205-211.

[41] Hollerbach, J. M., "A Recursive Lagrangian Formulation of Manipulator Dynamics and a Cooperative Study of Dynamics Formulation Complexity," IEEE Transactions on Systems, Man and Cybernetics, Vol. 10, No. 11, November 1980, pp. 730-736.

[42] Silver, W. M., "On the Equivalence of Lagrangian and Newton-Euler Dynamics for Manipulators," International Journal of Robotics Research, Vol. 1, No. 2, Summer 1982, pp. 60-70.

[43] Kane, T. R. and D. A. Levinson, "The Use of Kane's Dynamical Equations in Robotics," International Journal of Robotics Research, Vol. 2, No. 3, Fall 1983, pp. 3-21.

[44] Featherstone, R., "The Calculation of Robot Dynamics Using Ariculated-Body Inertias," International Journal of Robotics Research, Vol. 2, No. 1, Spring 1983, pp. 13-30.

[45] Bejczy, A. K. and R. P. Paul, "Simplified Robot Arm Dynamics for Control," Proceedings of 20th IEEE Conference on Decision and Control, December 16-18, 1981, San Diego, California, pp. 261-262.

[46] Bejczy, A. K., "Dynamic Analysis for Robot Arm Control," Proceedings of 1983 American Control Conference, June 22-24, 1983, San Francisco, California, pp. 503-504.

[47] Luh, J. Y. S. and C. S. Lin, "Automatic Generation of Dynamic Equations for Mechanical Manipulators," Proceedings of Joint Automatic Control Conference, June 17-19, 1981, Charlottesville, Virginia, pp. TA-2D.

[48] Bejczy, A. K. and S. Lee, "Robot Arm Dynamic Model Reduction for Control," Proceedings of 22nd IEEE Conference on Decision and Control, December 14-16, 1983, San Antonio, Texas, pp. 1466-1476.

[49] Brand, L., *Vector and Tensor Analysis,* Wiley and Sons, 1948, chapter 2.

[50] Dimentberg, F. M., *The Screw Calculus and Its Applications in Mechanics,* Izdatel'stvo "Nauka", Moskva 1965, English Translation by Foreign Technology Division, WP-AFB Ohio, Part No. 680 993, April 1968.

[51] Pennock, G. R. and A. T. Yang, "Dynamic Analysis of a Multi-Rigid-Body Open-Chain System," ASME Transactions, Journal of Mechanisms, Transmission, and Automation Design, Vol. 105, No. 1, March 1983, pp. 28-34.

[52] Rooney, J., "A Comparison of Representations of General Spatial Screw Displacement," Environment and Planning (England), Series B, Vol. 5, 1978, pp. 45-88.

[53] Yang, A. T., "Inertia Force Analysis of Spatial Mechanisms," ASME Transactions, Journal of Engineering for Industry, Vol. 93, No. 1, February 1971, pp. 27-33.

[54] Yang, A. T., "Calculus of Screws," in *Basic of Design Theory,* Edited by W. R. Spillers, North-Holland Publishing Co./American Elsevier Publishing Co., 1974, pp. 266-281.

V. DOCUMENTATION

[1] R. Paul, J. Luh, et al., "Advanced Industrial Robot Control Systems," First Report, NSF Grant APR77-14533, TR-EE-78-25, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, May 1978.

[2] ----, "Advanced Industrial Robot Control Systems," Second Report, NSF Grant APR77-14533, TR-EE-79-35, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907, July 1979.

[3] -----, 3rd Report, NSF Grant APR77-14533, Covering Period July 1, 1978 to January 1, 1979, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[4] -----, 4th Report, NSF Grant APR77-14533, Covering Period January 1, 1979 to July 1, 1979, TR-EE 80-29, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[5] -----, 5th Report, NSF Grant APR77-14533, Covering Period July 1, 1979 to January 1, 1980, TR-EE 80-30, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[6] -----, 6th Report, NSF Grant APR77-14533, Covering Period January 1, 1980 to July 1, 1980, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[7] ----,"Advanced Industrial Robot Control Systems," 7th Report, NSF Grant DAR 77-14533, Covering Period July 1, 1980 to January 1, 1981, TR-EE 81-8, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[8] ----, "Advanced Industrial Robot Control Systems," 8th Report, NSF Grant DAR 77-14533, Covering Period January 1, 1981 to July 1, 1981, TR-EE 81-16, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907.

[9] R. Paul, "Cartesian Coordinate Control of Robots in Joint Coordinates," presented at the Third CISM-IFTOMM International Symposium on Theory and Practice of Robot and Manipulators," Udine, Italy, September 1978.

[10] R. Paul, "Programming and Teaching of Industrial Robots," presented at the National Electronics Conference, Chicago, October 1978.

[11] R. Paul, "Robot Software and Servoing," Workshop on the Impact on the Academic Community of Required Research Activity for Generalized Robotic Manipulators, University of Florida, February 1978.

[12] J. Y. S. Luh, "Long Range Robotic Research Including Sensor Feedback," 23rd IEEE Machine Tools Conference, Cleveland, Ohio, October 25-27, 1977.

[13] J. Y. S. Luh, M. Walker, "Minimum-Time Along the Path for a Mechanical Arm," Proc. 1977 IEEE Conference on Decision and Control, Vol. 1, New Orleans, LA, December, 1977.

[14] R. P. Paul and S. Y. Nof, "Human and Robot Task Performance," in *Computer Vision and Sensor Based Robots,* G. G. Dodd and R. Lothar (Ed.), Plenum Press, New York, 1979.

[15] T. R. Anderson, R. P. Paul, "High Speed Coordinated Control of Industrial Robots," 9th I.S.J.R. Conference, Washington, D.C., May 1979.

[16] H. Takase, R. P. Paul, E. J. Berg, "A Structured Approach to Robot Programming and Teaching," 79 COMPSAC Conference, Chicago, November 1979.

[17] R. Paul, B. Shimano, "Kinematic Control Equations for Simple Manipulators," IEEE Conference on Decision Making and Control, San Diego, January 1979.

[18] J. Y. S. Luh with C. S. Lin, "Multiprocessor-Controllers for Mechanical Manipulators," Proceedings of COMPSAC 79, 3rd International Computer Software and Applications Conference, 79CH1515-6C, November 6-8, 1979, Chicago, pp. 458-463.

[19] R. L. Paul and S. Y. Nof, "Work Methods Measurement - A Comparison Between Robot and Human Task Performance," *International Journal of Production Research,* Vol. 17, No. 3, 1979, pp. 277-303.

[20] J. Y. S. Luh with M. W. Walker, "Controller for a Mechanical Manipulator," *Automatic Control Theory and Applications (Canada),* Vol. 8, No. 1, January 1980, pp. 24-29.

[21] J. Y. S. Luh with M. W. Walker and R. P. Paul, "Resolved-Acceleration Control of Mechanical Manipulators," *IEEE Transactions on Automatic Control,* Vol. 25, No. 3, June 1980, pp. 468-474.

[22] J. Y. S. Luh with M. W. Walker and R. P. Paul, "On-line Computational Scheme for Mechanical Manipulators," *ASME Transactions: Journal of Dynamic Systems, Measurement and Control,* Vol. 102, No. 2, June 1980, pp. 69-76.

[23] S. Y. Nof, J. L. Knight, and G. Salvendy, "Effective Utilization of Industrial Robots - A Job and Skills Analysis Approach," *AIIE Transactions,* Vol. 12, 1980.

[24] H. Lechtman, S. Y. Nof, "Robot Work Analysis: Task Performance by the Stanford Arm," Research Memorandum, No. 80-4, School of Industrial Engineering, Purdue University, February 1980.

[25] S. Y. Nof and R. P. Paul, "A Method for Advanced Planning of Assembly by Robots," SME AUTOFACT-WEST, California, October 1980.

[26] J. Y. S. Luh with C. S. Lin, "Optimum Path Planning for Mechanical Manipulators," *ASME Transactions: Journal of Dynamic Systems, Measurement and Control,* Vol. 103, No. 2, June 1981, pp. 142-151.

[27] H. Lechtman, "Robot Performance Models Based on R.T.M. Method," M.S. Thesis, School of Industrial Engineering, Purdue University, May 1981.

[28] Nof, S. Y., "Decision Aids for Planning Industrial Robot Operations," *Proc. IIE Conference,* New Orleans, May 1982, pp. 46-55.

[29] Nof, S. Y. and Lechtman, H., "Now It's Time for Rate-Fixing for Robots," *The Industrial Robot,* June 1982, pp. 106-110.

[30] Nof, S. Y. and Fisher, E. L., "Analysis of Robot Work Characteristics," *The Industrial Robot,* September 1982, pp. 166-171.

[31] Fisher, E. L., Nof, S. Y. and Seidmann, A., "Analysis of Robot Systems-Basic Techniques and Advanced Methods," *Proc. of IIE Fall Conference,* Cincinnati, Ohio, Nov. 1982, pp. 385-395.

[32] Lechtman, H. and S. Y. Nof, " Performance Time Models for Robot Point Operations," to appear in the *Int. J. of Production Research.*

[33] Seidmann, A. and Nof, S. Y., "Robotic Manufacturing Cell Design," TIMS-ORSA Conf., April 1982, Detroit, Michigan.

[34] Seidmann, A. and Nof, S. Y., "Manufacturing Cell Design with Random Product Feedback Flow" (forthcoming in IIE Transactions).

[35] Nof, S. Y. and H. Lechtman, "Robot Time and Motion System," *Industrial Engineering*, April 1982, pp. 38-48.

[36] Luh, J. Y. S. and C. S. Lin, "Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 12, No. 2, March/April 1982, pp. 214-234.

[37] Luh, J. Y. S. and C. E. Campbell, "Collision-free Path Planning for Industrial Robots," Proc. 21st IEEE Conference on Decision and Control, December 8-10, 1982, Orlando, Florida, pp. 84-88.

[38] Lin, C. S., P. R. Chang and J. Y. S. Luh, "Formulation and Optimization of Cubic Polynomial Joint Trajectories for Mechanical Manipulators," ibid, pp. 330-335.

[39] Nof, S. Y., Computer Aided Planning of Robotic Assembly, *Proc. of AUTOFACT Europe*, Geneva, Switzerland, September 1983.

[40] Robinson, A. P. and Nof, S. Y., SINDECS-R: A Robotic Work Cell Simulator, *Proc. of 1983 Winter Simulation Conf.*, Dec. 1983, pp. 350-355.

[41] Nof, S. Y., Robot Ergonomics: Optimizing Robot Work, a chapter in the *Handbook of Industrial Robotics*, (S. Y. Nof, Ed.), John Wiley & Sons, 1985.

Appendix 1

RCCL User's Manual Version 1.0

Vincent Hayward

TR-EE 83-46
October 1983

RCCL Users's Manual
Version 1.0

Vincent Hayward

School of Electrical Engineering
Purdue University
West Lafayette, Indiana, 47907

TR-EE 83-46

October 1983

This work was partially supported by a Grant from the CNRS project ARA (Automatique et Robotique Avancée), France, and by the Ransburg Chair of Robotics. This material is also based on work supported by the National Science Foundation under the Grant No. MEA-8119884. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. Facilities to perform this research are provided by the Purdue University CIDMAC project.

Table of Contents

1. Introduction

This manual describes the first version of the RCCL robot programming system. The reader is assumed to be familiar with the C programming language [1], and with the UNIX operating system. A thorough understanding of the control and programming techniques described by Paul in [2] is highly recommended if not mandatory. The design philosophy of RCCL is described in [3].

2. Overview

Using RCCL requires the user to be aware of the hardware and software components. The hardware involves a VAX computer operated under UNIX. A special high speed input-output interface [4] installed on the VAX Unibus extension establishes the communication with a Unimate robot controller [5]. The controller's hardware consists of an LSI1-11 microprocessor and several interfaces mounted on a Qbus (serial, parallel, adc/dac, and host machine interfaces). The LSI-11 microprocessor controls six 6503 joint processors via a special parallel interface. The joint processors control the manipulator's joints via digital and analog circuitry.

Software components can be listed in terms of levels. Starting at the lowest level, we find the *servo* code running in each joint processor. A *superviser* program, loaded in the LSI11 is driven by a hardware clock interrupt. Each time sample, the *superviser* program gathers data from the manipulator state : joint positions and torques, front panel switch register content, analog conversion readings, interrupts the VAX and transmits the data. It then enters a wait state until the VAX sends back low level commands that are transmitted to the joint processors. Interrupts are handled in the VAX by mean of a specialized *device driver.* Each time an interrupt occurs in the VAX, the manipulator state is monitored by a *real time robot interface* that checks for limit conditions. Error conditions are excessive joint rates or motor currents. The manipulator's state data is stored in a C structure available as a global variable [6]. The *real time interface,* after receiving the manipulator's state information, calls a initial user's function, examines the content of a second global C structure describing all the possible command combinations. It checks for validity, translates the requests into low level commands and transmit them to the robot controller. A second user function is then called and can run for the remainder of the sample period. The *real time interface* serves the purpose of a robot controller user's interface and its functions and operation are described in [6].

The *setpoint* process, or trajectory generator is part of RCCL, and uses the *real time interface* to control the manipulator and obtain the manipulator's state. The *setpoint* process is interrupt driven and acts according to asynchronous motion requests specified in the user's program via RCCL primitives.

3. Tutorial Introduction

The first program we shall introduce, uses a reference coordinate frame located at the base of the manipulator whose shoulder is at 864 mm above the base. The transform *T6* describes the position of a frame attached to the last link of the robot originated at the point of meeting of the axes of the last three joints, with respect to the shoulder. We want to move the manipulator at a position located at 600 mm in the X direction, 100 mm in the Y direction, and 800 mm in the Z direction with respect to the reference coordinate frame. We also want that the last link points downward. The program may look like:

```
#include "rccl.h"

pumatask()
{
        TRSF_PTR t, b;
        POS_PTR  p0;

        t = gentr_trsl("T",   0.,    0., 864.);
        b = gentr_rot("B", 600.  , 100., 800., yunit, 180.);

        p0 = makeposition("P0", t, t6, EQ, b, TL, t6);

        move(p0);
        move(park);
}
```

The file **rccl.h** contains C structure type definitions and external entry points the same way the system file "stdio.h" does. It gives access to what users programs may need in order to use RCCL functions, structures, and variables.

The variable declarations include the predeclared types TRSF_PTR, a pointer to a transformation structure, and POS_PTR, a pointer to a position structure. The system builds the transformations matrices needed to describe the task via the **gen_trsl** and **gen_rot** functions. The reference coordinate is called "T" and is set as a pure translation. As for all the RCCL functions that dynamically allocate memory space, the first argument is a string of characters naming the created object. This name is purely arbitrary and can be set to the empty string (""). However, giving meaningful names is a good idea because RCCL uses them in many occasions to print informative messages. The remaining arguments of the *gentr_trsl()* function are the X, Y, and Z values of the *p* (position) vector of the transform. The rotational part is automatically set to the *unit* rotation. The function *gentr_rot()* allocates memory and sets the positional part and the rotational part of the "B" transform. Arguments 1, 2, 3, and 4 have the same meaning as for *gentr_trsl()*. Among several possible ways to specify rotations, we use here a rotation around a vector. The variable 'yunit', which is of the type VECT_PTR, is a pointer to a vector. This variable is provided by RCCL as a pointer to a vector whose value is {0., 1., 0.}. The rotational part of the "B" transform is set to a 180 degrees rotation around the Y unit vector. (The fact that the Z direction of the *T6* transform is pointing in direction of the last link of the manipulator must be kept in mind). The Z axis of the "B" transform is now pointing downward, because "B" is described with respect to "T" whose Z direction points upward.

It is now time to set up a position equation using a call to *makeposition*. *Makeposition* returns a pointer to a ring data structure that is used by the *move* primitive. It accepts a variable number of arguments. The first one is the name of the position. Up to the 'EQ' constant, the list of arguments make

RCCL is systematically coded according to the conventions of the C language Version 6. Recent versions of C allow the passing by value of structures as function arguments. Although one may use these features in the programs, none of the RCCL functions make use of them and structure arguments are always passed by address.

up the left hand side of the position equation. Then comes the list of transforms making up the right hand side. The constant 'TL' introduces the transform that we choose to be the *tool* transform. The *tool* transform can be any of the frames contained in the equation, provided that it gives meaningful results, more on that later. For now, we can say that most of the time, $T6$ or one of the frames described with respect to $T6$ in the left hand side of the equation will be chosen. We obtain the following equation :

$$T\ T6 = B$$

The first *move* request causes the manipulator to move such that the position equation is satisfied. In practice the robot will not exactly reach "P0", but will perform a transition close to it before going back to "PARK". The 'park' position pointer is build into the system.

Before proceeding further, we shall add two modifications to this first example. We replace:

```
move(p0);
move(park);
```

by:

```
setmod('c');
move(p0);
stop(0);
move(park);
```

By default, RCCL tasks start in *joint* mode. By calling *setmod()* we ask for the moves to be performed in *Cartesian* mode, the *tool* frame, here $T6$, move along a line joining the "PARK" position and the "P0" position. The *stop* statement causes the manipulator to stop during a null time at "P0", that is to say, to bring the velocity to zero. In other words, it will actually reach the position "P0". The $T6$ transform, during the travel to "P0", will be evaluated at sample time intervals as:

$$T6 = T^{-1}\ B\ DRIVE$$

The purpose of the *DRIVE* transform is to produce a straight path motion [2]. Most of the time, the position equation will include one or several transforms to describe the end effector. This can be achieved by creating one more transform and adding one argument to the position equation.

```
e = gentr_trsl("E", 0., 0., 170.);

p0 = makeposition("P0", t, t6, e, EQ, b, TL, e);
```

Now the location described by the transformation "E" with respect to $T6$ will travel along a straight Cartesian path and $T6$ will be evaluated as :

$$T6 = T^{-1}\ B\ DRIVE\ E^{-1}$$

4. Basic Components : Numbers, Vectors, Transformations, Differential motions, Forces, and Events.

We shall now describe in more detail the meaning and form of a first set of RCCL primitives and how they can be used in manipulator programs.

REMARKS

All RCCL functions returning a structure, follow the convention that the result is the left argument (output argument) and that first argument is returned as the value of the function (in the same style as **strcat** does). This allows to code in the following style :

```
trans = rot(newtrans("TRANS", const), zunit, 90.);
```

which in one line, allocates a transform and sets it to a pure rotation around the Z direction. Because the type of each function is declared in the file **rccl.h** , the program lint will complain if the returned value is not used. Each function of this style is associated with a macro that capitalizes the first letter. In case of 'rot', the macro is :

```
#define Rot      (void) rot
```

such that the same above code can be written as:

```
trans = newtrans("TRANS", const);
Rot(trans, zunit, 90.);
```

without complains from lint.

4.1. Numbers

The **rccl.h** include file contains structured definitions of vectors and transformations that should be used in connection with the corresponding functions. These structure declarations are preceded with C 'typedef' definitions that better describe the implementations of basic data types :

```
typedef int bool;

typedef float real;
```

C knows two floating point variable types : double and float. They correspond on most machines to single and double precision floating point representation and arithmetic. For efficiency, all calculations are performed in single precision. In order to insure consistency throughout the RCCL code, the type 'real' has been declared as a C typedef. Every single floating point variable is declared as such. Because C structures are always passed by address, and because 'double' and 'float' variables have different sizes, the proper address calculations are insured. However, automatic type conversions will give meaningful results if type 'double' variables are assigned to or from RCCL variables.

A set of math constant global variables is included in the library :

```
real pi_m       is   PI
real pib2_m     is   PI / 2
real pit2_m     is   PI * 2
real dgtord_m   is   PI / 180
real rdtodg_m   is   180 / PI
```

The purpose of those variables is to avoid a unnecessary increase of the size of process data region (see end(2)) and they are initialized at compile time. Setting them to any other values guarantees unpredictable results.

4.2. Vectors

The type 'vector' is described by the following structure :

```
typedef struct vector {
                    real x, y, z;
} VECT, *VECT_PTR;
```

The C 'typedef' feature is a way of giving another name to basic data types.

A C structure variable 'k' implementing a vector can either be coded as:

```
        struct vector k;
```

or

```
        VECT k;
```

A pointer to a vector variable can either be coded as :

```
        struct vector *pk;
```

or

```
        VECT *pk;
```

or

```
        VECT_PTR pk;
```

The choice is according to taste and coding habits. Using this structure gives access to the following functions: **dot, assignvect, cross,** and **unit.** In order to describe the argument types of these functions and the type of the value that they return, their heading declarations are displayed :

```
        real dot(u, v)
        VECT_PTR u, v;

        VECT_PTR assignvect(v, u)
        VECT_PTR v, u;

        VECT_PTR cross(r, u, v)
        VECT_PTR r, u, v;

        VECT_PTR unit(v, u)
        VECT_PTR v, u;
```

The function **dot** returns the dot product of two vectors. The function **assignvect** copies its second argument into the first one. Likewise, the function **cross** returns in its left argument the cross product of the two remaining arguments. The function **unit** computes a vector collinear with its right argument vector but of unit magnitude. Taking the cross product of two identical vectors is meaningless. By contrast, the **unit** function can perfectly take two identical arguments. In that case, the magnitude of the vector would be set to unity 'in place'.

4.3. Transformations

The corresponding C structure is :

```
typedef int(* TRFN)();

typedef struct transform {
                    char *name;
                    TRFN fn;
                    VECT n, o, a, p;
                    int timeval;
} TRSF, *TRSF_PTR;
```

The first entry in the structure is a pointer to a string that stands for the transform name. The second, is a pointer to a function. The function pointer can be set to one of the user's background real-time function or to one of the system functions **const, varb,** or **hold.** A more complete discussion of this point occurs later. For now, we can assume that this pointer will most of the time point to the **const** function, meaning that the transform is constant transformation and will not change throughout the execution of the task. The next entry contains the value of the transform itself built in terms of four vectors: the **normal, orientation, approach,** and **position** vectors. The last row the transform is assumed to be : {0 0 0 1}. In other words, the transforms can only be orthogonal transforms. The last entry is the time of the last evaluation of the function, needed in the case of functionally described transforms.

This type declaration gives access to the following functions :

```
TRSF_PTR assigntr(t1, t2)
TRSF_PTR t1, t2;

TRSF_PTR taketrsl(t1, t2)
TRSF_PTR t1, t2;

TRSF_PTR takerot(t1, t2)
TRSF_PTR t1, t2;

TRSF_PTR trmult(r, t1, t2)
TRSF_PTR r, t1, t2;

TRSF_PTR trmultinp(r, t)
TRSF_PTR r, t;

TRSF_PTR trmultinv(r, t)
TRSF_PTR r, t;

TRSF_PTR invert(r, t)
TRSF_PTR r, t;

TRSF_PTR invertinp(t)
TRSF_PTR t;
```

The **assigntr** function is quite similar to the **assignvect** function above and the same remarks can be made. It must, however, be noticed that only the value part of the transform is copied and not the other components of the structure. The functions **taketrsl** and **takerot** perform a selective copy of the translational (resp. rotational) part, and leaves untouched the rotational (resp. translational) part. The function **trmult** multiplies the two right arguments transforms and leaves the result in the left argument. This function requires the three arguments to be different. The function **trmultinp** multiplies the two arguments and leaves the result in the left argument. The function **trmultinv** multiplies the left argument by the inverse of the right one and leave the result in the left argument. The function **invert** leaves in the left argument the inverse the right one. Since the arguments must be different, the function **invertinp** performs an inversion 'in place'.

The following functions selectively set the terms of the transformations :

```
TRSF_PTR trsl(t, px, py, pz)
TRSF_PTR t;
real px, py, pz;

TRSF_PTR vao(t, ax, ay, az, ox, oy, oz)
TRSF_PTR t;
real ax, ay, az, ox, oy, oz;

TRSF_PTR rot(t, k, h)
TRSF_PTR t;
VECT_PTR k;
real h;

TRSF_PTR eul(t, phi, the, psi)
TRSF_PTR t;
real phi, the, psi;

TRSF_PTR rpy(t, phi, the, psi)
TRSF_PTR t;
real phi, the, psi;
```

All these functions use a transformation pointer as left argument, which as usual is returned as a value of the function. The function **trsl** sets the terms of the *p* vector of the transformation and leaves the rotational part untouched. All distances in RCCL are expressed in millimeters. The function **vao** sets the vectors *n*, *o*, and *a* of the transformation. Since the vectors *n*, *o* and *a* are orthogonal, **vao** only needs the terms of *o* and *a* and builds the vector *n*. The vectors whose components are passed as arguments do not need to be orthogonal. The rotational part of the transform is built as follows: take the user's supplied *a* vector, normalize it and use it as the final *a* vector, take the user's supplied *o* vector (which may not be orthogonal) and build a possibly non unit vector *n* but orthogonal with *o* and *a*, reconstruct *o* as to be orthogonal with *n* and *a*, normalize it, and finally derive *n* from *o* and *a*. The function **rot** sets the rotational part of the transformation as a rotation around a vector possibly unnormalized, second argument, of a given angle, third argument, expressed in degrees. The function **eul** sets the rotational part of the transformation as a rotation expressed with Euler angles in degrees. Finally, the function **rpy** sets the rotational part of the transformation as a rotation expressed with roll, pitch, and yaw angles in degrees. These rotation setting functions leaves the translational part of the transform untouched.

The next set of functions are similar in form to the previous ones, except that the transform, left argument, is multiplied by a translation or a rotation (which is quite a different thing). As usual, the left argument transformation pointer is returned as value of the function.

```
        TRSF_PTR trslm(t, px, py, pz)
        TRSF_PTR t;
        real px, py, pz;

        TRSF_PTR vaom(t, ax, ay, az, ox, oy, oz)
        TRSF_PTR t;
        real ax, ay, az, ox, oy, oz;

        TRSF_PTR rotm(t, k, h)
        TRSF_PTR t;
        VECT_PTR k;
        real h;

        TRSF_PTR eulm(t, phi, the, psi)
        TRSF_PTR t;
        real phi, the, psi;

        TRSF_PTR rpym(t, phi, the, psi)
        TRSF_PTR t;
        real phi, the, psi;
```

As stated at the beginning of this section, when the value of the function is unwanted, a set a macros is provided. They produce the following list of names :

Assignvect	Cross	Unit		
Assigntr	Taketrsl	Takerot		
Trmult	Trmulinp	Trmultinv		
Invert	Invertinp			
Trsl	Vao	Rot	Eul	Rpy
Trslm	Vaom	Rotm	Eulm	Rpym

As we are able to specify the rotational part of transforms with Euler or roll, pitch, yaw angles, we may need to derive them from a given transformation. These representations are not unique for a given rotation. The functions are :

```
        noatoeul(phi, the, psi, t)
        real *phi, *the, *psi;
        TRSF_PTR t;

        noatorpy(phi, the, psi, t)
        real *phi, *the, *psi;
        TRSF_PTR t;
```

Please note that the three first arguments are **pointers** to the three results of the pseudo type 'real'.

We now need to use transformation as easily as we would use simple data types in C. At the beginning of manipulation functions, one needs to declare transformations and to allocate memory for them. This can be done in the following manner :

```
pumatask()
{
        TRSF base;

        ...
        ...
}
```

This way of allocating memory for transformations presents three major drawbacks. The first one is that dynamic variables, allocated in the stack, only live the duration of the function call. Since the execution of manipulator programs is not explicitly synchronized with the calculation of trajectories, the function may well exit before the requested motions are completed. All the memory space allocated in the stack would be allocated for other purposes. This will surely cause a lot a trouble because the values of the transformations are used for the trajectory calculations. One may go around this by writing :

```
        static TRSF base;
```

but the space would remain permanently allocated. The second trouble is that the value of the transforms and other entries in the structure need to be initialized. If one chooses to use dynamic stack allocations, one also need to synchronize the function such as it does not exit before the transforms are no longer in use :

```
pumatask()
{
        TRSF base;

        base.name = "NAME";                  /* set the name */
        base.fn = const;                     /* tell it's constant */
        Assigntr(&base, unitr);              /* init to unit transform */
        Trsl(&base, 0., 0., 200.);           /* set it to a translation */
        base.timeval = 0;                    /* reset time eval */

        ...

        waitfor(completed)                   /* make sure not any more in use */
}
```

The third drawback is that we will most of the time refer to transforms by pointers, and it would lead to a heavy use the the '&' operator. The initialization statement for a static 'TRSF' variable would not be any more convenient and would be very error prone:

```
        static TRSF base = {"BASE",
                            const,
                            1.,0.,0.,
                            0.,1.,0.,
                            0.,0.,1.,
                            0.,0.,200.
                            0,
                           };
```

Although the techniques described above are perfectly viable, RCCL provides a built-in dynamic memory allocation system for transforms (and positions). The basic call is the function **newtrans**:

```
        TRSF_PTR newtrans(n, fn)
        char *n, TRFN fn;
```

This function returns a pointer to a transform initialized to the unit transform. The second argument is a pointer to a function, either one of the user's functions which ,as we will see, have to possess certain properties, or one of the predefined functions **const, varb, hold.** Since **newtrans** dynamically allocate memory in user space, the creation of too many transforms will cause a program exit with the message "mem. alloc error". The statement :

```
        freetrans(t);
```

permits the system to free the allocated memory when needed (it is implemented as a macro).

Other RCCL functions make use of **newtrans** as a short hand for common coding patterns:

```
TRSF_PTR gentr_trsl(name, px, py, pz)
char *name;
real px, py, pz;

TRSF_PTR gentr_rot(name, px, py, pz, k, h)
char *name;
real px, py, pz, h;
VECT_PTR k;

TRSF_PTR gentr_pao(name, px, py, pz, ax, ay, az, ox, oy, oz)
char *name;
real px, py, pz, ax, ay, az, ox, oy, oz;

TRSF_PTR gentr_eul(name, px, py, pz, phi, the, psi)
char *name;
real px, py, pz, phi, the, psi;

TRSF_PTR gentr_rpy(name, px, py, pz, phi, the, psi)
char *name;
real px, py, pz, phi, the, psi;
```

These functions permit us to create transformations and initialize them all at once. They all return a pointer to the created transforms by default set as **const** transforms. The first four arguments are : the name (string), and the components of the *p* vector. For creating transforms containing non unit rotations, the expression of the rotational part is analogous to the previous family of functions. For example :

```
TRSF_PTR t1, t2, t3;     /* declare transform pointers */

...

t1 = trsl(eul(newtrans("T1", const), 10., 20., 30.), 1., 2., 3.);

t2 = gentr_eul("T2", 1., 2., 3., 10., 20., 30.);

t3 = gentr_trsl("T3", 1., 2., 3.);
Eul(t3, 10., 20., 30);
```

give three identical transforms.

The last group of transformation related functions are for output :

```
printr(t, fp)
TRSF_PTR t;
FILE *fp;

printe(e, fp)
TRSF_PTR e;
FILE *fp;

printy(e, fp)
TRSF_PTR e;
FILE *fp;

printrn(t, fp)
TRSF_PTR t;
FILE *fp;
```

The function **printr** prints the numerical value of the transform, first argument. The functions **printe** and **printy** respectively print the Euler and pith, roll, yaw angles. The function **printrn** prints the name, the numerical value and the angles altogether. All these functions take as a second argument a UNIX file pointer. As an example, the output of the following sequence of calls :

```
TRSF_PTR t1, t2, t3;

t1 = gentr_eul("T1", 10., 20., 30., 11., 12., 13.);
printf("part 1\n");
printe(t1, stdout);
printrn(t1, stdout);

t2 = newtrans("T2", const);
printf("part 2\n");
printr(t2, stdout);
Rot(t2, yunit, 90.);
TrsIm(t2, 10., 20., 30.);
printrn(t2, stdout);

t3 = newtrans("T3", const);
printf("part 3\n");
printrn(trmult(t3, t1, t2), stdout);

printf("part 4\n");
printrn(trmult(t3, t2, t1), stdout);
```

would be :

```
part 1
EUL x:10.000  y:20.000  z:30.000  phi:11.000  the:12.000  psi:13.000
T1 :
   0.893   -0.402    0.204   10.000
   0.403    0.914    0.040   20.000
  -0.203    0.047    0.978   30.000
EUL x:10.000  y:20.000  z:30.000  phi:11.000  the:12.000  psi:13.000
RPY x:10.000  y:20.000  z:30.000  phi:24.280  the:11.688  psi:2.738
part 2
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000    0.000
T2 :
   0.000    0.000    1.000   30.000
   0.000    1.000    0.000   20.000
  -1.000    0.000    0.000  -10.000
EUL x:30.000  y:20.000  z:-10.000 phi:0.000   the:90.000  psi:0.000
RPY x:30.000  y:20.000  z:-10.000 phi:0.000   the:90.000  psi:0.000
part 3
T3 :
  -0.204   -0.402    0.893   26.700
  -0.040    0.914    0.403   49.973
  -0.978    0.047   -0.203   15.076
EUL x:26.700  y:49.973  z:15.076  phi:24.280  the:101.688 psi:2.738
RPY x:26.700  y:49.973  z:15.076  phi:-169.000 the:78.000  psi:167.000
part 4
T3 :
  -0.203    0.047    0.978   60.000
   0.403    0.914    0.040   40.000
  -0.893    0.402   -0.204  -20.000
EUL x:60.000  y:40.000  z:-20.000 phi:2.323   the:101.776 psi:24.240
RPY x:60.000  y:40.000  z:-20.000 phi:116.707 the:63.207  psi:116.922
```

This should also suffice to remind us that the matrix product (and also orthogonal transforms products) is not commutative.

4.4. Differential Motions and Forces.

Although RCCL do not explicitly use the structured representation of differential motions or generalized forces in manipulation primitive calls, they are made available to the user. A differential motion is expressed in terms of a differential translation vector and differential rotation vector. A generalized force is expressed in terms of a linear force vector and a moment vector. The corresponding structures are :

```
typedef struct diff {
                VECT t, r;
} DIFF, *DIFF_PTR;

typedef struct force {
                VECT f, m;
} FORCE, *FORCE_PTR;
```

The associated functions are :

```
        DIFF_PTR assigndiff(t, o)
        register DIFF_PTR t, o;

        TRSF_PTR df_to_tr(t, d)
        register TRSF_PTR t;
        register DIFF_PTR d;

        DIFF_PTR tr_to_df(d, t)
        register DIFF_PTR d;
        register TRSF_PTR t;

        DIFF_PTR difftr(dt, d, tr)
        register DIFF_PTR dt, d;
        register TRSF_PTR tr;

        printd(d, fp)
        DIFF_PTR d;
        FILE *fp;

        FORCE_PTR assignforce(t, o)
        register FORCE_PTR t, o;

        FORCE_PTR forcetr(ft, f, tr)
        register FORCE_PTR ft, f;
        register TRSF_PTR tr;

        printm(d, fp)
        FORCE_PTR d;
        FILE *fp;
```

The function **assigndiff** performs a copy of a differential motion structure. The function **df_to_tr** builds a transformation out of a differential motion. The function **tr_to_df** builds a differential motion structure, given a transformation. The function **difftr** transforms a differential motion expressed with respect to one frame into the same differential motion expressed with respect to another frame. For example if $P1$ if a frame expressed in base coordinates and $P2$ its transformation by T such as :

$$P2 = T\ P1$$

A differential motion expressed with respect to P1, is obtained expressed with respect to $P2$. The left argument of **difftr** is the output argument : the transformed differential motion, the second argument is the original differential motion and the third argument is the transform expressing the differential relationship. The **prind** function prints on one line a differential motion.

The functions **assignforce, forcetr, printm** perform analogous processing of generalized forces and torques. Note that if the forces are expressed in Newtons, torques must be expressed in Newton-millimeters since distances are in millimeters, the conversions are straightforward. As for other functions of that kind the following name can be used instead, if the returned pointer is not used :

```
Assigndiff    Difftr        Df_to_tr      Tr_to_df
Assignforce   Forcetr
```

For example, the following sequence of program statements :

```
        DIFF  Dp1, Dp2;
        FORCE Fp1, Fp2;
        TRSF_PTR t = gentr_pao("T", 10., 5., 0., 1., 0., 0., 0., 0., 1.);

        printrn(t, stdout);

        Dp2.t.x = 1.;
        Dp2.t.y = 0.;
        Dp2.t.z = .5;
        Dp2.r.x = 0.;
        Dp2.r.y = .1;
        Dp2.r.z = 0.;
        printd(&Dp2, stdout);
        printd(difftr(&Dp1, &Dp2, t), stdout);

        Fp2.f.x = 10.;
        Fp2.f.y = 0.;
        Fp2.f.z = 0.;
        Fp2.m.x = 0.;
        Fp2.m.y = 100.;
        Fp2.m.z = 0.;
        printm(&Fp2, stdout);
        printm(forcetr(&Fp1, &Fp2, t), stdout);
```

will produce the following output :

```
T :
   0.000     0.000     1.000    10.000
   1.000     0.000     0.000     5.000
   0.000     1.000     0.000     0.000
EUL x:10.000  y:5.000   z:0.000   phi:0.000    the:90.000  psi:90.000
RPY x:10.000  y:5.000   z:0.000   phi:90.000   the:0.000   psi:90.000
tx  1.0e+00  ty  0.0e+00  tz  5.0e-01  rx  0.0e+00 ry  1.0e-01 rz  0.0e+00
tx  0.0e+00  ty -5.0e-01  tz  1.0e+00  rx  1.0e-01 ry  0.0e+00 rz  0.0e+00
fx  1.0e+01  fy  0.0e+00  fz  0.0e+00  mx  0.0e+00 my  1.0e+02 mz  0.0e+00
fx  0.0e+00  fy  0.0e+00  fz  1.0e+01  mx  1.0e+02 my  5.0e+01 mz  0.0e+00
```

4.5. Events

RCCL uses the notion of *event* to synchronize the user's program with the manipulator motions. *Motion requests* are entered into a queue at a given moment, and executed on the basis of the first in, first out, when all the previous request are served. The first snare one can run into is depicted by the following :

```
        for (i = 0; i < 10000; ++i) {
                move(p1);
                move(p2);
        }
```

The almost 'infinite' loop being asynchronously executed, the queue will be become saturated in a few milliseconds. In this situation, we have chosen to cause an error condition since it will most of the time be the result of a program flaw. In many occasions an *event* will be needed to explicitly synchronize the program with the arm motions, say for opening and closing a gripper.

An *event*, is defined in RCCL as an integer :

```
typedef int event;
```

An event is essentially a count, if positive, it represents the number of processes waiting for the occurrence. Occurrence of an event decreases the count by one, when the count drops to zero, no process are waiting for it. RCCL maintains the built in event **completed** that occurs when the motion queue becomes empty. The user's program may use the primitive **waitfor** implemented as a macro, to synchronize with events, for example :

```
        move(p1);
        move(p2);
        move(p3);
        waitfor(completed)
        printf("the arm has reached 'p3', proceeding...\n");
```

or else, using the *event* called 'end' associated with each position :

```
        move(p1);
        move(p2);
        move(p1);
        move(p2);
        OPEN;
        waitfor(p1->end)
        CLOSE;
        waitfor(p2->end)
        OPEN;
        waitfor(p1->end)
        CLOSE;
        waitfor(p2->end)
        OPEN;
```

to realize synchronization of gripper actions.

5. Task Description

Describing a task consist of specifying positions to be reached in space and motions to these positions. RCCL implements structured positions descriptions, and asynchronous motion requests.

5.1. Position Equations

Position equations do not necessitate the use of absolute reference coordinates. Position equations are one representation of the more general concept of transformation graphs. The position relationships of the frames $F_{i\ i=1,n}$ can be expressed in terms of transformations products. Let a transformation T_i describe the position of the frame F_{i+1} relative to the frame F_i with T_n describing the transformation from frame F_n to F_1, we have :

$$T_1\ T_2\ \cdots\ T_n = \mathit{Idendity}$$

A closed path of transformations from frame F_1 to frame F_1, via the frames $F_{i\ i=2,n}$ describes the position of F_1 with respect to itself : the identity transform. The situation is depicted by a directed closed graph :

$$\begin{tikzcd} F1 \arrow[r, "T1"] & F2 \arrow[r, "T2"] & F3 \arrow[r, phantom, "\cdots\cdots"] & Fn \arrow[lll, bend left, dashed, "Tn"] \end{tikzcd}$$

where the vertices are frames and the arcs transforms.

Given a set of frames, containing two frames A and B, we can certainly find more than one path connecting A to B. Let the frames on one path be called $F_{i\ i=1,n}$, and the the frames on the other path be called $G_{i\ i=1,m}$, we obtain :

$$\begin{tikzcd} & F1 \arrow[r, "T1"] & F2 \arrow[r, "T2"] & F3 \arrow[r, phantom, "\cdots\cdots"] & Fn \arrow[rd, "Tn"] & \\ A \arrow[ru, "T0"] \arrow[rd, "R0"'] & & & & & B \\ & G1 \arrow[r, "R1"'] & G2 \arrow[r, "R2"'] & G3 \arrow[r, phantom, "\cdots\cdots"] & Gn \arrow[ru, "Rm"'] & \end{tikzcd}$$

The corresponding transformation equation is :

$$T_0\ T_1\ T_2\ \cdots\ T_n = R_0\ R_2\ \cdots\ R_m$$

Closed transformation graphs can be expressed in terms of a set of transformation equations. Transformation graphs can be generalized, but we will restrict them to the form above.

RCCL uses transformations equations in order to describe the positions the manipulator has to reach. We will first introduce the dedicated transform $T6$. We are dealing with manipulators having six links and six joints, labeled from 1 to 6. The base of the manipulator is labeled link 0. Each of the manipulator links is assigned a frame A_i describing its position with respect to the previous one as a function of the joint variable. The position of link 1 is described with respect to the base. The transformation product :

$$T6 = A_1\ \cdots\ A_6$$

describes the position of the last link with respect to the base. Note that for the manipulators we are dealing with, it is convenient to assign the last three frames at the intersection of the three last joint axes. Therefore, $T6$ does not take into account the end effector description.

By convention the following transform decompositions are given :

$$T1 = A1$$

$$T2 = A1\,A2$$

$$T3 = A1\,A2\,A3$$

$$T4 = A1\,A2\,A3\,A4$$

$$T5 = A1\,A2\,A3\,A4\,A5$$

$$T6 = A1\,A2\,A3\,A4\,A5\,A6$$

$$U6 = A6$$

$$U5 = A5\,A6$$

$$U4 = A4\,A5\,A6$$

$$U3 = A3\,A4\,A5\,A6$$

$$U2 = A2\,A3\,A4\,A5\,A6$$

$$U1 = A1\,A2\,A3\,A4\,A5\,A6$$

$$T6 = T5\;U6 = T4\;U5 = T3\;U4 = T2\;U3 = T1\;U2$$

Let us set up a position equation that structurely describes the situation when the manipulator is to grasp an object lying on a table. We first need to assign frames to each of the elements involved :

- A frame is assigned to the shoulder of the manipulator : S.
- A frame is located at the last link of the manipulator : M.
- A tool is attached to the link 6, the frame T is assigned to the working end of tool.
- A frame W describes the position of the working table.
- The position of an object lying on the table is described by O.
- A grasp position is described by the frame G.

Suppose that the manipulator is moving such as to grasp the object, the corresponding graph is :

$$\begin{tikzcd}
& M \arrow[r, dash, dashed] & T \arrow[dr, dash] & \\
S \arrow[ur, dash] \arrow[dr, dash] & & & G \\
& W \arrow[r, dash, dashed] & O \arrow[ur, dash] &
\end{tikzcd}$$

In order to turn this graph into a transform equation, we first need to orient the arcs and label them with transforms. The choice is arbitrary but a convenient possibility is :

$$\begin{tikzcd}
& M \arrow[r, dashed, "TOOL"] & T & \\
S \arrow[ur, dashed, "T6"] & & & G \arrow[ul, dashed, "DRIVE"'] \\
& W \arrow[ul, dashed, "BASE"] \arrow[r, dashed, "OBJ"'] & O \arrow[ur, dashed, "GRASP"'] &
\end{tikzcd}$$

where *TOOL*, *BASE*, *GRASP*, *OBJ* are predetermined transforms. The transform $T6$ is to be changed such that the transform *DRIVE* comes to identity for the manipulator to reach the desired position, or in other words, such as the frames T and G become identical. The way the *DRIVE* transform changes (and therefore $T6$) as a function of time determines the way the frame M and T move with respect to S, W, O, or G (Note that no absolute coordinate system is involved and we could say that S, W, O, and

G move with respect to M and T).

The equivalent equation of the position can be written :

$$BASE\ T6\ TOOL = OBJ\ GRASP\ DRIVE$$

The equation of the final desired position can be written :

$$BASE\ T6\ TOOL = OBJ\ GRASP$$

Transformations equations can be rewritten, solved for any of the terms, or replaced by equivalent ones. For example, we have :

$$BASE\ T6 = OBJ\ GRASP\ TOOL^{-1}$$

$$T6 = BASE^{-1}\ OBJ\ GRASP\ TOOL^{-1}$$

$$T6 = COORD\ POS\ TOOL^{-1} \quad \text{if } COORD = BASE^{-1}, POS = OBJ\ GRASP$$

Etc....

The RCCL function **makeposition** permits the user to set up such a position equation. The *set-point* process will automatically compute the terms of the *DRIVE* transform such that the resulting motion possess certain properties. The **makeposition** function expects a variable number of arguments. They represent the left hand side of the equation, the right hand side, and a transform that will tell which frame is to be considered as the **tool** frame, the frame T in the example above. Assume that the transforms, *BASE*, *TOOL*, *OBJ*, and *GRASP* have been created via RCCL calls, and that we have the respective transforms pointer available : *base*, *tool*, *obj*, and *grasp*. The 'C' definition of the function **makeposition** is :

```
POS_PTR makeposition(n, lhs [, lhs] ..., EQ, rhs, [, rhs] ..., TL, tl)
char *n;
TRSF_PTR lhs ..., rhs ..., tl;
```

and the call corresponding to the above example is :

```
POS_PTR p;          /* a position pointer */

p = makeposition("P", base, t6, tool, EQ, obj, grasp, TL, tool);
```

The names 'EQ' and 'TL' are predefined constants. The function **makeposition** returns a pointer to a ring data structure implementing the transform graph. The first argument is a string, the name of the position. The transform pointer **t6**, is built in RCCL, and predeclared in **rccl.h.** As the position data structure is built, **makeposition** calls the function **optimize** in order to premultiply all possible pairs of constant transformation (declared as **const**), in order to decrease the run time computing load. The function **optimize** will internally replace the user specified position equation by an equivalent canonical form :

$$T6 = COORD\ POS\ TOOL$$

The terms *COORD* or *TOOL* of this canonical form can be missing. The calls :

```
makeposition("P1", t6, EQ, h, TL, t6);
makeposition("P2", t6, t, EQ, h, TL, t);
makeposition("P3", t6, t, EQ, h, g, TL, t6);
```

lead to the following canonical equations :

$T6 = POS \quad COORD = None\ ,\ POS = H\ ,\ TOOL = None$

$T6 = POS\ TOOL \quad COORD = None,\ POS = H\ ,\ TOOL = T^{-1}$

$T6 = COORD\ POS \quad COORD = H\ G\ ,\ POS = T^{-1}\ ,\ TOOL = None$

There is an arbitrary number of argument transform pointers for **makeposition.** The only restriction is that the left hand side of the equation must contain the predeclared pointer **t6** and the right hand side must contain at least one transform. The transforms can arbitrary belong to one of the following categories :

const : A transform of this type will be considered as constant through out the life of the corresponding position equation. Its value must not be changed, as the system can decided to premultiply it with another transform such as it may not appear in the internal equation used for the trajectory calculation. This is the default type of the functions of the style gentr_... and it is the type that one should use when possible.

varb : A transform of this type will not be premultiplied by the optimization function and its value will be used directly during the trajectory calculation. One sometimes need to change the value of a transform after the equation has been set up. If the change occurs while the equation is evaluated, the change will instantaneously be reflected in the manipulator's trajectory. This can cause jerky motions if the change is large and it should be carefully used. The function **update** described later on, knows when to change the value of the transform when it is safe.

hold : A transform of this type is not directly used in the position equations, but a copy of it. We will see that **move** requests are asynchronously issued and that a number of them can ve specified ahead of time. A **hold** transform belongs to the subsequent motion request and its value is taken into account only when the corresponding motion is actually performed.

The last category is the class of the functionally defined transforms. These transforms are attached to a function belonging to the user's manipulator program. The function is expected to compute the values of the transform as the corresponding motion is performed. The function is executed at interrupt level and therefore, is expected to have a reasonable execution time. As described in [6], these functions **cannot** perform **any** type of system calls, (prints, reads, etc...). If the function computes the values of the transform as a function of external data, one can obtain tracking. If the values are computed as a function of time, one obtains functionally defined trajectories. We shall call such a function a background function.

The type of a transform is indicated in the 'fn' field of the transform structure, a few examples :

```
int myfunction();

t1 = gentr_trsl("T1", 0., 0., 0.);

t2 = newtrans("T2", const);

t3 = newtrans("T3", varb);

t4 = gentr_trsl("T4", 0., 0., 0., );
t4->fn = hold;

t5 = gentr_rot("T5", 0., 0., 0., zunit, 90.);
t5->fn = myfunction;

t6 = newtrans("T6", myfunction);
Rot(t6, zunit, 90.);
```

t1 is a regular **const** transform initialized to the unit value.

t2 is a regular **const** transform initialized to the unit value.

t3 is a transform initialized as a unit transform of type **varb.**

t4 is first created as **const** but is turned into a **hold** transform.

t5 is first created as **const** but is turned into a functionally defined transform attached to the function 'myfunction' and is initialized as a rotation of 90 degrees around the Z axis.

t6 is created as functionally defined, and then initialized as a rotation of 90 degrees around the Z axis.

The **makeposition** function implements a restricted case of transformations graphs. This limitation may be removed one day. When multibranch transform graphs are required, the user must implement it in terms of the basic graph described above and a combination of other RCCL function like **Rot, Trslm, Trmult, Invert,** and so on. The **varb** and **hold** features are then very important.

It is now time to introduce the position structure as described in the file **rccl.h.**

```
typedef struct posit {
                char *name;
                int code;
                real scal;
                event end;
} POS, *POS_PTR;
```

The first entry in the structure is the name of the position. The same remarks can be made as for the transforms structures. The name is not directly relevant from the robot control point of view, but may help in debugging. The second entry 'code' is a termination code for the corresponding motion. Internal code values known by RCCL are currently :

```
#define OK      -1
#define LIMIT   -2
#define ONF     -3
#define OND     -4
```

After the position has been reached, the code is set by the system to the value 'OK' if the motion has not be interrupted by some condition. The value 'LIMIT' means that the motion caused some joints to dangerously approach their physical stops. RCCL automatically issue a stop to the current position. It is then possible to recover from this error condition as explained later. The code 'ONF' means that a prespecified force condition occurred, and the code 'OND' that prespecified differential motion condition occurred. The next entry is a floating point number 'scal'. The value of the field 'scal' varies from 0 to 1 as the motion is performed, and is useful for generating functionally defined motions or to trigger some action at a given point of a trajectory. The entry 'end' is classified as an **event.** It allows the user to synchronize the program flow with the execution of trajectories. The use and the function of the fields 'code', 'scal', and 'end' will be explained in more detail as we go on.

When a position is no longer needed, memory space can be retuned to the memory pool by :

```
freepos(p)
POS_PTR p;
```

Care must be taken so that the corresponding data in no longer in use. Transforms involved in the corresponding equation are not freed, an must be individually freed using **freetrans.**

5.2. Motion Description

RCCL implements two basic types of motions known as *joint* mode and *Cartesian* mode. The first one consist of solving for *T6* the position equation of the goal position at the beginning of the motion and obtaining for it the corresponding set of joint values. The trajectory is then generated by linear interpolation of the joint values from the current position to the goal position. This type of motion should be used for large motions as it requires the minimum joint motions and less computations. High velocities can be obtained, however trajectories are not always easily predictable. The *Cartesian mode* makes use the *DRIVE* transform to produce straight line trajectories for the *tool* frame. The transform equation is evaluated for *T6* each sample time interval and the set of joint values obtained. This type of motion permits us to obtain well predictable trajectories. If the position equation contains functionally defined transforms, the associated functions are also evaluated at sample time intervals. The values resulting from these evaluations will directly influence the arm trajectory. In that case the structure of the position equation must be carefully considered.

5.2.1. The Basic Move Statement

The basic function definition is :

```
move(p)
POS_PTR p;
```

The call :

```
move(pos);
```

where 'pos' is a pointer to a position equation returned by **makeposition.** instructs the system to move the arm toward the described position such as the equation becomes verified. When the arm is moving from position to position, transitions occur between each path segment. It is important to smooth out the velocity discontinuities that would be caused by an abrupt change of direction and velocity from one path segment to another. There are a number of options and parameters that can be set globally or for each path segment.

5.2.2. Setting Options and Parameters

The first group of parameters remains set until set to another value. The following calls cause a setting of the parameter starting at the next **move** request :

```
setvel(tv, rv)
int tv, rv;

setmod(m)
int m;

setconf(c)
char *c;

sample(s)
int s;
```

The **setvel** call takes two integer arguments. The first is the desired translational velocity of the *tool* frame in millimeters per second, the second one is the rotational velocity in degrees per second. The system will calculate the path segment durations to obtain the desired velocities. Since rotational and translational velocities lead to different durations, the system will pick which ever is the longest. One can give the priority to one or another by specifying very different values. For example, suppose that a motion involves a 30 millimeters translation and a 30 degrees rotation, the call

```
setvel(30, 300);
```

will result in a 1 second motion due to the translation, and not an unreasonable 1/10 of a second motion to perform the rotation. The function **setmod** defines the mode, *Cartesian* or *joint*, for the subsequent motions. The argument must be the character 'c' for *Cartesian* mode, and 'j' for *joint* mode. For example :

```
POS_PTR p, p1, p2;
int i, m;

p1 = makeposition(...);
p2 = makeposition(...);

for (move(p2), i = 0; i < 10; ++i) {
        if (i % 2 != 0) {
                m = 'c';
                p = p1;

        } else {
                m = 'j';
                p = p2;
        }
        setmod(m);
        move(p);
}
```

will cause the arm to move from p2 to p1 (i odd) in *Cartesian* mode and from p1 to p2 (i even) in *Joint* mode. For C experts a more concise version could be :

```
for (move(p2), i = 10; i--;) {
        setmod((m = i % 2) ? 'c' : 'j');
        move(m ? p1 : p2);
}
```

In *joint* mode the segment durations are computed based on the distances between the frame $T6$ at each end of the segments, since this type of motion is joint oriented. The function **setconf** permits us to obtain an arm configuration change during the subsequent motion. This motion **has** to be performed in *joint* mode, since a configuration change always involves a degenerate arm configuration unreachable in *Cartesian* mode. Once the configuration change is obtained, the motions can again be performed in *Cartesian* mode. For the PUMA arm, the configurations can be : shoulder righty - lefty (r/l); elbow down - up (d/u); wrist flip - noflip (f/n). The argument is a string specifying the configuration change. For example, if the arm is lefty, up, noflip ("lun"), in order to obtain a wrist configuration change to flip, the arm remaining lefty and up ("luf"), we code :

```
/* the arm is currently "lun" */

setmod('j');      /* go in joint mode if it was'nt */
setconf("f");     /* specify flip */
move(new);        /* go "luf" */
```

Note that several letters can be specified in the string argument. A program with many configuration changes is safely terminated by :

```
setconf("lun");
move(park);
```

The function **sample** allows us to change the sampling period of the whole system. Currently valid sample rates are: 14, 28, 56, and 112 milliseconds. However the function rounds down the specified value as : sample(15) leads to 14, sample(30) leads to 28, etc. The default value is 28 milliseconds which is a good compromise for most applications. The 14 millisecond rate is helpful for tracking applications, but it is good practice to reset the rate to 28 or 56 when not needed.

The next group of functions cause the parameter to be taken into account for the subsequent *move* request only.

```
setime(ta, ts)
int ta, ts;

evalfn(fn)
int (* fn)();

distance(s, v[, v] ...)
char *s, real v;
```

A call to **setime** allows us to force the motion to be performed within a given period of time. The first argument specifies the duration of the transition time at the end of the segment, and the second argument the duration of the segment itself. Times are specifies in milliseconds. Besides the cases when motion duration is the primary factor, this call serves two purposes. At the present time no call has been implemented to force a rate at the joint level. The consequence is that the system is unable to compute the correct segment duration to perform a configuration change, since the same position can sometimes be reached in different configurations. A duration calculation based on distances is in this case meaningless. Therefore, the user must explicitly specify the motion duration. For this reason a macro has been included in **rccl.h**:

```
#define moveconf(p, ta, ts, cf)  \
                  {setconf(cf); setmod('j'); setime(ta, ts); move(p);}
```

The example above can be more conveniently coded as :

```
moveconf(new, 100, 700, "f");
```

The configuration change will be performed in 7/10 of a seconds with a 1/10 of a second transition time. The function **setime** is also very useful for functionally defined motions. When a position equation includes functionally defined transforms, there are situations when the system cannot compute the correct segment duration based on the distance of the goal position because it can depend on arbitrary factors. Likewise a macro has been added :

```
#define movecart(p, ta, ts)     {setmod('c'); setime(ta, ts); move(p);}
```

The code would be :

```
        movecart(spiral, 300, 2000);
```

perform a spiraling motion during 2 seconds with a 3/10 of a second escape transition. In the cases when the segment duration calculation is left to the system but the acceleration time still needs to be explicitly specified, the call :

```
        setime(200, 0);
```

forces the acceleration time to be 2/10 of a seconds but the segment duration, being left unspecified, will be computed by the system. On the other hand, it is sometimes necessary to specify a zero acceleration time, meaning that no transition is desired. This is useful for some slow motions terminated on condition and when the reaction time is of primary importance. The acceleration time can be specified as zero :

```
        setime(0, 1000);
```

The function **evalfn** cause the function argument to be evaluated during the execution of the corresponding motion. One thus can code any needed synchronous processing. The first application is to perform some monitoring of external condition in order to interrupt a motion. For example, a flag called **nextmove** , causes at any moment the current path segment to terminate and the manipulator to transition to the next. Other applications can be to trigger some action at a precise point of a trajectory. For this the field 'scal' of a position structure can be used. The user's function given as argument is executed at sample time and therefore bears the same restrictions as the background functions of functionally defined transforms : short processing, no read, prints and so on. This type of function will also be called a background function.

The function **distance** specifies a distance modifier for the goal positions. Modifications are expressed in the *tool* frame. The first argument is a string specifying the directions. Each direction is expressed with two letters. The first letter can be either 'd' or 'r', standing for 'distance' and 'rotation'. The second letter can be either 'x', 'y', or 'z', standing for the principal axes of the tool frame. A valid directions specification is :

```
"dx rz" :  translation along X, rotation around X
```

The remaining arguments are the magnitudes of the modifications. For example :

```
        distance("dz", -30.);
        move(p);
        move(p);
```

implements a 'approach' style motion in the 'Z' direction of the tool. Modifications are obtained by internally multiplying the *POS* part of the canonical transformation equation by a modification transform. Any combination of directions can be specified, however magnitudes should remain small for rotations. The function distance is also very usefull for motions terminated on condition to purposely specify an overshooting motion.

When a stop is needed the call :

```
        stop(n);
```

where 'n' is a duration in milliseconds repeats the last move statement with all it's attributes, except the time attributes. For example :

```
evalfn(myfunction);
distance("dx ry", 10., 3.);
move(p);
evalfn(myfunction);
distance("dx ry", 10., 3.);
setime(200, 2000);
move(p);
```

is more conveniently written as :

```
evalfn(myfunction);
distance("dx ry", 10., 3.);
move(p);
stop(2000);
```

5.3. Synchronization

A more delicate programming of the time aspect is the price paid for the gain in flexibility obtained by the motion queue feature. Synchronization is basically achieved by 'suspending' the execution of the user's program while motion requests are performed. This is can be done in two ways or by a combinations of both. The program execution is suspended when spending time performing computations and input output. Suppose a program that interacts with a user or with some long response sensor, we obtain the following pattern :

```
TRSF_PTR z, e , b;
POS_PTR  p;
double iz;

z = gentr_trsl("Z",  0.,  0., 864.);
e = gentr_trsl("E" , 0. , 0. , 170.);
b = gentr_rot("B", 600. , 128., 800., yunit, 180.);
b->fn = hold;

p = makeposition("P" , z, t6, e, EQ, b, TL, e);

for (; ; ) {
        printf("enter Z increment ");
        scanf("%f", &iz);
        b->p.z += iz;
        move(p);
}
```

Each time the user enters data via 'scanf', the value of the *B* transform is changed, since its type is *hold* the new value is entered in the motion queue as well as the motion request itself and the next loop immediately prompts the user for a new data entry. If the user enters data quicker than the manipulator can move to the goal positions, she or he will be able to enter several requests ahead. If the user stops entering data, the requests will eventually be served, the manipulator brought to rest and the program execution suspended at the 'scanf' instruction. If the data is provided by some external device, say another computer, a file ,or a sensing device the program will look like :

```
for (; ; ) {
        gettr(b, device);
        move(p);
}
```

where 'gettr' returns a new transform value. The data is obtained asynchronously with respect to the motions, consequently two situations can occur. Either the external device is faster and the queue will fill up, either the arm is faster and it will wait for new data. In both cases we obtain an optimal utilization of resources. The only problem is to prevent the queue from becoming saturated. The external variable **requestnb** is maintained by RCCL as the number of non served motion requests. We can now introduce the primitive **waitas** (a macro) that takes as argument a predicate :

```
waitas(pred)
```

The macro **waitas** suspends the programs execution until the predicate becomes true. The final version of the loop is :

```
for (; ; ) {
        gettr(b, device);
        waitas(nbrequest < MAX)
        move(p);
}
```

The primitive **waitfor** (a macro) suspends program execution until occurrence of an *event*. We have seen that the 'end' **event** is associated with each position record. One application is to obtain a gripper opening and closing at given moments. The pattern of code :

```
distance("dz", -30.);
move(p);
move(p);
distance("dz", -30.);
move(p);
```

implements a possible position 'approach, reach, and depart' sequence. To obtain a synchronized gripper opening and closing, the pattern is :

```
distance("dz", -30.);
move(p);
move(p);
distance("dz", -30.);
move(p);

waitfor(p->end)
OPEN;
waitfor(p->end)
CLOSE;
```

The program is first suspended until, the 'approach' position is reached, opens the gripper, waits for the position to be reached, and closes the gripper. One other application of **waitfor** is to obtain a suspension of the program until all the requests are served. For example suppose that a function allocates positions and transforms that have to be freed of upon termination of the function, we must make sure that all the requests are executed before doing so :

```
dothat()
{
        TRSF_PTR t1 = gentr_ ...
        TRSF_PTR t2 = gentr_ ...

        POS_PTR p1 = makeposition( ...
        POS_PTR p2 = makeposition( ...

        move(p1);
        move(p2);
        ...

        move(there);
        waitfor(completed);
        freetrans(t1);
        freetrans(t2);
        ...
        freepos(p1);
        freepos(p2);
        ...
        return;
}
```

The following program makes use of the **there** position that always reflects the position that the manipulator is occupying at the end of the previous path segment. Thus, when the 'waitfor' statement is issued, we are sure that all the previous requests are served and the corresponding positions are no longer in use. Note that the statement :

```
        waitas(requestnb == 0)
```

would do equally well.

Another way to obtain synchronous actions is to trigger them from a motion associated background function. For example a gripper opening can be specified at a given point of a trajectory. We will make use of the external variable **goalpos**, which is an ordinary position pointer updated by the system such as to point at the position equation currently being evaluated. It can be used in the main program to decide which position equation is currently being evaluated. The background functions can also use the pointer **goalpos** to access the fields of a position structure, and the use of several global position pointers can be avoided. These function can then be written independently of a given motion statement.

```
pumatask()
{
        int openfn();
        ...


        evalfn(openfn);
        move(p);

        ...
}

openfn()
{
        if (goalpos->scal > .5) {
                OPEN;
        }
}
```

or using an event :

```
event openit = 0;          /* global variable */

pumatask()
{
        int openfn();
        ...

        evalfn(openfn);
        move(p);
        waitfor(openit)
        OPEN;
        ...
}

openfn()
{
        if (goalpos->scal > .5 && openit > 0) {
                --openit;
        }
}
```

Yet another possibility would be :

```
pumatask()
{
        ...

        move(p);
        waitas(goalpos == p && p->scal > .5)
        OPEN;
        ...
}
```

According to the situation, different combinations of these techniques can be used. Libraries of customized functions or macros could be written to best suit the requirements.

The 'wait' style primitives have the property of suspending the program execution until occurrence of an event or condition. One must be aware that the following code pattern :

```
        move(p0);
        waitfor(p0->end);
        move(p1);
        waitfor(p1->end);
```

or

```
        move(p0);
        waitfor(completed);
        move(p1);
        waitfor(completed);
```

causes each position to be evaluated twice, since a new request is entered into the queue only when the previous is completed. At that time, the system finds the queue empty and reissues a move to the last position.

In more detail we show the body of the macros **waitas** and **waitfor**

:

```
#define waitas(predicate)     {while(!(predicate)) suspendfg();}

#define waitfor(event)        {++(event); while(event > 0) suspendfg();}
```

The function **suspendfg** merely suspends the foreground program or user's process during a short period of time (currently .1 second). The 'setpoint' process, running in background at high priority maintains the *events* associated with positions and the *event* **completed.** The code in the *setpoint* process looks like :

```
        newm = dequeue(&mqueue);          /* try to get a new request     */
        if (newm == NULL) {               /* then none                    */
                --completed;              /* signal queue is empty        */
        }
```

Consider the following situation :

```
        move(p0);
        move(p1);

        /* do a lot of computations and/or io */
        ...

        waitfor(p0->end);
```

If the sequence of code between the move requests and the wait statement takes more time to execute than the motion to be performed, the task will not hang at the level of the wait statement.

One additonal point has to be considered, suppose we have the following situation :

```
move(p0);
distance(...);
move(p0);
distance(...);
move(p0);
move(p0);

waitfor(p0->end)
/* do something */
waitfor(p0->end)
/* do another thing */
```

In this sequence of code the 'p0->end' *event* will occur four times, but will be waited for only two times. If the sequence of code is in a loop, an unmatching number of moves and corresponding waits will shift the synchronization each loop by the difference. One way to get around that is to reset the event count prior to issuing the *move* requests :

```
for (...) {
        p0->end = 0;
        move(p0);
        distance(...);
        move(p0);
        distance(...);
        move(p0);
        move(p0);

        waitfor(p0->end)
        /* do something */
        waitfor(p0->end)
        /* do another thing */
}
```

5.4. Functionally Defined Motions

One of the principal features of RCCL is the provision for functionally defined motions. They are approached in very general terms. Except the *POS* transform of the canonical equation, any of the transforms of a position equation can be functionally defined. We will look in this section at transforms as functions of time. Let us examine again the typical transformation graph of a *Cartesian* motion :

$$\begin{tikzcd}
{} \arrow[r, "T6"] \arrow[d, dash] & M \arrow[r, "TOOL"] & T & {} \arrow[l, "DRIVE"'] \arrow[d, dash] \\
S \arrow[d, dash] & & & G \arrow[d, dash] \\
{} & W \arrow[l, "BASE"'] \arrow[r, "OBJ"'] & O \arrow[r, "GRASP"'] & {}
\end{tikzcd}$$

We notice that the transforms *T6* and *DRIVE* are themselves function of time. The system internally computes the values of the *DRIVE* transform such that the frame immediately on its left, *T*, moves along straight lines and rotates around fixed axes with respect to *S*, *W*, *O*, or *G*. One might notice that the combination *T TOOL* can be considered as a single transform function of time such that :

$$T6\ TOOL\ DRIVE^{-1} = CONSTANT$$

The *DRIVE* transform can be broken down into a translational part and a rotational part :

$$DRIVE = D_t \, D_r$$

The D_t transform determines the path of the *center of rotation,* while D_r determines the rotation itself, which is a motion that one can easily visualize. The decomposition

$$DRIVE = D_r \, D$$

is also possible but the second transformation cannot be a pure translation in the general case. It is also more difficult to visualize, because any change in the rotation part will also cause a change in the final translational position. Actually, this situation occurs for the transformation product $T6 \; TOOL$, which behaves symmetricly with respect to *DRIVE*. This effect can be observed when a manipulator equipped with a tool performs a pure rotation around the tip of the tool : the manipulator must perform translations and rotations whereas pure translations only require translations. This must be kept in mind when introducing functionally defined transforms in the position equations. It is important to carefully determine the placement of the center of rotation when laying out a functionally defined position equation.

For simplification we shall assume that the goal position has been reached so that the *DRIVE* transform is reduced to unity. We shall also only keep two frames, the world frame W, and the tool frame T. Let $T(t)$ and $R(t)$ two transforms, pure translation and pure rotation function of time. The four graphs leading to pure translations and pure rotations in *world* or base coordinates and then in *tool* coordinates are :

Pure translation in **world** coordinates :

$$\begin{tikzcd}
\cdot \arrow[r, "T6"] & \cdot \arrow[r, "TOOL"] & \cdot \\
W \arrow[u, dash] \arrow[d, dash] & & T \arrow[u, dash] \arrow[d, dash] \\
\cdot & \cdot \arrow[l, "BASE"'] \arrow[r, "T(t)"'] & \cdot
\end{tikzcd}$$

Pure translation in **tool** coordinates :

$$\begin{tikzcd}
\cdot \arrow[r, "T6"] & \cdot \arrow[r, "TOOL"] & \cdot \\
W \arrow[u, dash] \arrow[d, dash] & & T \arrow[u, dash] \arrow[d, dash] \\
\cdot & \cdot \arrow[l, "BASE"'] & \cdot \arrow[l, "T(t)"']
\end{tikzcd}$$

As mentioned before, pure translations lead to pure translations, no matter what frame we are working in. The difference between these two cases is that if $T(t)$ changes along a principal direction, the frame T will also change along a principal direction in *world* coordinates in the first case, and in *tool* coordinates in the second case.

Pure rotation in *world* coordinates.

$$\begin{tikzcd}
\cdot \arrow[r, "T6"] & \cdot \arrow[r, "TOOL"] & \cdot \\
W \arrow[u, dash] \arrow[d, dash] & & T \arrow[u, dash] \arrow[d, dash] \\
\cdot & \cdot \arrow[l, "BASE"'] \arrow[r, "R(t)"'] & \cdot
\end{tikzcd}$$

Pure rotation in tool coordinates :

$$\begin{tikzcd}
{} \arrow[r, dashed, "T6"] \arrow[d, dash, "W"'] & {} \arrow[r, dashed, "TOOL"] & {} \arrow[d, dash, "T"] \\
{} & {} \arrow[l, dashed, "BASE"] & {} \arrow[l, dashed, "R(t)"]
\end{tikzcd}$$

The first case will cause the center of rotation to be fixed with respect to W (move with respect to T). The second case will cause the center of rotation to move with respect to W (be fixed with respect to W). We leave to reader the writing of the corresponding equations. Armed with this conceptual tool we can introduce actual examples.

1) The first example defines two locations that differ by position and orientation. The two positions are described with respect to a moving frame in *world* coordinates. A loop causes a motion back and forth from one position to the other. The final motion translates along the Y axis.

```
 1  #include "rccl.h"
 2
 3  pumatask()
 4  {
 5          TRSF_PTR z, e , b, pa1, pa2, conv;
 6          POS_PTR  p0, pt1, pt2;
 7          int convfn();
 8          int i;
 9
10          conv = newtrans("CONV",convfn);
11          z = gentr_trsl("Z",  0.,  0., 864.);
12          e = gentr_trsl("E" , 0. , 0.  , 170.);
13          b = gentr_rot("B", 600.  , -500., 600., yunit, 180.);
14          pa1 = gentr_eul("PA1"   , 30., 0., 50., 0., 20., 0.);
15          pa2 = gentr_eul("PA2"   , -30., 0., 50., 0., -20., 0.);
16
17          p0 = makeposition("P0"  , z, t6, e, EQ, b, TL, e);
18          pt1 = makeposition("PT1", z, t6, e, EQ, conv, b, pa1, TL, e);
19          pt2 = makeposition("PT2", z, t6, e, EQ, conv, b, pa2, TL, e);
20
21          setvel(300, 50);
22          setmod('c');
23          setime(300, 0);
24          move(p0);
25          for (i = 0; i < 4; ++i) {
26                  movecart(pt1, 100, 1000);
27                  movecart(pt2, 100, 1000);
28          }
29          setmod('j');
30          move(park);
31  }
32
33  convfn(t)
34  TRSF_PTR t;
35  {
36          t->p.y += 3.;
37  }
```

Line 1 includes the necessary RCCL declarations. Line 3 deserves a comment : when using the puma manipulator, the RCCL library calls the function 'pumatask' as the task to be executed. Before calling the 'pumatask' function, the system perform some initializations. When the function returns, as you might expect, the system performs a 'waitfor(completed)' before concluding and exiting. Line 5 and 6, allocates transform and position pointers as needed by the task. Line 7 declares the name 'convfn' as a pointer to a function that describes the moving coordinate frame, and line 8 allocates a counter variable. Line 10, allocates a functionally defined transform attached to 'convfn'. Lines 11 through 15, allocate and initialize transforms as described earlier. The Z transform sets a frame at the base of the manipulator. The E and B transforms are the tool transform and a location with respect to the simulated conveyor. Note that the B transform contains a 180 degree rotation around the Y axis such as the Z direction of frame described by B points downward (relatively to $CONV$ and Z). The transforms $PA1$ and $PA2$ define two locations with respect to the frame described by B.

Lines 17, 18, and 19 set up the position equations as described earlier.

Line 21 sets the velocity to 300 millimeters per seconds and 50 degrees per second and the motion mode is set to *Cartesian* mode on line 22. The call to **setime** on line 23, containing a null segment time, and specifies a 3/10 of a second acceleration time when reaching P0 to allow for a sufficiently long transition time because the next motion occurs with respect to a moving frame (the system has no means to now how fast it is going to move). The 'for' loop, lines 25 to 28, causes eight move requests to be entered in the queue. The eight motions are performed in 1 second each with a 1/10 of a second transition time as specified by the macro **movecart.** Line 29 sets the mode to *joint* because the arm is to perform a large motion and the path the *tool* frame is going to follow is of no concern. Line 30 is the last motion request to the 'park' position.

The function 'convfn', lines 33 to 37, starts being evaluated when the first motion to "PT1" begins and during the seven subsequent motions. The background function attached to the transform is called by the system with one argument pointer, a pointer to the transform it is attached to. This permits us to write functions independently from the actual transform they are attached to. Since **newtrans** created the transform *CONV* as a unit transform, the value of the p_y element of the position vector increases from 0 to approximatively 286 millimeters(it is increased by 3 millimeters each 28 milliseconds for 8 seconds). At the time the manipulator reaches P0, the *CONV* transform is equal to the unity transform. The first time the manipulator moves to PT1, the motion is the result of a combination of the *Cartesian* motion from P0 toward PT1 and the motion due to the moving coordinate frame.

This example introduce the first method for generating functionally defined motion by a periodic increment of a static variable (here a transform element).

2) In this second example we will suppose that the manipulator is mounted on a revolving base or that the manipulator is working with respect to a circular conveyor whose rotation axis is collinear with the first joint. We have introduced minor differences in order to point out some other aspects.

```
 1  #include "rccl.h"
 2
 3  static int t0;
 4
 5  pumatask()
 6  {
 7          TRSF_PTR z, e , b, pa1, pa2, pivot;
 8          POS_PTR  p0, pt1, pt2;
 9          int pivotfn();
10          int i;
11
12          pivot = newtrans("PIVOT", pivotfn);
13          z = gentr_trsl("Z",  0.,  0., 864.);
14          e = gentr_trsl("E" , 0. , 0. , 170.);
15          b = gentr_rot("B1", 600. , -300., 700., yunit, 180.);
16          pa1 = gentr_eul("PA1"  , 30., 0., 50., -10., 10., 0.);
17          pa2 = gentr_eul("PA2"  , -30., 0., 50., 10., -10., 0.);
18
19          p0 = makeposition("P0" , z, t6, e, EQ, b, TL, e);
20          pt1 = makeposition("PT1", pivot, z, t6, e, EQ, b, pa1, TL, e);
21          pt2 = makeposition("PT2", pivot, z, t6, e, EQ, b, pa2, TL, e);
22
23          setvel(300, 20);
24          setmod('c');
25          setime(200, 0);
26          move(p0);
27          waitfor(completed);
28          t0 = rtime;
29          for (i = 0; i < 6; ++i) {
30                  move(pt1);
31                  move(pt2);
32          }
33          setvel(400, 100);
34          setmod('j');
35          move(park);
36  }
37
38  pivotfn(t)
39  TRSF_PTR t;
40  {
41          Rot(t, zunit, (t0 - rtime) * .010);
42  }
```

The lines 1 to 26 are basically the same ones and do not deserve further comments.

In this example, the moving coordinate frame will explicitly be written as a function of time. It makes use of the external variable **rtime** updated by the system each sample interval. The variable **rtime** reflects the time elapsed since the beginning of the task in milliseconds. Although this variable may be reset by the user to any value, we have chosen to record in 't0' the time when the functionally defined motion begins. Although the position of the moving coordinate frame is periodic, it is necessary to set the beginning of the motion at a given instant in order to keep the resulting task execution within the work range of the manipulator. The macro **waitfor** suspends execution until all the preceding motions are executed and the initial time is recorded at line 28. In actual implementations, the use of an *event* would permit us to synchronize the task execution with arbitrary motions of, say, the conveyor. The segment times in the 'for' loop, lines 29 to 32, are left unspecified and will be computed as to obtain an angular velocity of 20 degrees per second (line 23).

It is important to notice the placement of the functionally defined transform in position equations so as to produce the desired effect.

The functionally defined frame is merely described as a negative rotation around the Z axis of 10 degrees per second.

3) In this third example the positions $PA1$ and $PA2$ are now described with respect to rotating table off the axis of the manipulator's first joint. This will cause the end effector to rotate such as to maintain a constant orientation with respect to the table.

```
#include "rccl.h"

static int t0;

pumatask()
{
        TRSF_PTR z, e , b, pa1, pa2, table;
        POS_PTR  p0, pt1, pt2;
        int tablefn();
        int i;

        table = newtrans("TABLE", tablefn);
        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 170.);
        b = gentr_rot("B", 600. ,  300., 700., yunit, 180.);
        pa1 = gentr_rpy("PA1"  , 0., 0., 0., 0., 0., 10.);
        pa2 = gentr_rpy("PA2"  , 0., 0., 0., 0., 0., -10.);

        p0 = makeposition("P0" , z, t6, e, EQ, b, TL, e);
        pt1 = makeposition("PT1", z, t6, e, EQ, b, table, pa1, TL, e);
        pt2 = makeposition("PT2", z, t6, e, EQ, b, table, pa2, TL, e);

        setvel(300, 20);
        setmod('c');
        setime(200, 0);
        move(p0);
        waitfor(completed);
        t0 = rtime;
        for (i = 0; i < 6; ++i) {
                move(pt1);
                move(pt2);
        }
        setvel(400, 100);
        setmod('j');
        move(park);
}

tablefn(t)
TRSF_PTR t;
{
        real rps = .03;
        real alpha = rps * (t0 - rtime) * .001;

        t->p.x = 100. * cos(alpha * pit2_m);
        t->p.y = 100. * sin(alpha * pit2_m);
        Rot(t, zunit, alpha * 360.);
}
```

The positions equations set apart, this program is quite similar to the previous one. The main difference lies in the way those equations are set up in order to obtain the desired effect. The functionally described transformation is made up from a translation part and a rotation part. The variable 'rps' describes a rotational velocity of 3/100 of a rotation per second. The variable **pit2_m** belong to the set of math constant entry points.

4) The last example describes a task that causes the manipulator end effector to follow a circular path while always being perpendicular to its trajectory. This achieved by setting up a position equation to obtain a remote center of rotation.

```
#include "rccl.h"


pumatask()
{
        TRSF_PTR z, e , b, perp0, perp, roty;
        POS_PTR  p0, pt;
        int perpfn();

        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 170.);
        b = gentr_trsl("B", 500. ,  300., 600.);
        roty = gentr_rot("ROTY", 0., 0., 0., yunit, 180.);
        perp0 = gentr_rot("PERP0", 0., 0., 300., xunit, 0.);
        perp = newtrans("PERP", perpfn);

  p0 = makeposition("P0", z, t6, e, perp0, EQ, b, roty, TL, e);
  pt = makeposition("PT", z, t6, e, perp0, EQ, b, perp, roty, TL, e);

        setvel(400, 100);
        setmod('c');
        setime(300, 0);
        move(p0);
        setime(200, 4000);
        move(pt);
        setmod('j');
        move(park);
}

perpfn(t)
TRSF_PTR t;
{
        real rpm = .20;

        Rot(t, xunit, rpm * goalpos->scal * 360.);
}
```

In this example, the functional motion parameter is the 'scal' position structure entry. When the background function is evaluated, the global **goalpos** pointer is equal to 'pt'. The variable 'rpm' stands for rotations per motion.

We have introduce some examples for coding functionally described trajectories. The lay out of the programs, especially the position equation specifications are certainly not unique, and a lot of room is left to imagination.

6. Sensor Integration

By using sensors, the user has the ability to write programs that may depend on information acquired at run time. The behavior of the manipulator can be influenced by modifying the locations it is moving to or by interrupting a motion. If the location can be determined ahead of time, we shall call that presetting the world model. A special case is the transforms initializations. If the locations can be determined synchronously and permit us to influence the manipulator's path, we shall call that tracking. If the locations can be determined by stopping the manipulator on condition, we shall call that guarded motions. If the final position of the manipulator is to be retained for the determinations of locations, we shall call that updating the world model.

6.1. Presetting the World Model.

In the section 'Synchronization' we have already met such a situation. The example of a program interacting with a user was given :

```
for ( ; ; ) {
        printf("enter Z increment ");
        scanf("%f", &iz);
        b->p.z += iz;
        move(p);
}
```

The *hold* transform feature allowed us the specify different locations ahead at time and no synchronization is specified.

Let us consider the integration of a computer vision system. We assume that a camera has been attached to the link 4 of the PUMA manipulator. The computer vision is described in terms of a functions 'snapshot' which is supposed to take a picture of the scene and store it in memory, and of a function 'getobj' able to extract the position and orientation of an object in the camera coordinate frame. The operation of taking the picture is expected to be short but the task is programmed in such a way that the processing time of 'getobj' does not require to stop the arm. The strategy consist of moving the manipulator toward a position where we expect an object to be captured in the field of the camera. We synchronize the program such as the picture is taken at a given point of the trajectory. We also record at this instant the position of the manipulator given by $T6$ and we reconstruct the camera coordinate frame from the transform $U5$ internally maintained by the system. We could also compute the values of the transform $T4$ since we can know at any moment the joint angles values (see include files). Thus we have all the information necessary to perform an approach motion where the object has been found, grasp it, and move it to some other place.

A bare bone version of the task is described in terms of three position equations : the position where to expect an object to be seen by the camera, the position where to grasp the object, and the position where to put it :

```
#include "rccl.h"
#include "hand.h"
#include "which.h"
#include "kine.h"

pumatask()
{
        TRSF_PTR  z, e, cam, o, coord, t6r, u5i, expect, drop;
        POS_PTR look, get, put;

        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 170.);
        cam = gentr_rot("CAM", 0., 0., 50., xunit, 90.);
        expect = gentr_rot("EXPECT", 500. , 100., 600., yunit, 180.);
        drop = gentr_rot("DROP", 400. , -100., 500., yunit, 180.);
        o = newtrans("O", hold);
        coord = newtrans("COORD", hold);

        u5i = newtrans("U5I", varb);
        t6r = newtrans("T6R", varb);

        look = makeposition("LOOK", z, t6, e, EQ, expect, TL, e);
        get = makeposition("GET", t6, e, EQ, coord, cam, o, TL, e);
        put = makeposition("PUT" , z, t6, e, EQ, drop, TL, e);

26
27          setvel(200, 100);
28          for (; ; ) {
29                  move(look);
30                  waitas(goalpos == look && look->scal > .8);
31                  snapshot();
32                  Assigntr(t6r, t6);
33                  Invert(u5i, &sncs_d.u5);
34                  Trmult(coord, t6r, u5i);
35                  if (!getobj(o)) {
36                          break;
37                  }
38                  get->end = 0;
39                  distance("dz", -30.);
40                  move(get);
41                  move(get);
42                  stop(50);
43                  distance("dz", -30.);
44                  move(get);
45
46                  waitfor(get->end);
47                  waitfor(get->end);
48                  CLOSE;
49                  printf("closing\n");
50                  move(put);
51                  waitfor(put->end);
52                  OPEN;
53                  printf("opening\n");
54          }
55          move(park);
56  }
57
58  snapshot()
59  {
60          printf("snap\n");
61  }
62
63  getobj(t)
64  TRSF_PTR t;
65  {
66          static int number = 5;
67
68          Trsl(t, 0., 0., 200. + number * 30.);
69          Rot(t, yunit, 10. * number);
70          return(number--);
71  }
```

Line 1 includes RCCL declarations. Line 2 includes the file **hand.h** that contain the two macros 'OPEN' and 'CLOSE' to actuate the pneumatic gripper. Line 4 includes the file **kine.h** that contains manipulator dependent informations about the kinematics. This file contains the structure declarations and external declarations of variables internally used by RCCL. Since this file depends on the manipulator type it must be preceded with the definition of the particular manipulator ('PUMA' for the Puma 600, 'STAN' for the stanford arm). The file **which.h** included line 3 contains the line : "#define

PUMA" that describe the current implementation. We are primarily interested in the variable called **snsc_d** declared in **kine.h** as :

```
typedef struct sincos {
        real c1, s1, c2, s2, c23, s23, c3, s3, c4, s4, c5, s5, c6, s6;
        real d1x, d1y, d1z, r1x, r1z, d2x, d2y, d2z, d3x, d3y, d3z;
        real h;
        TRSF u5;
} SNCS, *SNCS_PTR;

extern  SNCS  sncs_d;
```

The elements of **sncs_d** are kinematic parameters updated at sample time that the user may use. For the Puma manipulator, the first line is the list of sines and cosines values of each joint angles. The second and third line exhibit the coefficients of the Jacobian matrix that contain multiplications computed in link 4 [9]. The fourth line of type 'TRSF' is the transform $U5$ and we shall make use of it.

Back to the program, after the pointer declaration we find, lines 11 through 20, the allocation of transforms. For simplicity, we will name them the same way as the frames that they describe.

Z : a reference frame at the base of the manipulator.

E : the end effector frame.

CAM : the camera frame described with respect to link 4.

EXPECT : the position where we expect to find an object in the camera field with respect to the Z frame.

DROP : the position from where we would like the manipulator to drop the object.

O : The position of the object in camera coordinates that we declare as *hold* since it will be changed at during the task execution.

COORD : The position of the camera in base coordinates that is changing as the manipulator moves.

U5I and T6R : auxiliary transforms that are used to hold the inverse of U5 and a copy of T6 at the moment of taking the picture. They will be used to compute the transform COORD, but are not used in a position equations.

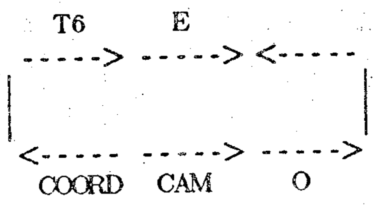
The three calls to makeposition, lines 22, 23, and 24 define to following transform graphs :

LOOK :

$$\begin{tikzcd}
{} \arrow[r, dashed, "T6"] \arrow[d, dash] & {} \arrow[r, dashed, "E"] & {} \arrow[d, dash] \\
{} & {} \arrow[l, dashed, "Z"'] \arrow[r, dashed, "EXPECT"'] & {}
\end{tikzcd}$$

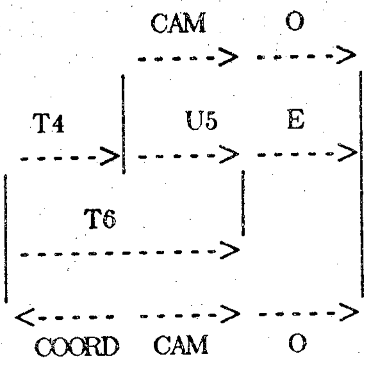
This first graph is quite ordinary.

GET :



This graph describes the final position entirely in manipulator coordinates. The transforms *COORD* and *O* are measured at the moment of taking the picture and their values memorized in the motion queue. It is mapped on the equivalent graph :

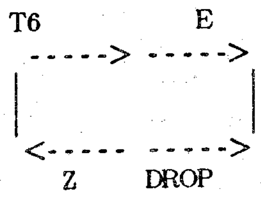
GET :



from which we can derive :

$$COORD = T6\ U5^{-1}$$

PUT :



Again, an ordinary graph.

Statement, line 27, sets the velocity and one can notice that the motion type is left in *joint* mode, since we are more interested in the end of path segment position than with the trajectories. The body of the program is essentially a loop that will exit when the function 'getobj' returns zero value. The function 'getobj' is simulated and returns five different successive values for the *O* transform.

At the beginning of the loop, line 29, a motion is requested toward the 'LOOK' position. The program is then synchronized such that 'snapshot' is called at 80 per cent of the trajectory. Note that a 'waitas(look->scal > .8)' statement may not be sufficient since after execution of the first loop the value of 'look->scal' is left at 1. as a side effect of the previous loop, and no synchronization would be achieved. Values of *T*6 and *U*5 are copied on the fly, and the *COORD* transform value computed, lines 32 to 34. The function 'getobj' has the remaining trajectory time to obtain of value for *O*. An 'approach - grasp - depart' sequence is then performed before the next loop, lines 39 to 44.

This program is only a sketch and many improvements are possible. For example, we may like to interrupt the motion and proceed to the grasp sequence as soon as a value for *O* is obtained. We shall see in the next section how to achieve this result. Another variation would be to introduce a conveyor carrying objects in the vision field of the camera. The only required modifications would be the position equation for 'GET' :

```
        get = makeposition("GET", z, t6, e, EQ, conv, coord, cam, o, TL, e);
```

This is because the object position information is entirely contained in the transforms $T4$, CAM, and O, and the position 'GET' would be tracking the conveyor.

6.2. Guarded Motions

This section explains how to interrup a motion on condition. Interrupting a motion can allow us to stop or to start the arm, to suspend or resume a motion toward a position. This can be achieved in all cases by setting the flag called **nextmove** at a non zero value. This causes the motion currently being performed to be interrupted and the next in the queue to begin. The value to which the flag **nextmove** has been set is returned in the 'code' field of the structure 'POS', described earlier. When using this feature, the user must be careful not to conflict with values that have a predetermined meaning for RCCL :

```
#define OK      -1
#define LIMIT   -2
#define ONF     -3
#define OND     -4
```

For example, one could use only positive values.

There are basically two ways of using the flag **nextmove**. It can be set either from a background function, or from the user's foreground process. Let us illustrate this by an example using a simple sensor which is a small linear potentiometer. The distance of which the shaft is inside the body of the potentiometer can be measured through analog channels. The robot controller performs analog to digital conversions when specified via the function **adcopen** and the values can be read from a global array updated at sample intervals [6]. We will make use of this information to program guarded motions.

```
#include "rccl.h"
#include "hand.h"
#include "rtc.h"
#include "umac.h"

extern struct how how;

static int sensor;

#define TOUCHED 10

pumatask()
{
        int touchfn();
        TRSF_PTR z, b1, b2, fing, getit, flip;
        POS_PTR  get, p1, p2;
        int q;

        z = gentr_trsl("Z",  0.,  0., 864.);
        b1 = gentr_trsl("B1", 600. ,-200., 400.);
        b2 = gentr_trsl("B2", 600. ,-100., 400.);
        fing = gentr_rot("FING", 0., 0., 200., zunit, -90.);
        getit = gentr_rot("GETIT", 600. , 0., 600., yunit, 180.);
        flip = gentr_rot("FLIP", 0., 0., 0., yunit, 180.);

        p1 = makeposition("P1" , z, t6, fing, EQ, b1, flip,TL , fing);
        p2 = makeposition("P2" , z, t6, fing, EQ, b2, flip,TL , fing);
        get = makeposition("GET", z, t6, EQ, getit, TL, t6);

        setvel(300, 100);
        move(get);
        waitfor(completed);
        OPEN;
        printf("put the sensor in the jaws ");
        QUERY(q);
        CLOSE;
        printf("go ahead ");
        QUERY(q);
        if (q == 'n') {
                move(park);
                return;
        }
43          sensor = adcopen(7);
44          setvel(100, 100);
45          for (; ; ) {
46                  p1->end = p2->end = 0;
47                  move(p1);
48                  evalfn(touchfn);
49                  setime(0, 0);
50                  distance("dz", 100.);
51                  move(p1);
52                  move(p1);
53
54                  move(p2);
55                  evalfn(touchfn);
56                  setime(0, 0);
57                  distance("dz", 100.);
58                  move(p2);
59                  move(p2);
60
61                  waitfor(p1->end)
62                  printf("guarded motion 1 starts\n");
63                  waitfor(p1->end)
64                  if (p1->code == TOUCHED)
65                          printf("touched\n");
66                  else
67                          printf("not touched\n");
68
69                  waitfor(p2->end)
70                  printf("guarded motion 2 starts\n");
71                  waitfor(p2->end)
72                  if (p2->code == TOUCHED)
73                          printf("touched\n");
74                  else
75                          printf("not touched\n");
76
77                  printf("more ? ");
78                  QUERY(q); if (q == 'n') break;
79          }
80          setvel(300, 100);
81          move(park);
82          waitfor(completed);
83          OPEN;
84  }
85
86
87  touchfn()
88  {
89          if (how.adcr[sensor] > 1) {
90                  nextmove = TOUCHED;
91          }
92  }
```

We are now familiar with lines 1 and 2. Line 3 includes the real time interface declarations [6], in order to gain access to the analog conversions. Line 4 includes the file **umac.h** that contains a set of useful macros (see include files), among them we shall use the macro 'QUERY' that causes the prompt " (y/n) " to be printed on the terminal and will return when the user has typed a 'y' or a 'n'. The typed character is then returned in the macro's argument. Line 6 declares the type of the array in which we get the analog readings. Line 8 allocates an integer that will be the index in the array 'adcr' of the readings of the opened analog channel. Line 10 defines the return code of the guarded motion. Let us skip the transforms and position initializations that do not show anything special. With a combination of queries we ask the operator to place the sensor in the gripper's jaws and to command the gripper to close. We leave to the operator the option of canceling the task on line 39 if something goes wrong. Line 43 allocates analog channel number 7 to the sensor. In the body of the loop, lines 45 to 78, the manipulator performs two guarded motions : moves to a position next to an expected obstacle, moves along the Z direction in the *tool* frame, while evaluating at sample intervals the monitoring function 'touchfn'. The call to **setime** specifies a zero transition time at the end of the motion in order to obtain a fast reaction time. The null transition time can be specified here as we have made sure that the velocities are small. We also make sure that the motion queue contains a position such as the arm will back up when the obstacle in encountered. This is the purpose of the move statements lines 52 and 59.

Using the **waitfor** macro, the program can print information at the terminal as the task proceeds. In particular, it is possible to decided if the guarded motion did encounter an obstacle. The value 'TOUCHED' is returned in the 'code' part of the positions if the monitoring function caused a motion interruption, otherwise the value 'OK' is returned. The monitoring function, lines 86 to 91, checks the analog conversion reading and sets the flag **nextmove** when appropriate.

Some other combinations are possible, as shown by the following code patterns :

```
move(p);
evalfn(monitorfn);
setime(100, 10000);
move(p);
waitfor(completed)
if (p->code != EXPECTED) {
        printf("timeout after 10 seconds\n");
        error();
}
move(p1);
```

causes the arm to stop at the position 'p' while evaluating a monitor function, and the motion to resume on condition. It is not possible to use a **stop** statement here, since **stop** keeps all the attributes of the previous motion and we need to specify a new **move** request. The sequence of code :

```
evalfn(monitorfn);
move(p);
stop(1000);
move(p);
```

does not causes a motion to be interrupted for one second, unless the position 'p' has been reached when the **stop** request is executed because it is equivalent to a new motion to the last position. Similarly, one must be careful that the **stop** statement does not necessarily mean that the manipulator will stop in absolute coordinates if the position equation for 'p' contains moving coordinate frames. When an absolute stop is needed or when a motion has to stop and the manipulator has to remain exactly at the position where it stopped, RCCL provides a built-in position equation of the following form :

$$T6 = HERE$$

where *HERE* is a transform internally maintained by the system to be always equal to $T6$ at the end of any path segment. At startup time, the system issues the following call :

```
there = makeposition("THERE", t6, EQ, here, TL, t6);
```

to implement this feature, where **here** is of type TRSF_PTR and **there** of type POS_PTR. The following code pattern shows how we can use the fact that the flag **nextmove** can be set from the user's process to implement a stop on terminal input :

```
move(p);
move(there);
printf("hit return to interrupt motion ");
getchar();
nextmove = YES;
```

When 'return' is hit, the system interrupts the motion toward 'p', and starts a transition to the position 'there' that causes the arm to over shoot by a magnitude as great as the velocity was high. When the velocity is small and a sharp stop is needed we can write :

```
move(p);
setime(0, 0);
move(there);
printf("hit return to interrupt motion ");
getchar();
nextmove = YES;
```

In the same way monitoring can achieved with :

```
move(p);
waitas(goalpos == p)
p->end = 1;                          /* preset event */
while(p->end) {
        if (condition) {
                nextmove = YES;
        }
        suspendfg();
}
```

This way of coding can be useful in the cases when it is not possible to place the condition calculations in the background function.

RCCL internally monitors if the joint physical limits are going to be reached (within a few degrees for each joint). If such an error condition occurs, the system automatically issues a move to the 'there' position, that causes an immediate stop next to the limit position and returns the code 'LIMIT' for the motion that caused the error condition. A new motion is then taken out of queue and the error condition is reset. If the new motion persists in causing a joint limit error, the whole task will abort. If motions are likely to cause such joint limit errors, the returned codes should be checked and the appropriate action undertaken.

6.3. Tracking

Tracking is obtained by synchronously updating functionally defined transforms from sensor readings. All the examples given in the section "Functionally defined motions" would become examples for tracking motions if we would replace the parameter 'time' by some sensor readings reflecting the position of the moving coordinate frames. We shall however explain yet another example using the simple potentiometer based position sensor introduced in the previous section. The sensor, placed in the *tool* frame, allows the manipulator to track an arbitrary surface intersecting the Z axis of the tool frame. The tracking function is written in such a way that it causes the motion velocity to be proportional to the distance the shaft of the linear potentiometer is protruding out of the sensor. A velocity control of the manipulator end effector is implemented such as that the shaft is partially inside the body of the sensor, the velocity along the Z axis of the *tool* frame is controlled to be zero.

```
#include "rccl.h"
#include "rtc.h"
#include "umac.h"

extern struct how how;

int sensor;

pumatask()
{
        TRSF_PTR z, b1, b2, fing, fins, over;
        POS_PTR  p1, p2, get;
        int fingfn();
        int q;

        fing = newtrans("FING",fingfn);
        Rot(fing, zunit, -90.);
        fins = gentr_rot("FINS", 0., 0., 0., zunit, -90.);
        z = gentr_rot("Z",  0.,  0., 864., zunit, 0.);
        b1 = gentr_rot("B1", 600. ,-300., 450., yunit, 180.);
        b2 = gentr_rot("B2", 600. , 300., 450., yunit, 180.);
        over = gentr_rot("OVER", 600., 0., 600., yunit, 180.);

        p1 = makeposition("P1" , z, t6, fins, EQ, b1, TL, fins);
        p2 = makeposition("P2" , z, t6, fing, EQ, b2, TL, fing);
        get = makeposition("GET", z, t6, EQ, over, TL, t6);


        sensor = adcopen(7);

        setmod('c');
        for (; ; ) {
                setvel(400, 300);
                move(get);
                move(p1);
                setvel(100, 100);
                sample(15);
                move(p2);
                sample(30);
                printf("more ?"); QUERY(q); if (q == 'n') break;
        }
        setvel(400, 300);
        setmod('j');
        move(park);
}



fingfn(t)
TRSF_PTR t;
{
        t->p.z += (how.adcr[sensor] * .01 - 3.) / 3.;
}
```

This example uses three positions equations. The position P1 from where the tracking motion begins, uses a *TOOL* transform FINS set as a translation and a rotation such as to present the sensor with a proper angle. The position P2 is the end of the tracking motion, the *TOOL* transform FING is initialized to be equal to FINS. However, as the motion progresses, its p_z element will be changed by the function 'fingfn' in order modify the trajectory in the desired direction. The function 'fingfn', lines 49 to 53, implements the control law whose parameters have been experimently determined, in terms of a gain and an offset. The sample rate is set at a higher value during the tracking motion in order to obtain a faster response.

An interresting variation of this program would be to record the value of $T6$ as the tracking proceeds. Since it is not possible to perform any input-output from a background function, a buffer alternating technique would need to be implemented : while the background function fills one buffer, the user's foreground process could dump another one on file. It would then become possible to replay very long motion sequences, as they have been recorded or in such a way that the *tool* frame would have a fixed angle with respect to the tracking trajectory as required in welding applications (Section "Functionally defined motions", example 4).

When the sensory input is too slow or when computations are too lengthy to be performed in a background function (10 ms cpu time every 28 ms would really tie the machine down), a pseudo tracking can be obtained by using an asynchronous loop in the user's process updating a *varb* type transform. For example :

```
TRSF_PTR z, e, b, r;
POS_PTR p0, p1, ... , pt;
TRSF changes;

z = ...
e = ...
b = ...
r = ...
upd = newtrans("UPD", varb);

p0 = makeposition(...);
p1 = makeposition(...);
pt = makeposition("P", z, t6, e, EQ, b, upd, r, TL, e);

...

move(p0);
move(p1);
move(pt);

waitas(goalpos == pt)
while(goalpos == pt) {
        getsensor(&changes);
        Trmultinp(upd, &changes);
        suspendfg();
}
....
```

The function 'getsensor' returns alterations to be performed on the transform *UPD*, that are accumulated by successive multiplications. The function **suspendfg** is used to allow the machine to "breath". The changes should not cause more than a few millimeters or degrees position steps at the end effector.

The **update** function causes the position equation 'pos' to be solved for the transform 'trans', using the value of *T6* at the end of the corresponding path segment. Of course, the transform must belong to the position equation. The transform must also be of type *varb*. The second argument, a position equation pointer, is not necessarily the same argument as the one of the corresponding motion statement. For example, we can update a transform on user request :

```
a = gentr...
e = gentr...
y = gentr...
z = gentr...

x = newtrans("X", varb);

p1 = makeposition("P1", z, t6, e, EQ, a, TL, e);
p2 = makeposition("P2", z, t6, e, EQ, x, y, TL, e);

...

update(x, p2);
move(p1);
move(p2);
printf("hit return to interrupt motion ");
getchar();
nextmove = YES;
```

An update request is associated with position P1. When the user hits 'return', the motion toward P1 is interrupted and the transform X is solved as :

$$X = Z\ T6\ E\ Y^{-1}$$

and the position P2 corresponds exactly to the position the manipulator was and a stop is obtained. Subsequent motions to this position are therefore possible. The transform X can also be used in other position equations. One must notice that all the positions containing X are consequently changed. This leads to numerous applications of **update.**

7. Force Control

In assembly tasks, objects are required to be brought into contact, and motions have to be stopped when the collision occurs. Once objects are in contact the task is said to be constrained because arbitrary motions are no longer possible in every direction. The notion of guarded motion has been introduced earlier and force guarded motions are quite similar. The force monitoring function is built into the system considering that it is somewhat dependent on the installation and that force specifications are really an integral part of a motion description. Force specifications are transmitted to the background process via the **limit** primitive. When the task is constrained, the arm is requested to exert forces on objects and is no longer position servoed for each of the six degrees of freedom. Depending on the geometry of the task, one or several directions are selected to provide for compliance. The arm is then said to enter a *comply* mode which is specified by the **comply** primitive. When the contact between objects ceases, or when constraints disappear, the arm has to gain back position servoed directions. This is achieved by the **lock** primitive. The cessation of contact can be detected by differential motions of the arm when the constraints disappear. The primitive **limit** is also used for this purpose.

7.1. Stop, Go on Force, on Displacement

As we have seen before, stop and go are not essentially different, they both correspond to the interruption of a motion. When a limit condition is specified, a monitor function is internally activated for the subsequent motion. The form of **limit** is :

```
limit(dirs, value [, value] ...)
char *dirs;
double value;
```

The limit directions specifications are expressed in the string first argument with a combination of the following, separated by one or several spaces :

```
fx fy fz : force        along X Y Z
tx ty tz : torque       about X Y Z
dx dy dz : displacement along X Y Z
rx ry rz : rotation     about X Y Z
```

All limit specifications are expressed in the *tool* frame of the corresponding position equation and take effect for the subsequent *move* request. To each direction specification must correspond a value, for example :

```
limit("fx tz", 10., 5.);
```

will request a force of 10 **Newtons** along X, and of a torque 5. **Newton-meter** about Z to be monitored. When either of the specifications is exceeded, the corresponding motion is interrupted and the task proceeds with the next request in the motion queue. The 'code' field of the corresponding position is set to 'ONF'. Likewise, distance specifications can be coded as :

```
limit("dx ry", 3., 1.);
```

that causes the motion to be interrupted when the distance change along X exceeds 3 **millimeters** or when the rotations about Y exceeds 1 **degree.** Only absolute values of the limit specifications are taken into account.

7.2. Servo Modes, Comply and Lock

A comply servo mode is requested via the **comply** primitive according to the following format :

```
comply(dirs, value [, value] ...)
char *dirs;
double value;
```

The **comply** primitive causes the subsequent motion request to be executed such that forces and/or torques are maintained in the *tool* frame instead of positions and/or rotations. The arm then enters the specified *comply* mode in the corresponding motion and all the following motions until the **lock** primitive brings the manipulator back into position servo mode for the selected directions. The format for **lock** is :

```
lock(dirs)
char *dirs;
```

The first argument of **comply** and **lock** is a string containing direction specifications made up of a combination of the following :

```
fx fy fz : force  along X Y Z
tx ty tz : torque about X Y Z
```

The second argument of **comply** is the signed magnitude of the desired forces and/or torques.

Care must be taken that the sequence of servo mode specification is consistent. Requiring the arm to comply along or about a direction already in *comply* servo mode or locking a direction not in *comply* servo mode will cause an error. In order to keep track of the different specifications, line indentation is advised :

```
move(p0);
comply("dy", 0.);
          move(p1);
          move(p2);
          comply("dx ry", 5., 3.);
                              move(p3);
                    lock("ry");
                    move(p4);
                    move(p5);
          lock("dx");
          move(p6);
lock("dy");
move(p7);
```

Either *Cartesian* or *joint* motion modes can used for complying motion sequences. However, they behave differently. In *Cartesian* mode, the system automatically compensate for position errors due to unwanted accommodating joint motions [2]. In *joint* mode, there is no compensation.

7.3. Carrying Loads

The function **massis** allows the user to specify that a mass is to be carried by the manipulator. The mass of the object, expressed in kilograms, causes the system to compute gravity compensation terms in the motions using force control. The mass of object is initially set to 0. and can be set or reset via **massis**:

```
massis(mass)
double mass;
```

As usual, the new value is taken into account the next motion request.

7.4. Examples

1) The first example involves a solid horizontal surface. The manipulator is programmed to reach to the table in a motion normal to the expected surface. It then enters the *comply* mode in order to slide along. During the second sliding motion, it detects an edge of the surface by exerting a preload force and monitoring the position changes in the Z direction of the *tool* frame.

```
#include "rccl.h"
#include "umac.h"

pumatask()
{
        TRSF_PTR z, e , b1, b2, b3, b4, b5;
        POS_PTR  p1, p2, p3, p4, p5;
        int q;

        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 170.);
        b1 = gentr_rot("B1", 600. ,-100.,  300., yunit, 180.);
        b2 = gentr_rot("B2", 600. , 200.,  300., yunit, 180.);
        b3 = gentr_rot("B3", 600. , 200.,  400., yunit, 180.);
        b4 = gentr_rot("B4", 600. ,-100.,  400., yunit, 180.);
        b5 = gentr_rot("B5", 500. ,-100.,  300., yunit, 180.);

        p1   = makeposition("P1" , z, t6, e, EQ, b1, TL, e);
        p2   = makeposition("P2" , z, t6, e, EQ, b2, TL, e);
        p3   = makeposition("P3" , z, t6, e, EQ, b3, TL, e);
        p4   = makeposition("P4" , z, t6, e, EQ, b4, TL, e);
        p5   = makeposition("P5" , z, t6, e, EQ, b5, TL, e);


        setmod('c');
        setvel(200, 100);
        move(p4);
        for (; ; ) {
                QUERY(q); if (q == 'n') break;

                p1->end = 0;

                setvel(20, 20);
                limit("fz", 20.);
                move(p1);

                setvel(100, 50);
                comply("fz", 10.);
                        move(p2);
                lock("fz");


                waitfor(p1->end);
                if (p1->code != ONF) {
                        nextmove = YES;
                        printf("where is the table ?\n");
                        setvel(200, 100);
                        move (park);
                        return;
                }

                move(p3);
                move(p4);

                limit("fz", 20.);
                setvel(50, 50);
                move(p1);

                limit("dz", 3.);
                comply("fz", 10.);
                        move(p5);
                lock("fz");

                move(p4);

                waitfor(p5->end);
                if (p5->code != OND) {
                        printf("where is the edge ?\n");
                }
        }
        setvel(300, 50);
        move(park);
}
```

The transforms and positions define a set of five position next to the surface. The surface if assumed to be less than 100 millimeters below the position P4, such that a collision should occur when moving from P4 to P1, located 100 millimeter below P4. Position P3 is at the same level than P4 and above P2. Position P5 is assumed to bring the end effector off the boundaries of the table, when moving from P1 to P5. As we may have more motions toward P1 that waits for the end of motion, the 'p1->end' event is reset for each loop line 31. Lines 33 to 40, implement a sequence of motion requests so as to program the manipulator to enter the *comply* when the obstacle in encountered. Lines 43 through 50, we make sure that the limit have actually been met, otherwise the motion toward P2 is canceled as well as the task. In the normal case, lines 52 and 53 bring the arm back above the surface. The same guarded motion is then performed toward P1, but now the motion in *comply* servo mode is performed toward P5 where the edge of the surface is expected to be found. The termination of this last motion is also checked at lines 66 through 69. A preload force in the Z direction is applied for all motions to make sure that the contact is maintained.

2) In this second example, the manipulator is programmed for the task of turning a crank. Two compliant directions are required in this operation. During the compliant motion, the *tool* frame rotates so as to keep a constant orientation with respect to the crank handle. We define the compliant directions with respect to this frame. A grasp position is also defined to allow for some clearance. The task will turn the crank a given number of times. One turn is programmed to last 4 seconds.

```
#include "rccl.h"
#include "umac.h"
#include "hand.h"

int turns;

pumatask()
{
        TRSF_PTR z, e, shaft, handle, apro, grasp, rotpx, rotnx;
        POS_PTR  get, away;
        POS_PTR turn;
        int pxfn(), nxfn();
        int q;

        rotpx = newtrans("ROTPX", pxfn);
        rotnx = newtrans("ROTNX", nxfn);
        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 170.);
        shaft = gentr_trsl("SHAFT", -200., 500., 600.);
        shaft->fn = varb;
        handle = gentr_trsl("HANDLE", 0., 0., 50.);
        apro = gentr_trsl("APRO", -50., 0., 0.);
        grasp = gentr_rpy("GRASP", 0., 0., 0., 0., 190., 0.);

        get = makeposition(
        "GET", z, t6, e, EQ, shaft, handle, grasp, TL, e);

        away = makeposition(
        "AWAY", z, t6, e, EQ, shaft, handle, grasp, apro, TL, e);

        turn = makeposition(
        "TURN", z, t6, e, EQ, shaft, rotpx, handle, rotnx, grasp, TL, r

        setvel(300, 300);
        move(away);
        OPEN;
        if (!teach(shaft, get)) {
                move(away);
                move(park);
                return;
        }
42              shaft->fn = const;
43              optimize(turn);
44              turns = 4;
45              waitfor(completed);
46              CLOSE;
47              comply("fx fz ", 0., 0.);
48                      movecart(turn, 200, 4000 * turns);
49              lock("fx fz ");
50              move(get);
51              waitfor(turn->end);
52              OPEN;
53              distance("dx", -30.);
54                      move(get);
55              setvel(200, 50);
56              setmod('j');
57              move(park);
58      }
59
60      pxfn(t)
61      TRSF_PTR t;
62      {
63              Rot(t, xunit, goalpos->scal * 360 * turns);
64      }
65
66      nxfn(t)
67      TRSF_PTR t;
68      {
69              Rot(t, xunit, - goalpos->scal * 360 * turns);
70      }
```

The manipulator is first moved to a safe position away from obstacles. Lines 37 to 41, the manual teach mode built in RCCL is called. This teach mode makes use of the **update** function to record a position. That is why the transform *SHAFT* is first declared as a *varb* transform. Once this transform is taught, its type can be changed, line 42, and the position *TURN* optimized line 43. Gripper actions are obtained as usual. Once these preliminary operations are performed, the turning motion can begin. It is obtained in terms of a functionally defined motion, line 48, executed in *comply* servo mode. The duration of the motion is the number of turns times four seconds. Care has been taken line 32, such that the compliance frame is properly specified.

3) The third example illustrates the peg in a hole insertion task. The strategy consists of moving toward the expected location of the hole with a small approach angle. Even with a poor position accuracy the end of the peg will enter the hole with a high degree of confidence. As soon as a collision occurs, the manipulator is programmed to go in *comply* mode in the Z direction with a preload force in the same direction. While complying, the peg is rotated so as to be aligned with the axis of the hole. The manipulator is then programmed to comply in the normal directions of the hole axis and a motion inside the hole is immediately started. The presence of a small chamfer helps the peg not to slip off the initial insertion position. The force in Z is also limited since the insertion may jam due to a misalignment. The fit is not very tight, and we can expect that a portion of the peg is inserted before the jam occurs. A sequence of four accommodation rotating motions using **update**, allows the manipulator to "feel" the walls of the hole and to record a correct alignment. In a final effort, the peg is inserted all the way. Finally, the peg is pulled out with no difficulty since the alignment has been corrected. The moment when the task becomes unconstrained is detected by monitoring the differential motions.

```
#include "rccl.h"
#include "umac.h"
#include "hand.h"

pumatask()
{
        TRSF_PTR z, e, peg, hole, roty, bottom, angle;
        POS_PTR align, in, touch;
        int q;

        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0.  , 0.  , 140.);
        peg = gentr_trsl("PEG", 0., 0., 10.);
        hole = gentr_trsl("HOLE", -50., 450., 500.);
        hole->fn = varb;
        bottom = gentr_trsl("BOTTOM", 0., 0., -20.);
        roty = gentr_rot("ROTY", 0., 0., 0., yunit, 180.);
        angle = gentr_rot("ANGLE", 0., 0., 0., yunit, 12.);

        align = makeposition(
        "ALIGN", z, t6, e, peg, EQ, hole, roty, TL, peg);

        touch = makeposition(
        "TOUCH", z, t6, e, peg, EQ, hole, angle, roty, TL, peg);

        in = makeposition(
        "IN", z, t6, e, peg, EQ, hole, bottom, roty, TL, peg);

        setvel(300, 50);
        move(align);
        if (!teach(hole, align)) {
                setvel(300, 50);

                distance("dz", -100.);
                        move(align);

                setmod('j');
                move(park);
                return;
        }
        setmod('c');
        setvel(100, 100);

        distance("dz", -10.);
                move(touch);

        QUERY(q);

        if (q == 'n') {
                setvel(300, 100);
                setmod('j');
                move(park);
                return;
54          }
```

```
        setvel(4, 7);
        distance("dz", -4.);
                move(touch);

        limit("fz", 25.);
        distance("dz", 5.);
                move(touch);

        comply("fz", 15.);
                move(align);
        lock("fz");

        comply("fx fy", 0., 0.);
                limit("fz", 20.);
                move(in);

                update(hole, in);
                limit("fz tx", 40., 10.);
                distance("rx", 2.);
                        move(in);

                update(hole, in);
                limit("fz tx", 40., 10.);
                distance("rx", -2.);
                        move(in);

                update(hole, in);
                limit("fz ty", 40., 10.);
                distance("ry", 2.);
                        move(in);

                update(hole, in);
                limit("fz ty", 40., 10.);
                distance("ry", -2.);
                        move(in);

                update(hole, in);
                limit("fz", 20.);
                distance("dz", 10.);
                        move(in);

                limit("dx dy", 1., 1.);
                move(align);
        lock("fx fy");

        setvel(50, 50);
        distance("dz", -50.);
                move(align);

        setvel(300, 50);
        setmod('j');
        move(park);
}
```

The beginning of this task is quite similar to the previous example and also makes use of the manual teach mode to record an approximate initial position. Lines 44 and 45, the manipulator moves to an approach position and a chance is given to the user to cancel the task. Line 57 to 63, an approach motion and a purposely over shooting motion is programmed in order to obtain the initial phase of the insertion process. While complying and exerting a preloading force the peg is rotated, lines 65 to 67, to the aligned position. The first insertion attempt is performed line 70 and 71. Lines 70 to 91, are programmed the accommodation motions using **update** to record the successive alignments. The final phase of the insertion process is performed lines at 93 to 96. The peg is then pulled out of the hole, while monitoring the differential motions signaling that the motion becomes unconstrained. The peg is then taken away lines 102 to 104.

4) The last example demonstrates how compliant degrees of freedom can be accumulated as constraints are met. The manipulator is programmed to detect the walls of a corner and to record the position of the corner. The program then uses this position information to move the manipulator next to the corner within a very small distance.

```
#include "rccl.h"
#include "umac.h"

pumatask()
{
        TRSF_PTR z, e, peg, corner, roty;
        POS_PTR pcor;
        int q;

        z = gentr_trsl("Z",  0.,  0., 864.);
        e = gentr_trsl("E" , 0. , 0. , 140.);
        peg = gentr_trsl("PEG", 0., 0., 10.);
        corner = gentr_trsl("CORNER", -50., 500., 550.);
        corner->fn = varb;
        roty = gentr_rot("ROTY", 0., 0., 0., yunit, 180.);

        pcor = makeposition(
        "PCOR", z, t6, e, peg, EQ, corner, roty, TL, peg);

        setvel(300, 50);
        move(pcor);
        if (!teach(corner, pcor)) {
                setvel(300, 50);

                setmod('j');
                move(park);
                return;
        }
        setmod('c');
        setvel(100, 100);

        distance("dz", -50.);
                move(pcor);

        QUERY(q);

        if (q == 'n') {
                setvel(300, 100);
                setmod('j');
                move(park);
                return;
        }
        move(pcor);

        setvel(5, 20);
46
47          limit("fz", 20.);
48          distance("dz", 10.);
49                  move(pcor);
50          comply("fz", 10.);
51                  limit("fy", 15.);
52                  distance("dy", -10.);
53                          move(pcor);
54                  comply("fy", -10.);
55                          update(corner, pcor);
56                          limit("fx", 25.);
57                          distance("dx", 10.);
58                                  move(pcor);
59          lock("fz fy");
60          setvel(50, 50);
61          distance("dx dy dz", -10., 10., -50.);
62                  move(pcor);
63
64          setvel(300, 50);
65          setmod('j');
66          move(park);
67          distance("dx dy dz", -10., 10., -50.);
68                  move(pcor);
69          setmod('c');
70          setvel(50, 50);
71          distance("dx dy dz", -1., 1., -1.);
72                  move(pcor);
73          distance("dx dy dz", -10., 10., -50.);
74                  move(pcor);
75          setvel(300, 50);
76          setmod('j');
77          move(park);
78  }
```

Again the preliminary phase is quite similar to the previous example. The approximate location of the corner is taught by an operator and a chance is also given, line 32 to 43, to cancel the task. The reader may notice that in this example, the corner is oriented in such a way that approaching it corresponds to positive displacements in the X and Z directions, and a negative one in the Y direction. The manipulator approaches the first wall of the corner moving along the Z direction, lines 48 and 49, and enters the first *comply* mode, line 50, before moving to the next wall. The same process is repeated for the Y and Z directions. In each case a preload force is exerted in the appropriate direction in order to maintain contact with the walls. The last accommodation motion, line 58, is associated to a call to **update** so as to record the final position. Two compliance degrees of freedom are accumulated and the manipulator is brought back to position servo mode line 59. The peg is then taken away, lines 60 to 62. Before going back to the recorded position, the arm is moved at high velocity to the PARK position.

8. Structuring Programs

We shall attempt in this section to show how higher level primitives can be written in term of RCCL functions. We shall make use of the macro processing facilities to define in a few lines some manipulator language statements often encountered. A primitive **insert** based on a bare bone version of the insertion task explained earlier is described. This **insert** primitive, newly defined is used in a repetitive task. Each loop the manipulator moves to a 'get' position where a feeder conveys pegs to be inserted on an assembly. The holes locations are stored on file and may have been taught in a previous operation or obtained from a CAD/CAM system. The loop synchronizes with the feeder's actions via an external variable :

```
#include "rccl.h"
#include "umac.h"
#include "hand.h"

#define AWAYZ(p, l)       {distance("dz", - (l)); move(p);}
#define OVERSHOOTZ(p, l)  {distance("dz",   (l)); move(p);}
#define FAST              setvel(300, 300.);
#define SLOW              setvel(50, 50.);
#define CAUTIOUS          setvel(7, 7);

/*
 * do one insertion
 */

insert(z, grip, peg, hole, depth, ang)
TRSF_PTR z, grip, peg, hole;
real depth, ang;
{
        TRSF_PTR bottom, angle, roty;
        POS_PTR align, in, touch;

        bottom = gentr_trsl("BOTTOM", 0., 0., -depth);
        angle = gentr_rot("ANGLE", 0., 0., 0., yunit, ang);
        roty = gentr_rot("ROTY", 0., 0., 0., yunit, 180.);

        align = makeposition(
        "ALIGN", z, t6, grip, peg, EQ, hole, angle, roty, TL, peg);

        touch = makeposition(
        "TOUCH", z, t6, grip, peg, EQ, hole, angle, roty, TL, peg);

        in = makeposition(
        "IN", z, t6, grip, peg, EQ, hole, bottom, roty, TL, peg);

        setmod('c');
        FAST
        AWAYZ(touch, 10.)
        CAUTIOUS
        AWAYZ(touch, 4.)
        limit("fz", 25.);
        OVERSHOOTZ(touch, 5.)
        comply("fz", 15.);
                move(align);
        lock("fz");
        comply("fx fy", 0., 0.);
                update(hole, in);
                limit("fz", 20.);
                OVERSHOOTZ(in, 10.);
        lock("fx fy");

        SLOW
        AWAYZ(align, 50.)
        waitfor(in->end)
        OPEN
        move(there);
        waitfor(completed);
        freepos(align);
        freepos(in);
        freepos(touch);
        freetrans(bottom);
        freetrans(angle);
        freetrans(roty);
        return;
}
```

```
/*
 * monitors feeder
 */

#define PARTS   1
#define EMPTY   2

monfeeder()
{
        if (feedersensor == PARTS) {
                nextmove = YES;
                CLOSE
        }
        if (feedersensor == EMPTY) {
                parts = 0;
        }
}

/*
 * Do insertions
 */

int parts = YES;

pumatask()
{
        TRSF_PTR z, e, assy, h, feeder, grasp, pegs;
        POS_PTR get;

        z = gentr();                                    /* base frame */
        e = gentr();                                    /* end effector */
        assy = gentr();                                 /* assembly */
        grasp = gentr();                                /* gasp pos */
        feeder = gentr();                               /* feeder */
        pegs = gentr();                                 /* peg rel. e */
        h = newtrans("H", hold);                        /* h rel. to assy */

        get = makeposition("GET", z, t6, e, EQ, feeder, grasp, TL, e);

        while(parts) {
                move(get);
                evalfn(monfeeder);
                setime(200, 10000);
                move(get);
                gettr(h, file);
                insert(z, e, pegs, h, 20., 15.);
        }
}
```

Conveyors are expensive, and rugged objects could be thrown from place to place. We shall see here how a 'throw' primitive (seldom found in regular robot programming languages) can be easily written. In order to obtain a maximum acceleration, we shall program a sequence of motions that only uses the transition part. This example is only given as an illustration because the dynamic qualities of the Puma manipulator proved to be not quite sufficient.

```
#include "rccl.h"
#include "hand.h"
#include "umac.h"

real when;                  /* to open the gripper */

#define MAXACC  .015       /* mm/ms2 */

throw(v0)
VECT_PTR v0;
{
        int openat();

        real Tx = (12. * v0->x) / MAXACC;       /* compute          */
        real Ty = (12. * v0->y) / MAXACC;       /* the acceleration */
        real Tz = (12. * v0->z) / MAXACC;       /* times            */
        int  T = ((FABS(Tx) > (Ty))             /* and pick up      */
                        ? ((FABS(Tx) > FABS(Tz))
                                ? Tx : Tz)      /* the longest one  */
                        : ((FABS(Ty) > FABS(Tz))
                                ? Ty : Tz));

        real dx, dy, dz;

        stop(0);
        setmod('c');
        dx = Tx * v0->x / 2.;
        dy = Ty * v0->y / 2.;
        dz = Tz * v0->z / 2.;

        distance("dx dy dz", -dx, -dy, -dz);
        setime(T / 2, T);
        move(there);            /* back up      */

        when = .90;             /* open the gripper just */
        evalfn(openat);         /* before the end        */
        distance("dx dy dz", 2. * dx,  2. * dy,  2. * dz);
        setime(T / 2, T);
        move(there);            /* move as fast as possible */

        setime(T / 2, T);       /* come back             */
        move(there);
        stop(0);
        return;
}


openat() /* opens the gripper at a given moment */
{
        if (goalpos->scal >= when) {
                OPEN
        }
}
```

```
54
55  pumatask()
56  {
57          TRSF_PTR b0, grip;
58          POS_PTR p0;
59          VECT vel;
60          int q;
61
62          grip = gentr_trsl("GRIP", 0., 0., 170.);
63          b0 = gentr_rot("B0", 400., 150., 700., yunit, 45.);
64
65          p0 = makeposition("P0", t6, grip, EQ, b0, TL ,grip);
66
67          QUERY(q)
68          CLOSE
69          setconf("d");   /* elbow down, like the Great Di Maggio */
70          setime(100, 3000);
71          move(p0);                   /* move above the shoulder */
72          vel.x = .0;
73          vel.y = .0;
74          vel.z = .6;                 /* m/s at 45 degrees, see B0 */
75
76          throw(&vel);
77
78          setmod('j');                /* back to park */
79          setconf("u");               /* elbow up     */
80          setime(100, 3000);
81          move(park);
82  }
```

The acceleration times , lines 14 to 21, and the magnitudes, lines 27 to 29, are derived from the coefficients of the quartic polynomial functions used to generate the transitions [2]. The segment times are exactly twice the accelerations times.

9. Limitations

9.1. Force Control

In the case of the Puma manipulator, the implementation of force control suffers a number a limitations due to the simplicity of the implemented method. Force measurements are obtained by monitoring the motor currents. Coulomb friction terms, at the joint level, have been experimently measured [8]. When the velocity of a joint is small or null, these terms are irrelevant and cannot be used to improve the accuracy of the control. When the arm if to stop on force, this is of little importance since the joints likely to provide the guarded motion are moving. Nevertheless, this fact has to be kept in mind. Gravity loadings have also been experimently measured. Experiments have shown that although the mass of an object carried by the manipulator could be measured, the accuracy is not sufficient and is likely to cause offset errors for the gravity loadings. The function **massis** has been implemented to get around this.

Force specifications possess an estimated accuracy of approximately 10 Newton in most of the work space. This is pretty close to the load capabilities of the manipulator, therefore extreme prudence is recommanded. Despite this lack of accuracy, the tasks using force control explained in this document all run with a good reliability.

When the manipulator transitions from *comply* servo mode to *position* servo mode, a glitch often occurs and is as noticeable as the velocity is high and the load important. It is usually harmless, and correspond to the position servo correcting the first setpoint.

Compliance in a given direction is obtained by selecting the joints most suitable to provide the desired effects [2]. The joint selection method is simplified. It does not take into account the translation part the *tool* frame. This means that in *comply* servo mode, force specifications will always match the inner joints (1, 2, 3) and torques specifications the wrist joints (4, 5, 6). Although the method is reliable and simple, it suffers the drawback that no remote center of compliance can be specified. Time constraints have prevented further work on this points, and any contributions are welcome.

9.2. Machine Errors.

When the robot task is running in real time, the process is locked into core memory and the interrupt function of the system as well as the user's background functions are run at very high priority in kernel mode. Any system call (machine trap), will crash the system (beware of the prints). The same problem occurs for any machine error like a bad memory reference of a floating point exception in any part of the process. Some debugging tools are provided as explained later.

9.3. Process Size

When the real time process is run, it is locked into core memory and the virtual memory system desactivated. Therefore, the process cannot grow it's allocated region. Dynamic allocation is performed within a preallocated memory area. The system calls like 'malloc' are replaced by alternative functions [6]. A set a macros :

```
#define malloc  malloc_l
#define free    free_l
#define realloc realloc_l
#define calloc  calloc_l
#define cfree   cfree_l
```

allows the user to safely write :

```
        p = malloc(20);
```

This causes a more annoying problem when it comes to opening files. Files can be opened only when the real time channel is closed. However, the user can always code :

```
...
move(p);
stop(200);
waitfor(completed);
release("opening files");
fd1 = creat(...
fd2 = open(...

startup();
move(...);
```

The process is temporally put back in normal mode by the function **release** [6], and file 'opens' can take place. The function **startup** will resume real time execution by the depressing the ARM POWER button when requested by the system. Failing to follow the procedure will also cause a system crash.

9.4. Sample

The sample period is normally 28 ms. It can be set to 14 via the function **sample** and when not needed, the sample period can be reset to 28 ms. Changing the sample period can cause a slight glitch. If the velocity if the manipulator is small, the glitch is negligible. For example the for loop of the example section 7.3 can be coded :

```
32        for (; ; ) {
33                setvel(400, 300);
34                move(get);
35                move(p1);
                  stop(0);
36                setvel(100, 100);
37                sample(15);
38                move(p2);
                  stop(0);
39                sample(30);
40                printf("more ?"); QUERY(q); if (q == 'n') break;
41        }
```

If the user's background functions take to much time to execute, the behavior of the real time interface no always easy to predict. In the best cases, it causes a crash of the superviser program running in the LSI-11. The arm power is immediately turned off, and nothing annoying happens. The superviser is restarted and everything comes back to normal. It seems that when the user's functions processing time is slightly too long, the VAX still accepts interrupts, but stacks them and this quickly causes a system crash. Finally, if the interrupt code is very long (an infinite loop, for example), the system is totally blocked and a manual boot is necessary.

9.5. Large Rotations.

For a reason that has not been yet determined, some motion transitions involving large rotations do not behave quite correctly.

10. The Planner and Play Program

In order to write and debug the first draft of manipulator programs, a special library is provided. This library has exactly the same entry points as the regular library, but replaces the interrupt code with a loop. Exactly the same programs can by run and tested. The synchronization features are simulated so that everything happens in the same order as in the real time version. The user is advised to run the programs in this mode before actual execution. The resulting sequence of points can be dumped onto file for execution by the *play* program [6]. Trajectories can be also stored and displayed on the terminal by a special program called *dsp*.

When programs with guarded motions are run in this fashion, the conditions will never be met, unless special simulation monitoring functions are written. When programs include comply statements, the comply mode is simulated as follow : the compliant joints are selected according to the geometry of the task and are artificially frozen as if the resulting forces would keep them immobile. The accommodation motions compensation feature being still activated, it may produce funny but meaningful trajectories. Tracking with external information can produce various results according to the situation at hand. Nevertheless it is very useful to test ahead manipulator programs. All branches must be tested because manipulator control is essentially programming with side effects. It is always useful to 'play' the resulting trajectories in free space to get an idea of what is going to happen.

11. Program Options

Programs can be run with a number of options :

v This option allows the user to specify the printing of information. A file called '@@.out' is created in the current directory. It contains informations about what the system understood of the calls to **makeposition.** A record will be printed for each move request. For the planning version only, a record will be printed by the trajectory generator at the time the request is executed, for example the beginning of the file '@@.out' for the camera guided task Section 7.1. is :

```
makeposition, pos       "LOOK"   Z T6 E  = EXPECT
optim, initial equation :        T6 =  -Z EXPECT -E
optim, final equation   :        T6 =  _TEMP1 -E
        "COORD":  "TOOL": -E  "POS": _TEMP1

makeposition, pos       "GET"    T6 E  = COORD CAM O
optim, initial equation :        T6 =  COORD(h) CAM O(h) -E
optim, final equation   :        T6 =  COORD(h) CAM O(h) -E
        "COORD": COORD(h) CAM  "TOOL": -E  "POS": O(h)

makeposition, pos       "PUT"    Z T6 E  = DROP
optim, initial equation :        T6 =  -Z DROP -E
optim, final equation   :        T6 =  _TEMP2 -E
        "COORD":  "TOOL": -E  "POS": _TEMP2


request LOOK mode j acct 56  sgt 0  velt 200 velr 100
conf     upd :  smpl 0 mass 0.000000

PARK -1 28 j 84 84 280 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   00 0 0 0 0 0 0
LOOK -1 336 j 56 56 2660 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   00 0 0 0 0 0 0
```

```
request GET mode j acct 56  sgt 0  velt 200 velr 100
conf    upd :  smpl 0 mass 0.000000
dist dz : -30

request GET mode j acct 56  sgt 0  velt 200 velr 100
conf    upd :  smpl 0 mass 0.000000

request STOP mode j acct 0  sgt 28  velt 200 velr 100
conf    upd :  smpl 0 mass 0.000000

request GET mode j acct 56  sgt 0  velt 200 velr 100
conf    upd :  smpl 0 mass 0.000000
dist dz : -30

GET -1 3024 j  56 56 1568 28
          force 00 0 0 0 0 0 0
          cply  00 0 0 0 0 0 0
          dst   00 0 0 0 0 0 0
          exd   04 0 0 -30 0 0 0
GET -1 4592 j  56 56 280 28
          force 00 0 0 0 0 0 0
          cply  00 0 0 0 0 0 0
          dst   00 0 0 0 0 0 0
          exd   00 0 0 0 0 0 0
GET -1 4872 j  56 56 140 28
          force 00 0 0 0 0 0 0
          cply  00 0 0 0 0 0 0
          dst   00 0 0 0 0 0 0
          exd   00 0 0 0 0 0 0
```

```
request PUT mode j acct 56  sgt 0  velt 200 velr 100
conf     upd :   smpl 0 mass 0.000000

GET -1 5012 j 56 56 280 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   04 0 0 -30 0 0 0
PUT -1 5292 j 56 56 2492 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   00 0 0 0 0 0 0
PUT -1 7784 j 56 56 112 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   00 0 0 0 0 0 0
request LOOK mode j acct 56  sgt 0  velt 200 velr 100
conf     upd :   smpl 0 mass 0.000000

Etc ...
```

The equations are printed, then their canonized form before and after optimization. Transforms are marked according to their type : varb (v), hold (h), functional (s). The optimization premultiplications generate the '_TEMPx' names. For each request all the parameters are printed, for example :

```
request LOOK mode j acct 56  sgt 0  velt 200 velr 100
conf     upd :   smpl 0 mass 0.000000
```

means : position 'LOOK', mode 'joint', acceleration time 56 ms, segment time is 0 that is : will be computed at execution time, current velocities are 200 mm/s and 100 deg/s, no configuration change required, no transform to update, no sample time change, current mass of object is 0. kg.

The trajectory generator prints in a compact format the specification at the beginning of each motion (planning version only) :

```
GET -1 5012 j 56 56 280 28
        force 00 0 0 0 0 0 0
        cply  00 0 0 0 0 0 0
        dst   00 0 0 0 0 0 0
        exd   04 0 0 -30 0 0 0
```

means : previous motion terminated 'OK' (-1), time is 5012 ms, mode is 'joint', accelerations times first transition is 56, second 56, segment time is 280, time increment is 28. No force limit, no comply, no differential motion stop, distance is -30 mm in Z direction. For the records 'force', 'cpy', 'dst', and 'exd', the first number is an octal code (00 for no specification, translation or forces : 01 for X, 02 for Y, 04 for Z, rotations or torques : 10 for X, 20 for Y, 40 for Z, and the combinations : 01, 03, 05, 06, etc ...);

If the the option '-vv' as very verbose is given, the values of the transforms created by the 'gentr...' style function is also printed.

This option corresponds to the global flag **prints_out**. This flag can be turned on or off the text of the programs themselves :

```
        . . .
        prints_out = YES;
        p0 = makeposition(...);
        p1 = makeposition(...);

        move(p0);
        move(p1);
        prints_out = NO;
```

The information is printed to the **fpi** file pointer :

```
        FILE *fpi;
```

This file pointer is initialized to the 'stderr' file pointer. When the flag **prints_out** is set to a non zero value, the **makeposition** and **move** messages go to the terminal. When the option '-v' is specified, the file '@@.out' is created and **fpi** points to it, and the messages are stored on the file. One can use this feature for any purposes, for example :

```
pumatask()
{
        TRSF_PTR ...
        POS_PTR ...
        . . .

        prints_out = NO;
        p = makeposition(...);
        . . .

        move(p);
        . . .
        fprintf(fpi, "bla bla");
        . . .
}
```

If the task is run without option '-v', "bla bla" goes on 'stderr' file, if the task is run with option '-v', "bla bla" goes into '@@.out'.

g This is the 'graphic' option (planing version only). The setpoints are stored in the files '../g/f1.out', '../g/f2.out', one for each joint. When displayed with the program *dsp* a character 'J' stands for joint mode straight part, 'T' for joint mode transition, 'E' for first point of joint mode transition, 'C' for a Cartesian mode straight part, 'H' for Cartesian transition, and 'V' for first point of Cartesian transition. In order to use this option, the user is required to have a 'graphic' directory '../g' at the same level in the file tree hierarchy as the current directory. This will avoid having the current directory constantly clustered with junk files.

d This is the 'data' option (planning version only), when specified, the system creates the file '@.out' in the current directory that will contain one line per setpoint according to the following format :

```
      POS M  time tseg j1  j2  j3  j4  j5  j6 sel
```

Where 'POS' is the name of the goal position, 'M' is the mode (J, T, E, C ,V ,H) as described above, 'j1'...'j6' are the joint angles in range coordinate [6], and 'sel' an octal value showing which joint are complying in comply mode (0 no joint, 01 for joint 1 , 02 for joint 2, 04 for joint 3, 10 for joint 4, 20 for joint 5, 40 for joint 6, 3 for joint 1 and 2, etc.).

a This option when set, causes the joint angles to be output in solution coordinates [6]. It serves for option 'd' and 'g'.

k This option when set, causes the values of *T6* to be printed in lieu of the joint angles. For the option 'g' twelve files (f1.out ... f12.out) are created, the values of the vectors 'p', 'n', 'o', and 'a'; For the option 'd' the format becomes :

```
POS M  time tseg px  py  pz  nx  ny  nz ox oy oz ax ay az
```

It serves for option 'd' and 'g'.

e This option causes the file '@@@.out' to be created in the current directory (planing version only). The file contains the sequence of encoder values suitable to be used by the *play* program [6].

Dname This option specifies the file 'name' as a data base of transforms. Can be used in association with the teach mode (see below).

This option corresponds to the global file descriptor **fddb** initialized to '-1'. When the option '-Dname' is specified, **fddb** describes the file 'name'. If the file 'name' does not exit the user is prompt as :

```
name does'nt exit, create ? (y/n)
```

Answer as appropriate.

b This option turns off the force control features (brute option). In the case of the planning version, no simulated joint accommodation will occur. In the case of the real time version, it allows us to test the manipulator programs free of obstacles.

This option corresponds to the global flag **force_ctl** which is turn off by the '-b' option. The flag can be turned on or off in the text of the programs.

12. Teaching

The *teach* mode is activated by a call to the **teach** function :

```
        teach(trans, pos)
        TRSF_PTR trans;
        POS_PTR pos;
```

The **teach** function gives control to the user on the manipulator motions. When the teach mode begins, the following message appears on the terminal :

```
teach mode V1.0, transform TRANS, position POS
?
```

a simple command line language allows the user to move the manipulator around. When the desired position is obtained, the transform 'TRANS' can be solved for the position equation 'POS' for the current value of *T6*. This is obtained on user's command by a call to **update**. Once the position is recorded, the manipulator can be moved elsewhere. Upon exit of the *teach* mode, if no position have been recorded the user is prompted as :

```
nothing taught, exit ? (y/n)
```

If 'n' is answered, the *teach* mode is not exited, if 'y' is answered the *teach* mode exits and the **teach** function return the value 'NO'. When a transform has been recorded, upon exit the function **teach** returns the value 'YES'. Even if a transform has been recorded, **teach** can be forced to return 'NO' by typing a 'q!'. Applications of this have been shown in section 7. If successive records are made, only the last one is taken into account.

When a data base file has been specified, the *teach* mode behaves differently. The transform to be taught is searched in the data base under its name, if found, the function **teach** directly uses the value and immediately exits returning 'YES', **update** is then not called. If the transform cannot be found in the data base, **teach** enter the regular manual mode. The user can record the transform value and save it on the data base. If no data base has been specified the user is informed of that fact. A data base editor (see below) can be used for off-line maintenance.

The interactive commands are displayed when a '?' mark is typed. By convention, the lower case 'x', 'y', 'z' characters stand for translations or forces, and the upper case 'X', 'Y', 'Z' stand for rotations of moments :

```
These commands are executed one per line:
        <return>          interrupt arm motion
        q                 quit
        q!                quit, ignore not recorded
        r                 record transform
        p                 display current settings
        s                 save transform on data base
        l                 toggle force monitor
      v <t r>           set velocities
      m <m>             set mass of object
        ?                 this message
These commands cumulate:
        o                            open hand
        c                            close hand
      w <x/y/z/X/Y/Z> <k>        move world coordinates
      t <x/y/z/X/Y/Z> <k>        move tool coordinates
      e <x/y/z/X/Y/Z> <k>        change tool transform
      f <x/y/z/X/Y/Z> <k>        set force limits
```

Messages from the system can be :

```
no data base specified

nothing recorded

stopped on force

>> stopped

next to limit(s)

not so fast
```

A teach session can be obtained by running the program :

```
#include "rccl.h"
#include "umac.h"


pumatask()
{
        TRSF_PTR z, e , b0;
        POS_PTR  p0;

      z = gentr_trsl("Z",  0.,  0., 864.);
      e = gentr_trsl("E" , 0. , 0. , 170.);
        b0 = gentr_rot("B0", 600. , -200., 800., yunit, 180.);
        b0->fn = varb;

        p0 = makeposition("P0" , z, t6, e, EQ, b0, TL, e);

        setmod('c');
        setvel(300, 100);
        move(p0);
        while (teach(b0, p0))
                ;
        setvel(300, 100);
        move(park);
}
```

The session can look like :

```
$ a.out -Ddata
data does'nt exit, create ? (y/n) y
gettr : B0 not found
teach mode V1.0, transform B0, position P0
?p
T6:
  -1.000   -0.000   -0.000  600.001
  -0.000    1.000    0.000 -200.000
   0.000    0.000   -1.000  106.000
E:
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000  170.000
veloc t:100 r:10
____1_______ _______2____ _________3__ ____4_______ _________5__ ____6_______
no force limit
mass of object : 0.000000 kg
?v 30 7
?wx200 wz -300 wY10
?
>> stopped
not so fast
?wz200
?l
?p
T6:
  -0.985    0.000   -0.171  826.436
  -0.000    1.000    0.000 -200.000
   0.171    0.000   -0.985    6.044
E:
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000  170.000
veloc t:30 r:7
____1_______ ________2___ ______3_____ ___4________ __________5_ ____6_______
force limits :fx 0.00   fy 0.00   fz 0.00    fX 0.00   fY 0.00   fZ 0.00
mass of object : 0.000000 kg
?fx20 fY5
?wz-30
stopped on force
?p
T6:
  -0.985    0.000   -0.171  826.436
  -0.000    1.000    0.000 -200.000
   0.171    0.000   -0.985   16.013
E:
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000  170.000
veloc t:30 r:7
____1_______ ________2___ ______3_____ ___4________ __________5_ ____6_______
force limits :fx 20.00  fy 0.00   fz 0.00    fX 0.00   fY 5.00   fZ 0.00
mass of object : 0.000000 kg
```

The *teach* mode uses its own position equation to move the arm around. The tool transform is preset to a 170 mm translation in the Z direction, but can be changed. The messages "not so fast" or "next to limit(s)" do not appear when the condition occurs, but when the next command is typed. The 'p' command prints the current values of $T6$, E, velocities, the relative position of the joints in their range, the current force limits when toggled on, and the current mass of object.

13. Summary

13.1. Error Messages

An RCCL internal error, causes a message to be printed and an exit of the process for the planning version. When run in real time mode, the process does not exit but the arm power is turned off and the process is put to sleep, this is to allow the user to 'break' the program and take advantage of the automatic home position return [6]. If the error occur at the level of the real time interface, we refer the reader to [6] for a determination of the error. If the error is a RCCL error condition, the messages can be :

"position "POS" : transform not initialized - makeposition" : one of the transform pointer is 'NULL'.

"position "POS" : missing t6 or tool - makeposition" : bad position equations structure.

"position "POS" : missing rhs - makeposition" : bad position equation structure.

"position "POS", transform "TRANS" : pos functionally defined - makeposition" : the *POS* part of the canonized equation cannot be a moving frame.

"position "POS" : pos cannot seriously be t6 - makeposition" : the *POS* part is equal to *T6* due to a bad choice of the *TOOL* part.

"giveup" : The function **giveup** has been called, a message follows.

"bad spec. - limit" : wrong directions specifications.

"bad force spec. - comply" : wrong directions specifications.

"bad force spec. - lock" : ditto.

"bad distance spec. - distance" : ditto.

"conf must change in joint mode" : the current motion mode is not correct for a configuration change.

"invalid update transform type" : the involved transform must be of type *varb*.

"could'nt find updatable transform" : a transform has been required to be updated but does not belong to the specified equation.

"alloc err" : motion queue saturated.

"mem. alloc error" : no more dynamic memory allocation space.

"limit 'time'" : a joint limit occurred at time 'time', the program does not exit and tries to recover by stopping and getting a new motion from the queue. (planning version only).

"joint(s) limit" : an unrecoverable joint limit occurred. (planning version only).

"glitch 'time'" : a velocity discontinuity occurred at time 'time' (planning version only).

"jam" : unexpected behavior of the queue management, should never occur.

"cannot unit vector" : the unit function has been required to unit a zero magnitude vector.

"Write io error", "write io error", "close io error" : an i/o error occurred while writing data (planning version only).

"*** could'nt queue at 'time'", this message may occasionally appear, but it never did so far.

Note

The user can use the function **giveup** to cause a task cancellation :

```
giveup(message, level);
```

The first argument is a string, the second argument tells if the error condition occur in a background function (level 0) or in the user process (level 1), for example :

```
pumatask()
{
        ...

        evalfn(monitor);
        move(p);
        while (goalpos == p) {
                ...
                if (big_mess) {
                        giveup("cannot do that", 1);
                }
        }
}

monitor()
{
        ...
        if (not_good) {
                giveup("wrong data", 0);
        }
}
```

The error message would be :

```
cannot do that
giveup

or

wrong data
giveup
```

13.2. Functions, Global Variables, and Macros

Follows a brief description of the RCCL function library :

Dictionary of the terms

ax : x element of 'a' vector of a transform (real).

ay : y element of 'a' vector of a transform (real).

az : z element of 'a' vector of a transform (real).

bool : an integer expression evaluating to 0 or non zero.

code : an integer expression (OK LIMIT ONF OND predefined).

conf : a string at most one of the 'l' 'r' 'u' 'd' 'f' 'n' characters and ' '.

diff : DIFF_PTR, a pointer to a DIFF structure.

dirs : a string of the form "fx ty", "tz", ..., for force and distance specs.

eve : an event count.

force : FORCE_PTR, a pointer to a FORCE structure.

fp : a UNIX file pointer *FILE (stdout, stderr ...).

func : pointer to a function.

level : an integer expression evaluating to 0 (interrupt) or 1 (user).

list : a list of transform pointers (TRSF_PTR) separated by commas.

msg : a string.

mode : the character 'j' or 'c'.

name : a string.

ox : x element of 'o' vector of a transform (real).

oy : y element of 'o' vector of a transform (real).

oz : z element of 'o' vector of a transform (real).

period : an integer expression in milliseconds.

phi : an angle in degrees (real).

pos : a pointer to a position structure (POS_PTR).

pphi : a pointer to a angle in degrees (*real).

ppsi : a pointer to a angle in degrees (*real).

psi : an angle in degrees (real).

pthe : a pointer to a angle in degrees (*real).

px : x element of 'p' vector of a transform in millimeters (real).

py : y element of 'p' vector of a transform in millimeters (real).

pz : z element of 'p' vector of a transform in millimeters (real).

rotvel : a rotational velocity in degrees per second (int).

tacc : an acceleration time in milliseconds (int).

the : an angle in degrees (real).

time : a time in milliseconds (int).

trans : a pointer to a transform structure (TRSF_PTR).

transvel : a translational velocity in millimeters per second (int).

values : a list of specifications in millimeters, degrees, Newtons, or Newton-meters.

vect : pointer to a vector structure (VECT_PTR).

Description of Functions, Variables, and Macros

Note : if p is pointer, *p is what is pointed to. functions names are marked 'f', variables names 'v', macros names 'm'.

f **assigndiff(diff1, diff2)** : copy *diff2 into *diff1, return diff1.

f **assignforce(force1, force2)** : copy *force2 into *force1, return force1.

f **assigntr(trans1, trans2)** : copy *trans2 into *trans1, return trans1.

f **assignvect(vect1, vect2)** : copy *vect2 into *vect1, return vect1.

v **completed** : signaled when motion queue goes empty and the arm is evaluating last position (event).

f **comply(dirs, values)** : specify compliance for subsequent requests.

f **const()** : does nothing but typifies a transform as constant (TRFN).

f **cross(vect1, vect2, vect3)** : compute in *vect1 cross product of *vect2 and *vect3, return vect1.

f **df_to_tr(trans, diff)** : builds differential transform *trans out of differential motion *diff, return trans.

v **dgtord_m** : read only (real), convert from degrees to radians what is multiplied by.

f **difftr(diff1, diff2, trans)** : transforms differential motion *diff2 into differential motion *diff1, with a frame differential relationship *trans, return diff1.

f **distance(dirs, values)** : internally changes the position expressed in *tool* frame.

f **dot(vect1, vect2)** : return (real) the dot product of *vect1 and *vect2.

f **eul(trans, phi, theta, psi)** : set the rotational part of *trans from Euler angles, return trans.

f **eulm(trans, phi, the, psi)** : multiplies *trans, by a rotation expressed with Euler angles, returns trans.

f **evalfn(func)** : causes the function *func to be evaluated for next motion request.

v **force_ctl** : turns on/off force control features (bool).

f **forcetr(force1, force2, trans)** : transform generalized forces *force2 into generalized forces *force1, with a frame differential relationship *trans, return force1.

v **fpi** : information file pointer (*FILE).

f **freepos(pos)** : returns to the memory pool the storage allocated for building a positions equation ring structure.

f **gensym()** : return a pointer to an always different string (_TEMP1, _TEMP2, ...).

f **gentr_eul(name, px, py, pz, phi, theta, psi)** : make a constant transforms out of a 'p' vector and Euler angles, return a trans.

f **gentr_pao(name, px, py, pz, ax, ay, az, ox, oy, oz)** : make a constant transforms out of a 'p' vector and 'a', 'o' vectors, return a trans.

f **gentr_rot(name, px, py, pz, vect, theta)** : make a constant transform out of a 'p' vector and a rotation of theta degrees around *vect, return a trans.

f **gentr_rpy(name, px, py, pz, phi, theta, psi)** : make a constant transforms out of a 'p' vector and roll, pitch, yaw angles. return a trans.

f **gentr_trsl(name, px, py, pz)** : make a constant transforms out of a 'p' vector and a unit rotation, return a trans.

f **giveup(msg, level)** : cancel a task, and print msg when broken.

v **goalpos** : a read only (POS_PTR), equal to the position pointer of the equation currently evaluated.

v **hdpos** : a write only (short), hand position information.

v **here** : a read only (TRSF_PTR), equal to $T6$ at segment termination.

f **hold()** : does nothing but typifies a transform as to be held (TRFN).

f **invert(trans1, trans2)** : store in *trans1 the inverse of *trans2, trans1 and trans2 different, return trans1.

f **invertinp(trans)** : stores in *trans the inverse of *trans, return trans.

v **j6** : a read only (JNTS_PTR), the current desired joint setpoint in range coordinates.

v **jd** : a read only (JNTS_PTR), the desired differential joint setpoint.

v **lastpos** : a read only (POS_PTR), equal to the position pointer of the last evaluated equation.

f **limit(dirs, values)** : trigger force or differential motion monitoring for the next motion request.

f	**lock(dirs) :** bring back the arm in position servo mode for the specified directions.
f	**makeposition(name, list, EQ, list, TL, trans) :** build a position equation ring structure, returns a pos.
f	**move(pos) :** enter a motion request toward a position described by pos in the motion queue.
m	**movecart(pos, tacc, time) :** do setmod('c'); setime(tacc, time); move(p).
m	**moveconf(pos, tacc, time, conf) :** do setconf(conf); setmod('j'); setime(tacc, time); move(p).
m	**movejnts(pos, tacc, time) :** do setmod('j'); setime(tacc, time); move(p).
f	**newtrans(name, func) :** allocate storage for a *trans, attach it to function *func, return a trans.
v	**nextmove :** a write only code, when set, causes the current motion interruption and the value returned in the corresponding position structure field 'code'.
f	**noatoeul(pphi, pthe, ppsi, trans) :** derive the Euler angles from *trans.
f	**noatorpy(pphi, pthe, ppsi, trans) :** derive the roll pitch yaw angles from *trans.
f	**optimize(pos) :** optimize a position equation ring structure.
v	**park :** a read only (POS_PTR), the park position.
v	**pi_m :** a read only approximation of the number pi (real).
v	**pib2_m :** a read only approximation of the number pi/2 (real).
v	**pit2_m :** a read only an approximation of the number pi*2 (real).
f	**printd(diff, fp) :** print *diff on file *fp.
f	**printe(trans, fp) :** print *trans on file *fp (Euler angles).
f	**printm(force, fp) :** print *force on file *fp.
f	**printr(trans, fp) :** print *trans on file *fp (n o a p).
f	**printrn(trans, fp) :** printf *trans on file *fp (name, n o a p, Euler, rpy).
v	**prints_out :** causes prints when set (bool).

f	**printy(trans, fp)** : print *trans on file *fp (Euler angles).
v	**rdtodg_m** : a read only (real), convert from radians to degrees what is multiplied by.
f	**release(msg)** : closes real time channel.
f	**requestnb** : read only (int), the number of not served motion requests.
v	**rest** : a read only (TRSF_PTR), T6 at the park position.
f	**rot(trans, vect, theta)** : set the rotation part of *trans from a rotation around *vect, of angle theta, returns trans.
f	**rotm(trans, vect, theta)** : multiplies *trans by a rotations made out of a rotation around *vect, of angle theta, return trans.
f	**rpy(trans, phi, the, psi)** : set the rotation part of *trans from a rotations of roll pitch an yaw angles, return trans.
f	**rpym(trans, phi, the, psi)** : multiplies *trans by a rotation of roll pitch and yaw angles, return trans.
v	**rtime** : an (int), the time spend since the last reset, in milliseconds.
f	**sample(period)** : change the sample rate, next motion request.
f	**setconf(conf)** : change the arm configuration next motion request.
f	**setime(tacc, time)** : set the acceleration and segment time next motion request.
f	**setmod(mode)** : set the motion mode, next motion request.
f	**setvel(transvel, rotvel)** : set the translational an rotational velocities, next motion request.
f	**startup()** : start real time channel.
f	**stop(time)** : repeat last motion request, during time.
f	**strsave(string)** : copies string in allocated storage and return pointer to it.
f	**suspendfg()** : put foreground process to sleep for 1/10 of a second.
f	**takerot(trans1, trans2)** : copy 'n' 'o' 'a' vectors of *trans2 into *trans1, return trans1.

f **taketrsl(trans1, trans2)** : copy 'p' vector of *trans2 into *trans1, return trans1.

v **t6** : read only (TRSF_PTR), the current desired value of T6.

f **teach(trans, pos)** : enters manual teach mode, may update *trans, using pos, return user's exit style.

v **there** : a (POS_PTR) such as move(there) stops the arm.

v **timeincrement** : a read only (int), the current sample time.

f **tr_to_df(diff, trans)** : make *diff out a differential transform *trans, returns diff.

f **trmult(trans1, trans2, trans3)** : multiply *trans2 by *trans3, and store the result in distinct *trans1, return trans1.

f **trmultinp(trans1, trans2)** : multiply *trans1 by *trans2, and store the result in *trans1, return trans1.

f **trmultinv(trans1, trans2)** : multiply *trans1 by inverse of *trans2, and store the result in *trans1, return trans1.

f **trsl(trans, px, py, pz)** : sets the translation part of *trans from p vector, return trans.

f **trslm(trans, px, py, pz)** : multiply *trans by a translation from p vector, return trans.

f **unit(vect1, vect2)** : store in *vect1, the unit magnitude vector, collinear with vect2, return vect1.

v **unitr** : a read only (TRSF_PTR), the unit transform.

f **update(trans, pos)** : solve *trans in equation *pos, for the value of T6 at the end of the execution of the subsequent motion request.

f **vao(trans, ax, ay, az, ox, oy, oz)** : set rotation part of *trans from elements of non necessarily orthogonal vectors, return trans.

f **vaom(trans, ax, ay, az, ox, oy, oz)** : multiply *trans by a rotation from elements of non necessarily orthogonal vectors, return trans.

f **varb()** : does nothing but typifies a transform as to be variable (TRFN).

m **waitas(bool)** : evaluates bool every 1/10 of a second and proceed if exp is not 0.

m **waitfor(eve)** : increment eve, test eve every 1/10 of a second, proceed if eve drops to 0.

v **xunit :** a read only (VECT_PTR), the X unit vector.

v **yunit :** a read only (VECT_PTR), the Y unit vector.

v **zunit :** a read only (VECT_PTR), the Z unit vector.

13.3. Undocumeted Library Entry Points

The following list is a set of undocumented entry points of the basic RCCL library that may cause link conflicts. The labels always end with a recognizable suffix. The user must keep in mind that the entry points of the *real* time control library are still available, but should normally be used only for reading analog to digital conversions, for example.

Functions :

```
assignjs_n
checkstate_n
dequeue_n
diffjnts_n
drivefn_n
enqueue_n
focpyc_n
fojnts_n
fopar_n
getobsj_n
getobst_n
gravload_n
jacobD_n
jacobI_n
jacobT_n
jns_to_tr_n
jnsend_n
newposition_n
newterm_n
polycpyc_n
polyjnts_n
polypar_n
select_n
setpar_n
setpoint_n
shifttr_n
solveconf_n
solved_n
solvedo_n
solvei_n
solveio_n
t2jnts_n
t2par_n
tr_to_jns_n
```

Variables :

```
armk_c
iobf_n
motionreq_n
mqueue_n
opsw_n
sncs_d
```

13.4. Include Files

rccl.h

This file includes all the necessary ingredients for writing programs that will link with the RCCL library :

rccl.h

```
#include <stdio.h>                    /* included here for safety              */
#include <math.h>                     /* ..................................... */

#define YES       1
#define NO        0
#define UNDEF     2

#define OK        -1                  /* normal path segment termination code */
#define LIMIT     -2                  /* ran into a limit, arm stopped         */
#define ONF       -3                  /* terminated on force                   */
#define OND       -4                  /* terminated on differential motion    */

#define PIB2              1.57079632679489660     /* pi / 2              */
#define PI                3.14159265358979320     /* pi                  */
#define PIT2              6.28318530717958650     /* pi * 2              */
#define RADTODEG          57.29577951308232100    /* 180 / pi            */
#define DEGTORAD          0.01745329251994330     /* pi / 180            */
#define SMALL             (1.e-5)                 /* considered as small */
#define EQ                1                       /* lhs = rhs           */
#define TL                2                       /* tool =              */

#define malloc   malloc_l          /* ..................................... */
#define free     free_l
#define realloc  realloc_l         /* replace dynamic allocation entries    */
#define calloc   calloc_l
#define cfree    cfree_l           /* ..................................... */
```

rccl.h

```
/*
 * RCCL typedefs
 */

typedef int bool;

typedef float real;

typedef int event;

typedef struct vector {
                real x, y, z;
} VECT, *VECT_PTR;

typedef int(* TRFN)();

typedef struct transform {
                char *name;
                TRFN fn;
                VECT n, o, a, p;
                int timeval;
} TRSF, *TRSF_PTR;

typedef struct jns {
                char *conf;
                real th1, th2, th3, th4, th5, th6;
} JNTS, *JNTS_PTR;

typedef struct posit {
                char *name;
                int code;
                real scal;
                event end;
} POS, *POS_PTR;

typedef struct force {
                VECT f, m;
} FORCE, *FORCE_PTR;

typedef struct diff {
                VECT t, r;
} DIFF, *DIFF_PTR;
```

rccl.h

```
/*
 * RCCL functions
 */

extern POS_PTR  makeposition();

extern TRSF_PTR newtrans(),
                gentr_rot(),
                gentr_eul(),
                gentr_rpy(),
                gentr_pao(),
                gentr_trsl(),
                assigntr(),
                taketrsl(),
                takerot(),
                trmult(),
                trmultinp(),
                trmultinv(),
                invert(),
                invertinp(),
                trsl(),
                vao(),
                rot(),
                eul(),
                rpy(),
                trslm(),
                vaom(),
                rotm(),
                eulm(),
                rpym(),
                df_to_tr();

extern DIFF_PTR assigndiff(),
                tr_to_df(),
                difftr();

extern FORCE_PTR assignforce(),
                 forcetr();

extern VECT_PTR assignvect(),
                cross(),
                unit();

extern real     dot();
```

rccl.h

```
extern int      const(),
                hold(),
                varb(),
                optimize(),
                printd(),
                printm(),
                freepos(),
                startup(),
                suspendfg(),
                giveup(),
                release(),
                setmod(),
                setime(),
                setvel(),
                evalfn(),
                setconf(),
                update(),
                sample(),
                massis(),
                limit(),
                comply(),
                lock(),
                distance(),
                move(),
                stop(),
                noatoeul(),
                noatorpy(),
                printr(),
                printrn(),
                printe(),
                printy(),
                teach();
```

rccl.h

```
/*
 * variables
 */

extern JNTS_PTR j6,                          /* current joint range values    */
                jd;                          /* current joint increments      */

extern TRSF_PTR t6,                          /* current T6                    */
                here,                        /* equals T6 each end of segment*/
                rest,                        /* T6 park position              */
                unitr;                       /* unit transform                */

extern VECT_PTR xunit,                       /* X unit vector                 */
                yunit,                       /* Y unit vector                 */
                zunit;                       /* Z unit vector                 */

extern  POS_PTR lastpos,                     /* last evaluated position       */
                goalpos,                     /* current evaluated position    */
                there,                       /* such as t6 = here             */
                park;                        /* such as t6 = rest             */

extern event    completed;                   /* queue empty                   */

extern  FILE    *fpi;                        /* info file pointer            */

extern  bool    prints_out,                  /* info prints switch            */
                force_ctl;                   /* force control switch          */

extern  int     fddb;                        /* data base file descriptor    */

extern  int     rtime,                       /* current time since reset      */
                timeincrement,               /* current sample period         */
                requestnb,                   /* number of requests in queue   */
                nextmove,                    /* motion interruption flag     */
                terminate;                   /* in rtc                        */

extern real     pi_m,                        /* math constants                */
                pib2_m,
                pit2_m,
                dgtord_m,
                rdtodg_m;

extern short    hdpos;                /* hand control information              */

#define waitas(predicate)             {while(!(predicate)) suspendfg();}

#define waitfor(event)                {++(event);\
                                       while(event > 0) suspendfg();}
```

rccl.h

```
#define Assigntr            (void)assigntr
#define Taketrsl            (void)taketrsl
#define Takerot             (void)takerot
#define Trmult              (void)trmult
#define Trmultinp           (void)trmultinp
#define Trmultinv           (void)trmultinv
#define Invert              (void)invert
#define Invertinp           (void)invertinp
#define Trsl                (void)trsl
#define Vao                 (void)vao
#define Rot                 (void)rot
#define Eul                 (void)eul
#define Rpy                 (void)rpy
#define Trslm               (void)trslm
#define Vaom                (void)vaom
#define Rotm                (void)rotm
#define Eulm                (void)eulm
#define Rpym                (void)rpym

#define Assigndiff          (void)assigndiff
#define Df_to_tr            (void)df_to_tr
#define Tr_to_df            (void)tr_to_df
#define Assignforce         (void)assignforce
#define Forcetr             (void)forcetr
#define Difftr              (void)difftr

#define Assignvect          (void)assignvect
#define Cross               (void)cross
#define Unit                (void)unit

#define movecart(p, ta, ts)       {setmod('c'); setime(ta, ts); move(p);}
#define movejnts(p, ta, ts)       {setmod('j'); setime(ta, ts); move(p);}
#define moveconf(p, ta, ts, cf)    {setconf(cf); setmod('j'); setime(ta, ts); move

#define freetrans(t)              {free((char *)t); t = NULL;}
#define freeposition(p)           {freepos(p); p = NULL;}
```

kine.h

This file describes the items related to the kinematics of the considered manipulator. That is why, if you are using the Puma 600, the name 'PUMA' must be #defined somehow. The macros updates the jacobian coefficients, they can be ignored and are listed here for completeness only. The external entries may be of some importance.

kine.h

```
#ifdef PUMA

#define ELBOW_DEG       01              /* elbow degeneracy            */
#define ALIGN_DEG       02              /* T6 in X Z Jt 1 plan         */
#define WRIST_DEG       03              /* wrist degeneracy            */

typedef struct kindyn {
                real a2, a3, d3, d4, d32, e432, aa3d4, e4aa4ad;
                real cp21, cp31, cp32, cp50;
} KINDYN, *KINDYN_PTR;

typedef struct sincos {
        real c1, s1, c2, s2, c23, s23, c3, s3, c4, s4, c5, s5, c6, s6;
        real d1x, d1y, d1z, r1x, r1z, d2x, d2y, d2z, d3x, d3y, d3z;
        real h;
        TRSF u5;
} SNCS, *SNCS_PTR;
```

kine.h

```
/*
 * Macro updates coef of Jacob from the sin cos
 */

#define GETH\
{\
        sncs_d.h = sncs_d.c2 * armk_c.a2 +\
                   sncs_d.s23 * armk_c.d4 +\
                   sncs_d.c23 * armk_c.a3;\
}
#define UPDJ\
{\
        sncs_d.d1x = sncs_d.h * sncs_d.s4 -\
                     armk_c.d3 * sncs_d.c23 * sncs_d.c4;\
        sncs_d.d1y = sncs_d.s23 * armk_c.d3;\
        sncs_d.d1z = sncs_d.h * sncs_d.c4 + armk_c.d3 * sncs_d.c23 * sncs_d.s4;
        sncs_d.r1x = -sncs_d.s23 * sncs_d.c4;\
        sncs_d.r1z = sncs_d.s23 * sncs_d.s4;\
        sncs_d.d2x = armk_c.a2 * sncs_d.s3 * sncs_d.c4;\
        sncs_d.d2y = armk_c.a2 * sncs_d.c3;\
        sncs_d.d2z = -armk_c.a2 * sncs_d.s3 * sncs_d.s4;\
        sncs_d.d3x = sncs_d.c4 * armk_c.d4;\
        sncs_d.d3y = armk_c.a3;\
        sncs_d.d3z = -sncs_d.s4 * armk_c.d4;\
}
#define GETU5\
{\
        sncs_d.u5.n.x = sncs_d.c5 * sncs_d.c6;\
        sncs_d.u5.n.y = sncs_d.s5 * sncs_d.c6;\
        sncs_d.u5.n.z = sncs_d.s6;\
        sncs_d.u5.o.x= -sncs_d.c5 * sncs_d.s6;\
        sncs_d.u5.o.y= -sncs_d.s5 * sncs_d.s6;\
        sncs_d.u5.o.z = sncs_d.c6;\
        sncs_d.u5.a.x = sncs_d.s5;\
        sncs_d.u5.a.y= -sncs_d.c5;\
        sncs_d.u5.a.z = 0.;\
}
#endif

#ifdef STAN
typedef struct kindyn {
        real d2, d22;
} KINDYN, *KINDYN_PTR;

typedef struct sincos {
        real c1, s1, c2, s2, d3, c4, s4, c5, s5, c6, s6;
        real d1x, d1y, d1z, r1x, r1y, r1z, d2x, d2y, d2z, r2x, r2y, r2z,
                d3x, d3y, d3z, r4x, r4y;
} SNCS, *SNCS_PTR;
```

kine.h

```
#define UPDJ\
{\
        real\
        k1 = sncs_d.c4 * sncs_d.c5,\
        k2 = sncs_d.s4 * sncs_d.c5,\
        k3 = sncs_d.c4 * sncs_d.s5,\
        k4 = sncs_d.s2 * sncs_d.d3,\
        k5 = k1      * sncs_d.c6,\
        k6 = sncs_d.s4 * sncs_d.c6,\
        k7 = k5 - sncs_d.s4 * sncs_d.s6,\
        k8 = k2 * sncs_d.c6 + sncs_d.c4 * sncs_d.s6,\
        k9 = k1 * sncs_d.s6 + k6,\
        k10= - k2 * sncs_d.s6 + sncs_d.c4 * sncs_d.c6,\
        k11= k5 + k6,\
        k12= sncs_d.s4 * sncs_d.s5,\
        k13= sncs_d.s5 * sncs_d.c6,\
        k14= sncs_d.s5 * sncs_d.s6;\
        sncs_d.d1x = (-armk_c.d2 * (sncs_d.c2 * k7 - sncs_d.s2 * k13) +\
                      k4 * k8);\
        sncs_d.d2x = sncs_d.d3 * k7;\
        sncs_d.d3x = -k13;\
        sncs_d.d1y = (-armk_c.d2 * (- sncs_d.c2 * k9 + sncs_d.s2 * k14) +\
                      k4 * k10);\
        sncs_d.d2y = -sncs_d.d3 * k11;\
        sncs_d.d3y = k14;\
        sncs_d.d1z = (-armk_c.d2 * (sncs_d.c2 * k3 + sncs_d.s2 * sncs_d.c5) +\
                      k4 * k12);\
        sncs_d.d2z = sncs_d.d3 * k3;\
        sncs_d.d3z = sncs_d.c5;\
        sncs_d.r1x = (-sncs_d.s2 * k7 - sncs_d.c2 * k13);\
        sncs_d.r2x = k8;\
        sncs_d.r4x = -k13;\
        sncs_d.r1y = (sncs_d.s2 * k9 + sncs_d.c2 * k14);\
        sncs_d.r2y = k10;\
        sncs_d.r4y = k14;\
        sncs_d.r1z = (-sncs_d.s2 * k3 + sncs_d.c2 * sncs_d.c5);\
        sncs_d.r2z = k12;\
}
#endif
```

kine.h

```
extern  KINDYN  armk_c;              /* arm kinematic and dynamic    */
                                     /* constants                    */

extern  SNCS    sncs_d;              /* current sin cos, jacob coeff */
                                     /* and U5 matrix                */

extern  JNTS    jcal_c;              /* rest position joint range    */

extern  JNTS    jmin_c;              /* angles range offset values   */

extern  JNTS    jrng_c;              /* maximum joint range values   */

extern  JNTS    jmxv_c;              /* max joint velocities */
```

The variable **armk_c** contains all the arm constants : link parameters, and gravity joint loads. The variable **sncs_d** contains a set of variable kinematic parameters updated at sample time intervals : joint angles sines and cosines, the terms of 3 by 3 upper left Jacobian submatrix, computed in link 4, and the matrix $U5$. The variable **jcal_c** is the joint angle values at the 'park' position, in radians. The variable **jmin_c** is the set of angle offsets used to map joint angles expressed in solution coordinate frame $[-\pi, +\pi]$ onto joint angles expressed in range coordinates [0, range]. The variable **jrng_c** is the set of joint ranges in radians. The variable **jmxv_c** is the set of admissible velocities in radians per second.

which.h

Including this file is equivalent to #define PUMA for now.

which.h

```
#define PUMA            /* current system setting     */



#ifdef PUMA
#define ARMTYPE 1       /* for the interface          */
#define NJOINTS 6
#define VAL11           /* for the hardware clock     */
#else
#ifdef STAN
#define ARMTYPE 2
#define NJOINTS 6
#else
        not rich enough
#endif
#endif
```

hand.h

Macros to operate the pneumatic gripper.

hand.h

```
#define CLOSE    hdpos = 'o';            /* close pneumatic gripper      */

#define OPEN     hdpos = 'c';            /* open pneumatic gripper       */
```

umac.h

This file defines some useful macros that are self explanatory. The dangerous side effects of macros must be kept in mind, for example :

```
FABS(dot(vect))
```

will call **dot** twice !

umac.h

```
#define SINCOS(s, c, a) {s = sin(a); c = cos(a);}

#define FABS(a)         (((a) < 0.) ? -(a) : (a))

#define ABS(a)          (((a) < 0) ? -(a) : (a))

#define ROUND(a)        ((a - (double)(int)a >= .5) ? (int)a + 1 : (int)a)

#define TERMIO(z)       do {errno = 0; z; pause();} while (errno == EINTR);

#define GETCHAR(c)      while ((c = getchar()) == ' '               \
                               || c == '\t' || c == '\n') ;

#define QUERY(c)        printf(" (y/n) ");                          \
                         do {                                       \
                                GETCHAR(c);                         \
                         } while (c != 'y' && c != 'n');            \
                         {int v;                                    \
                         if ((v = getchar()) != '\n')               \
                          (void) ungetc(v, stdin);}
```

exiod.h

This file describes the bit definition of the 'exio' field of the **how** structure of the real time interface [6].

exiod.h

```
#define  EXTERN0     01        /* external input/output                       */
#define  EXTERN1     02        /* bit definitions                             */
#define  EXTERN2     04        /*                                             */
#define  EXTERN3     010       /*                                             */
#define  EXTERN4     020       /*                                             */
#define  EXTERN5     040       /*                                             */
#define  EXTERN6     0100      /*                                             */
#define  EXTERN7     0200      /*                                             */
#define  ARMPWR      0400      /* high power on/off bit  (high/low)           */
#define  OFFL        01000     /* external low signal to stop the arm         */
#define  RUN         02000     /* front panel switch - run bit low            */
#define  RESTART     04000     /* front panel switch - restart bit low        */
#define  HNDOH       01000     /* close pneumatic hand/release  (high/low)    */
#define  HNDCH       02000     /* open pneumatic hand/release  (high/low)     */
#define  EXTRA4      04000     /* spare output bit (not wired)                */
#define  EXTRA0      010000    /* spare I/O bit (not wired)                   */
#define  EXTRA1      020000    /* spare I/O bit (not wired)                   */
#define  EXTRA2      040000    /* spare I/O bit (not wired)                   */
#define  EXTRA3      0100000   /* spare I/O bit (not wired)                   */
```

14. Transform Data Base

A very simple data base system is implemented. Transforms are stored under their names as set in the 'name' field of the 'TRSF' structure. From the programming point of view the following functions can be called :

```
maketdb(name)
char *name;

savetr(trans, fd)
TRSF_PTR trans;
int fd;

gettr(trans, fd)
TRSF_PTR trans;
int fd;

remtr(name, fd)
char *name;
int fd;

dumpdb(fd, v)
int fd;
bool v;

compact(name)
char *n;
```

The function **maketdb** creates an empty transform data base and returns the corresponding file descriptor. This function **cannot** be called, when the real time channel is opened, this is the purpose of the option '-D'. The function **savetr** stores a transform under its name in the data base. If the transform already exits the user is prompt :

```
change ? (y/n)
```

if 'y' is answered, the value '1' is returned otherwise '0' is returned. The function **gettr** retrieves a transform and sets its value. The value '0' is returned, when the transform is found, '-2' if not. Both functions print an informative message on 'stderr' at the time the action is performed. The function **remtr** removes a transform from the data base. The value '0' is returned, when the transform is found, '-2' if not. The function **dumpdb** dumps the contents of the data base described by the first argument on the 'stdout' file. The second argument, when non zero, specifies a 'verbose' dump. The function **compact** compacts the data base, and permits to save some file space if the data base as been extensively used. This function should not be called from manipulator programs.

In manipulator programs, use the file descriptor **fddb** as argument for the data base functions. All these functions return '-1' if something goes wrong. The messages are :

Informative messages :

```
savetr : NAME created : DATE
savetr : NAME changed at DATE
savetr : NAME added at DATE
gettr : NAME last change : DATE
gettr : NAME not found
remtr : NAME removed
remtr : NAME not found
dump : NUMBER entries
```

Errors messages are :

```
read error on data base file
write error on data base file
seek error on data base file
can't duplicate data base file
could'nt unlink
bad magic number
could'nt creat transform data base file
open error on data base file
search error
data base file saturated
```

A data base editor called *edb* allows the user to maintain transforms files. The user can modify an active transform with patches or multiplications The active transform can also be read from the data base, renamed, or reset to the unity transform. Transforms can be added to, changed in, or removed from the data base. All combinations are thus allowed. When a 'break' is typed at the terminal the following message is printed :

```
These commands are executed one per line:
        q               quit and save file
        q!              quit and do not save
        d[v]            dump data base [verbose]
        u <name>        use transform 'name'
        s               save active transform
        n <name>        rename active transform
        :               show active transform
        r <name>        remove transform 'name' from file
        i               invert active transform
        pt x y z        patch a translation x y z
      p <X/Y/Z> a     patch a rotation a around X, Y, or Z axis
        pa x y z x y z  patch a rotation defined by a and o vectors
        pe phi the psi  patch a rotation from Euler angles
        pr phi the psi  patch a rotation from roll pitch and yaw angles
These commands are cumulative:
        mt x y z        multiply by translation x y z
      m <X/Y/Z> a     multiply by rotation a around X, Y, or Z axis
        ma x y z x y z  multiply by rotation defined by a and o vectors
        me phi the psi  multiply by rotation from Euler angles
        mr phi the psi  multiply by rotation from roll pitch and yaw angles
```

15. Details

15.1. Compile

Nothing special about compilations, use UNIX's *cc* command. In order to be able to include the declaration files independently from the directory they may have been be stored in, a possibility is to define a shell variable, 'rccl' say, in your .login or .profile files as

```
rccl="-I/b/rccl/h"        for sh users
set rccl=(-I/b/rccl/h)    for csh users
```

15.2. Link

Your code must be linked with four libraries :

rccl.a	The real time version basic library
dbot.a	The data base library
rtc.a	The real time channel
libnm.a	system new math library

One may conveniently expand the 'rccl' shell variable :

```
rccl="-I/b/rccl/h /b/rccl/l/rccl.a /b/rccl/l/dbot.a /b/rccl/l/rtc.a -lnm"
set rccl=(-I/b/rccl/h /b/rccl/l/rccl.a /b/rccl/l/dbot.a /b/rccl/l/rtc.a -lnm)
```

Such that you can type :

```
$ cc myprog.c $rccl
```

In order to get the planning version, just set a shell variable, 'plan' say :

```
plan="-I/b/rccl/h /b/rccl/l/rccl.plan /b/rccl/l/dbot.a /b/rccl/l/rtc.a -lnm"
set plan=(-I/b/rccl/h /b/rccl/l/rccl.plan /b/rccl/l/dbot.a /b/rccl/l/rtc.a -lnm)
```

and type :

```
$ cc myprog.c $plan
```

15.3. Lint

Linting programs proves to be very useful, set a shell variable, 'rlint' say :

```
rlint="-I/b/rccl/h -v /b/rccl/l/llib-rccl /b/rccl/l/llib-dbot /b/rccl/l/llib-rtc"
 set rlint=(-I/b/rccl/h -v /b/rccl/l/llib-rccl /b/rccl/l/llib-dbot /b/rccl/l/lli
```

and type :

```
$ lint myprog.c $rlint
```

The llib-rccl, llib-dbot, llib-rtc files contain the descriptions of the functions compiled and stored in the corresponding libraries.

15.4. Run

Type :

```
$ a.out [-options]
```

once the channel has been set up and the arm calibrated. The options can be cumulated after '-' (except the 'D' option) :

```
$ a.out -b -v -e -d -g -k -Ddata
```

is equivalent to

```
$ a.out -bvedgk -Ddata
```

You will get the programs *calib, mkenc, play, dl ,edb, and dsp* if the 'path' of your shell leads to the right directory :

```
PATH=$PATH:/b/rccl/s            (for sh users)
export PATH

set path=($path /b/rccl/s)      (for csh users)
```

16. The display program.

The *dsp* program uses the terminal in pseudo graphic mode like a page editor. The user's terminal must possess screen addressing capabilities (see termcap(5)). The user's session environment shell variable TERM must be set to the corresponding terminal (adm3a, adm5, vt100, etc..). By default, the *dsp* program reads files of the form :

```
        ../g/file.out
```

The display of this file is obtained by typing :

```
$ dsp file
```

If no argument is given, the user is prompted.

The program displays files that are a sequence of numbers of type **double.** The program also looks for a file of the form :

```
        ../g/t.out
```

that must be a sequence of same length of numbers of type **int.** If the file 't.out' has the proper length, these numbers will appear in the left column of the display. The program also looks for a file of the form :

```
        ../g/c.out
```

that must be a sequence of characters. These characters will be used for the display on the basis of a one to one correspondence. If the character file is not found, *dsp* uses a '#'. The pseudo graphic display is tilted of 90 degrees to provide a maximum resolution. (low on the left, hight on the right, instead of the usual bottom/top). If you do not like the idea of the "../g" directory place in your shell's environment :

```
GRAPHDIR="the directory you like"
```

but the planning library assume that the "../g" directory exists. The program is interactive and the help message is :

```
!*/s/r/u/d/g/h/b/f/a/v/p/q/+-n/?

     ?      his message
   +-n    [+,-]digits <space> : direct addressing
     q      quit
     p      position display
     v      velocity display
     a      acceleration display
     f      forward one page
     b      backward one page
     h      half page forward
     g      half page backward
     d      down one line
     u      up one line
     r      redraw
     s      scale
     @      back to prompt
    !*    ! <space> file <space> : show another file

Type any character to continue
```

17. References

[1] Kernighan ,B. K., "The C Programming Language", Prentice-Hall, 1978.

[2] Paul, R. P., "Robot Manipulators: Mathematics, Programming, and Control", MIT Press 1981.

[3] Hayward V., "Introduction to RCCL : A Robot Control "C" Library", TR-EE 83-43, October 1983.

[4] "High Speed QBUS-UNIBUS Interface", Engineering Drawings, School of EE, Purdue University, Nov. 1963.

[5] Fisher, W. D., "The Modification of a Robotic Manipulator and Digital Controller to Incorporate Both Force and Possition Control", MSEE Thesis, Purdue University, May 1981.

[6] Hayward V., "Robot Real Time Control User's Manual", TR-EE 83-42, October 1983.

[7] Paul, R. P., Shimano, B. E., Mayer , E. G., "Kinematic Control Equations for Simple Manipulator", IEEE Transactions on Systems, Man, and Cybernetics, Vol SMC-11, No 6, June 1981.

[8] Zhang, H., Paul, R. P., "Determination of Simplified Dynamics of Puma Manipulator", Purdue University.

[9] Paul, R. P., Rong Ma, Zhang H., "The Dynamics of the Puma Manipulator", The International Journal of Robotic Research, (to be published).

(Revised 8/83)

MINIMUM DISTANCE COLLISION-FREE PATH PLANNING FOR INDUSTRIAL ROBOTS WITH A PRISMATIC JOINT*

J. Y. S. Luh and C. E. Campbell
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Abstract

A collision-free path is a path which an industrial robot can physically take while traveling from one location to another in an environment containing obstacles. Usually the obstacles are expanded to compensate the body width of the robot. For robots with a prismatic joint, which allows only a translational motion along its axis, additional problems created by the long boom are handled by means of pseudo obstacles which are generated by real obstacle's edges and faces. The environment is then modified by the inclusion of pseudo obstacles which contribute to the forbidden regions. This process allows the robot itself again to be represented by a point specifying the location of its end effector in space. An algorithm for determining the shortest distance collision-free path given a sequence of edges to be traversed has been developed for the case of stationary obstacles.

* Supported by NSF Grants DAR (APR) 77-14533 and MEA-8119884.

I. INTRODUCTION

Industrial robots are computer-controlled mechanical manipulators which perform tasks for industrial applications. One of the essential operations in all the assigned tasks involves the physical motion of the manipulator whose end effector travels from a known initial position and orientation to a specified goal position and orientation. In reality, the workspace of the robot is not free from obstacles such as fixtures, mechanical parts, etc., so that a collision may result if the robot moves freely without any guidance. If, however, the positions and orientations of all the obstacles are known for the entire time interval of operation, it is possible to plan a collision-free path, if one exists, for the robot to travel along while performing its task.

The subject of collision-free path planning is relatively new. Within the past five years, only a handful of people have been actively working on this subject. Among them are Pieper [1] and Widdoes [2] who used planes, cylinders, and spheres to represent obstacles (objects). The use of spheres has an advantage of avoiding the orientation problem. However, the free space that is occupied by parts of the spheres is wasted for planning purposes. In addition, the intersection functions are often nonlinear involving square roots or transcendental functions. Udupa [3], Lozano-Perez and Wesley [4], and Lozano-Perez [5,6], and Brooks [7] adopted the polyhedra as the models which result in linear intersection functions. But the orientation problem must be handled with care. Udupa discretized the space into cells which were labelled free if not occupied by obstacles and objects. Lists of free cells are joined together to form a collision-free path. To allow for arbitrary orientation, the obstacles' expansions overcompensate, which reduce the number and/or size of the free cells available for path planning. Lozano-Perez described linked polyhedra using swept volumes. The rotation range is then divided into a finite number of slices. Brooks adopts the idea of generalized cones [8] which are equivalent to swept volumes. Free space is then represented as overlapping generalized cones.

In the methods described above, some determine the free space inside which the point robot may move freely without collisions with obstacles, while others determine the forbidden region so that a collision-free path may be traced along the boundaries of the region. This paper adopts the second approach to the problem which involves objects and obstacles that interact with a robot which has a prismatic[†] link, such as the Stanford manipulator [9]. The prismatic joint, however, creates additional problems. As usual, the objects and obstacles are approximated by enclosing polyhedra. The manipulator is represented by a point; in particular, the point at the tip of the end effector. Its real body width is compensated for by expanding the polyhedral obstacles [3-6]. Methods of constructing the expanded polyhedra are given in these references. If the point robot enters into the expanded polyhedra, a collision will then occur. Now since the prismatic joint of the manipulator has a long boom, it creates two pseudo obstacles: one by the restriction that the front of the boom remain free of collision and the other by any confinement of the rear of the boom due to obstacles. The pseudo obstacle is not a physical object but a region of shadow in the workspace. However, when the point robot enters into the pseudo obstacle, a collision between the boom and a polyhedral obstacle occurs somewhere along its length. Thus the pseudo obstacles together with the expanded polyhedra form the forbidden regions that the point robot must stay away to avoid collisions. The discussion begins with 2-dimensional problems with stationary objects and obstacles, and is then extended to 3-dimensional problems.

[†]The joint that allows only a translational motion along its axis is conventionally called the prismatic joint. The sliding link of the joint is called the prismatic link. The terminology was introduced by J. Denavitt and R. S. Hartenberg in their paper entitled, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," ASME Transactions (Vol. 77): Journal of Applied Mechanics, Vol. 22, June 1955, pp. 215-221.

II. PSEUDO OBSTACLES OF STANFORD MANIPULATOR

A Stanford manipulator is shown in Figure 1 which consists of a platform, a pillar, a long boom, and a forearm with a wrist and a gripper. A 2-dimensional space will be considered first. Refer to Figure 2(a) and (b) for a relative location between a polygonal object and the boom. It is seen that at the front end of the boom there is a pseudo obstacle, the shaded area in Figure 2(b), which is created by the geometrical configurations of the polygonal object and the robot boom and their relative locations. The pseudo obstacle is completely determined by three parameters: d, θ_1 and θ_2. These three parameters can be determined as follows. Extend the boom towards the polygonal object and locate the two vertices which lie farthest from the boom pivot and that the boom can touch: one at the left and one at the right. Connect these two corners by a straight line, and d is the distance from the line to the boom pivot. Also connect the two corners to the boom pivot by straight lines. θ_1 and θ_2 are the angles between these two lines and d. Intuitively, the existence of the pseudo obstacle depends on the condition that $d < L$, where L is the length of the boom. Finally, the forbidden region at the front end of the boom is the union of the pseudo obstacle and the polygonal object. Using this approach, the problem associated with hidden lines of obstacles is avoided.

In some industrial robots such as Unimate 2100G of Unimation Inc., the rear of the boom is concealed in its housing. Obviously the rear of the boom is of no concern as long as the body of its housing is compensated for. However, some other robots, such as the Stanford manipulator, have the entire booms exposed openly without covers, and additional care must be exercised as follows. If a polygonal obstacle is close to the rear of the boom, another pseudo obstacle is created by the restrictions imposed by requiring that the rear of the boom remain free of collision. Refer to Figure 2(c), where the rear end rests against an edge of the polygonal object. If the edge is long enough, then, by sliding the rear end against the edge, the front end traces a "water-drop-like" figure. The "drop" forms a pseudo obstacle at the front of the boom in the sense that whenever the point robot enters into the "drop", a collision somewhere along the rear end occurs. Thus the pseudo obstacle outlines an additional forbidden region.

The "drop" may be described in terms of two constants L and h, and two variables z and u, as shown in Figure 2(c), where L is the length of the boom and h the distance from the boom pivot to the edge of the polygonal object. From the figure it is seen that

$$\frac{z}{h} = L/(a^2+h^2)^{1/2} \quad \text{for} \quad z \geq h \tag{1}$$

which leads to a relation between z and u:

$$z^2 = (hL)^2/[u^2+L^2-z^2+h^2+2u(L^2-z^2)^{1/2}],\ z \geq h \tag{2}$$

To construct a polygon that encloses the drop, as shown in Figure 3, first the tangent function is determined:

$$\frac{dz}{du} = ahL(L^2-z^2)^{1/2}/[ahLz-(L^2-z^2)^{1/2}(a^2+h^2)^{3/2}] \tag{3}$$

The polygonal enclosure has three defining points: the top where the tangent is horizontal, the side where u has its extreme value, and the cusp at the bottom.

<u>Top</u> $a \rightarrow 0$, $u \rightarrow 0$, $z \rightarrow L$, and $dz/du \rightarrow 0$.

<u>Side</u> $dz/du \rightarrow \infty$ so that

$$ahLz - (L^2-z^2)^{1/2}(a^2+h^2)^{3/2} = 0 \tag{4}$$

which leads to

$$u = h[(L/h)^{2/3} - 1]^{3/2} \tag{5}$$

If $L < h$, then u becomes imaginary which implies that the "drop" does not exist since the rear of the boom is not able to touch the edge of the polygonal object.

Cusp $u \rightarrow 0$, $z \rightarrow h$, $a^2 \rightarrow (L^2-h^2)$ so that

$$dz/du = -h/(L^2-h^2)^{1/2} \tag{6}$$

which describes the slope of the tangent to the left of the cusp.

The polygonal enclosure of the "drop" wastes some free space but preserves polygonal descriptions of the forbidden region. The amount of waste may be computed as follows. From Figure 4, it is seen that the area of enclosure is

$$A_e = [2(L-z_s) + (z_s-h)]|u| \tag{7}$$

where $\quad z_s = h[1 + |u|/(L^2-h^2)^{1/2}]$ (8)

The area of the "drop", however, is

$$A_d = 2\int_h^L (1 - h/z)(L^2-z^2)^{1/2}dz \tag{9}$$

Let $\gamma = L/h$, then the ratio

$$\frac{A_d}{A_e} = \frac{\gamma^2\pi/2 + (\gamma^2-1)^{1/2} - \gamma^2\sin^{-1}(1/\gamma) - 2\gamma\ln[\gamma + (\gamma^2-1)^{1/2}]}{[2(\gamma-1) - (\gamma^{2/3}-1)^{3/2}(\gamma^2-1)^{-1/2}][\gamma^{2/3}-1]^{3/2}} \tag{10}$$

and $A_d/A_e \rightarrow \pi/4 \cong 0.785$ as $\gamma \rightarrow \infty$. It should be noticed that the ratio depends on γ only. A few values were computed and tabulated below:

γ	=	1.001	5.0	10.0	10^2	10^3	10^4
A_d/A_e	=	0.8584	0.8148	0.8080	0.7982	0.7908	0.7871

Thus the efficiency of the 2-dimensional enclosure of the "drop" varies from 78.5% to near 85.84%.

If the edge of the confining polygonal obstacle at the rear of the boom is short enough, then the forbidden region "drop" is clipped or deformed as shown in Figure 5(a) and (b) respectively. For the first case, the clipped polygon can be determined once θ is determined. In the second case, the deformed "drop" can be modified by clipping and is contiguous to another clipped drop due to the second edge if β is known. For simplicity, the original polygonal forbidden region is used in clipping to avoid additional computation although some free space is wasted.

If the 2-dimensional polygonal enclosure of the "drop" is rotated about the z-axis, a cylinder capped by a circular disk on the top and a cone at the bottom results as shown in Figure 6(a). To retain the 3-dimensional pseudo obstacle in the form of a polyhedron, another enclosure must be further developed. First the cylindrical section is modelled by an n-faceted polyhedral pillar which encloses the cylinder. Refer to Figure 6(b) in which

$$R = |u| / \cos(\pi/n) \tag{11}$$

Then, in the Cartesian coordinates as shown in the figure, the n vertices of the n-faceted polygon of the pillar at the top are located at $(R \cos k\pi/n, R \sin k\pi/n, L)$ for $k = 0,1,\ldots,n-1$; while those at the lower end of the pillar are located at $(R \cos k\pi/n, R \sin k\pi/n, z_s)$. The cusp point is situated at $(0,0,h)$.

III. SHORTEST PATH AMONG AN ORDERED SET OF CHORDS

It is known that a shortest, collision-free path for a point robot in an environment composed of polyhedral obstacles in 2-dimensional workspace consists of line segments connecting an ordered set of vertices of some of these obstacles [4,5]. To determine the shortest path in a 3-dimensional workspace, consider an ordered set of N chords which represent the edges of polygonal forbidden regions as shown in Figure 7, where $\underline{A}_i$ and $\underline{B}_i$ are the two end points of chord i for i = 1,2,...,N. Then, any point on chord i can be represented by

$$\underline{P}_i = \alpha_i(\underline{A}_i - \underline{B}_i) + \underline{B}_i \quad \text{for} \quad 0 \leq \alpha_i \leq 1. \tag{12}$$

Thus $(\underline{P}_i - \underline{P}_{i-1})'(\underline{P}_i - \underline{P}_{i-1})$ is the distance squared between two points on two adjacent chords (adjacent in terms of the sequence to be traversed), where ( )' = transpose of ( ).

Let $$J(\alpha_1, \ldots, \alpha_N) = \sum_{j=2}^{N} (\underline{P}_j - \underline{P}_{j-1})'(\underline{P}_j - \underline{P}_{j-1}) \tag{13}$$

be the total distance squared from $\underline{P}_1$ to $\underline{P}_2$ ... to $\underline{P}_N$. Since the initial starting location and the terminal goal location is known, α_1 and α_N are given. Thus minimizing J with respect to α_i for i = 2,3,...,N-1 is equivalent to finding the shortest path. This is a simple quadratic programming problem with N variables and N constraints, which can be solved by a number of known methods. One of them is the symmetric variant approach which is a generalization of the simplex for linear programming [10, p. 270]. In this method, the number of primal basic variables is not always equal to N, but may vary from N to 2N [10, p. 274]. The asymmetric method [10, p. 280] by Dantzig [11], and later independently by van de Panne [12], does not have the parametric variation of the incoming variable. Instead, this variable enters the basis as soon as it is assigned a nonzero value. Consequently the nonstandard tableaux occur. Graves [13] and Lemke [14, 15] propose a parametric method which is the generalization of Dantzig's self-dual parametric method for linear programming [10, p. 310]. The difference is that

Graves' approach is equivalent to the symmetric variant of the self-dual parametric method while Lemke's is to the asymmetric variant scheme. It is known [10, p. 279] that the symmetric variant approach converges in no more than $(2N)!/(N!)^2$ steps. The asymmetric method, however, does not make any use of the existing symmetry so that the computational effort is increased [10, p. 320]. In the following, an iterative algorithm is presented which converges in no more than $(N-2)(N-1)/2$ iterations, which is much less than $(2N)!/(N!)^2$ iterations.

From (12) one obtains

$$\underline{P}_j-\underline{P}_{j-1} = \alpha_j(\underline{A}_j-\underline{B}_j) - \alpha_{j-1}(\underline{A}_{j-1}-\underline{B}_{j-1}) + \underline{B}_j-\underline{B}_{j-1} \tag{14}$$

Let $\underline{C}_j = \underline{A}_j-\underline{B}_j$, $\underline{D}_j = \underline{B}_j-\underline{B}_{j-1}$, and ()' be the transpose of (), then

$$(\underline{P}_j-\underline{P}_{j-1})'(\underline{P}_j-\underline{P}_{j-1}) + (\underline{P}_{j+1}-\underline{P}_j)'(\underline{P}_{j+1}-\underline{P}_j)$$

$$= 2\alpha_j^2\underline{C}_j'\underline{C}_j + 2\alpha_j\underline{C}_j'[(-\alpha_{j-1}\underline{C}_{j-1}+\underline{D}_j) - (\alpha_{j+1}\underline{C}_{j+1}+\underline{D}_{j+1})] \tag{15}$$

so that

$$\partial J/\partial\alpha_i = \partial[\sum_{j=2}^{N}(\underline{P}_j-\underline{P}_{j-1})'(\underline{P}_j-\underline{P}_{j-1})]/\partial\alpha_i$$

$$= 4\alpha_i\underline{C}_i'\underline{C}_i + 2\underline{C}_i'[(-\alpha_{i-1}\underline{C}_{i-1}+\underline{D}_i) - (\alpha_{i+1}\underline{C}_{i+1}+\underline{D}_{i+1})] \tag{16}$$

and

$$\partial^2J/\partial\alpha_i^2 = 4\underline{C}_i'\underline{C}_i \tag{17}$$

Since for $i=2,3,\ldots,N-1$, $\underline{C}_i = \underline{A}_i-\underline{B}_i \neq \underline{0}$. Then $\partial^2J/\partial\alpha_i^2 > 0$. Let $\{\alpha_i^*\}$ be the solution set to $\partial J/\partial\alpha_i = 0$, i.e., $\{\alpha_i^*\}$ satisfies

$$-(\underline{A}_i-\underline{B}_i)'(\underline{A}_{i-1}-\underline{B}_{i-1})\alpha_{i-1}^* + 2(\underline{A}_i-\underline{B}_i)'(\underline{A}_i-\underline{B}_i)\alpha_i^*$$

$$-(\underline{A}_i-\underline{B}_i)'(\underline{A}_{i+1}-\underline{B}_{i+1})\alpha^*_{i+1} = (\underline{B}_{i-1}-2\underline{B}_i+\underline{B}_{i+1})'(\underline{A}_i-\underline{B}_i) \tag{18}$$

for i = 2,3,...,N-1. Hence $\{\alpha_i^*\}$ yields a minimum J. System (18) yields a system of (N-2) equations with (N-2) unknown α_i^*'s. The system is a tri-banded structure which is easy to solve. However, since the constraints $0 \leq \alpha_i \leq 1$ are not imposed in equation (18), the solution may be infeasible. To convert the infeasible solution to a feasible solution, suppose that among the α_i^*'s, at least one α_k^* is such that $\alpha_k^* \notin [0,1]$. Let $\hat{\alpha}_k$ be the value indicating the intersection of a path at edge k. Then

$$J(\alpha_1,\alpha_2^*, \ldots, \hat{\alpha}_k, \ldots, \alpha_{N-1}^*,\alpha_N) > J(\alpha_1,\alpha_2^*, \ldots, \alpha_k^*, \ldots, \alpha_{N-1}^*,\alpha_N) \tag{19}$$

By (15), one may express J as

$$J = a\,\alpha_i^2 + b\,\alpha_i + c$$

where the parameters a, b and c do not contain any terms of α_i. Thus $\partial J/\partial\alpha_i = 0$ yields

$$\alpha_i^* = -b/(2a)$$

so that

$$J^* = J(\{\alpha_i^*\}) = -b^2/(4a) + c$$

Now

$$J(\alpha_1,\alpha_2^*, \ldots, \hat{\alpha}_i,\ldots,\alpha_{N-1}^*,\alpha_N) - J^* = a\hat{\alpha}_i^2+b\hat{\alpha}_i+c - [-b^2/(4a)+c]$$

$$= a[\hat{\alpha}_i+b/(2a)]^2$$

$$= a[\hat{\alpha}_i-\alpha_i^*]^2 \geq 0$$

since $a = 2\underline{C}_i'\underline{C}_i > 0$. Now J^* is fixed, thus $J(\alpha_1,\alpha_2^*, \ldots, \hat{\alpha}_k, \ldots, \alpha_{N-1}^*,\alpha_N)$ decreases as $|\hat{\alpha}_k-\alpha_k^*|$ decreases. Based on this property, $\hat{\alpha}_i$ is set to zero if α_i^* is less than zero, or set to one if greater than one. An iterative algorithm for obtaining a feasible solution is then developed. In the algorithm, α_i^* is written as α_i for simplicity. Essentially, the procedure starts with solving the system equation (18). If the solution is infeasible, then choose one of the α_i's that is not in the range [0,1], but it is closest to the starting edge. Set this α_i to its closest allowable extreme value (i.e., either zero or one) temporarily, then divide the remaining α_i's into two groups and solve equation (18) again for them. Now, if there are infeasible α_i's which are even closer to the initial edge than before, then repeat the procedure for all the α_i's that were computed during this iteration. Otherwise only the second group is re-iterated. The procedure repeats until all the feasible α_i's are obtained. The detailed algorithm is as follows:

Step 1. Set I = 1 and J = 1.

Step 2. Solve (18) for $\{\alpha_i\}_{I+1}^{N-1}$ and $\{\alpha_i\}_{J+1}^{I-1}$. Here $\{\alpha_i\}_{V}^{U}$ = empty ϕ if $V > U$.

Step 3. Collect all those α_i's such that $\alpha_i \notin [0,1]$. Among these α_i's, find the smallest i and call it k; i.e., find $k = \min\limits_{\alpha_i} i$ for $\alpha_i \notin [0,1]$.

Step 4. If $\{\alpha_i | \alpha_i < 0 \cup \alpha_i > 1\} = \phi$, (i.e., if no disallowed α_k exists), stop the process and output the feasible solution. Otherwise, continue.

Step 5. If $\alpha_k < 0$, set $\alpha_k = 0$. If $\alpha_k > 1$, set $\alpha_k = 1$. Continue.

Step 6. If $k < I$: set $I = k$. Go to Step 2.
Else if $k > I$: set $J = I$, then $I = k$. Go to Step 2.

The convergence proof of the algorithm is shown in the next section. Using the algorithm, a feasible solution may be obtained in no more than $\frac{(N-2)(N-1)}{2}$ iterations. Usually, the solution will be found in fewer than N steps. A FORTRAN program has been written for the implementation. Using numerical examples, the feasible solutions produced from the program agree with those that were computed from the Powell's improved Davidon-Fletcher-Powell method [16].

IV. CONVERGENCE OF THE ALGORITHM

To show the convergence of the algorithm, let I_n, J_n and k_n be the values of indices I, J and k, respectively, at the n-th iteration. Initially let $I_0 = 1$ and $J_0 = 1$ according to Step 1. Since $I_0-1 < J_0+1$, then in Step 2 one obtains $\{\alpha_i\}_{J_0+1}^{I_0-1} = \phi$ and $I_0+1 \leq i \leq N-1$. Thus by Step 3, min $i = k_0 > I_0$ provided k_0 exists. From Step 6, one obtains $J_1 = I_0$ and $I_1 = k_0$ so that

$$J_1 < I_1 \tag{20}$$

For n=1, one obtains from Step 2 that $J_1+1 \leq i \leq I_1-1$ and $I_1+1 \leq i \leq N-1$ but $i \neq I_1$. However, $k_1 = \min i$ for $\alpha_i \notin [0,1]$ so that, if k_1 exists then either $J_1+1 \leq k_1 \leq I_1-1$ or $I_1+1 \leq k_1 \leq N-1$.

(a) $J_1+1 \leq k_1 \leq I_1-1$. This implies $J_1 < k_1 < I_1$. Now Step 6 yields $I_2 = k_1$ so that

$$I_2 < I_1 \tag{21}$$

and

$$J_1 < I_2 \tag{22}$$

Inequality (21) implies that the value of I is decreasing. But the value of J is not altered, i.e.

$$J_2 = J_1 \tag{23}$$

Hence combine (22) and (23) to yield

$$J_2 < I_2 \tag{24}$$

(b) $I_1+1 \leq k_1 \leq N-1$. This implies $k_1 > I_1$, and Step 6 yields $J_2 = I_1$ and $I_2 = k_1$. The second equality implies

$$I_2 > I_1 \tag{25}$$

which means that the value of I is increasing. Combining the first equality of case (b), i.e., $J_2 = I_1$, and (20) yields

$$J_2 > J_1 \tag{26}$$

i.e., the value of J is increasing. Again, combining the first equality and (25) yields

$$J_2 < I_2 \tag{27}$$

Thus, in either case, $J_2 < I_2$.

By induction, one obtains $J_n < I_n$ for $n=1,2,\ldots$ in general. Now Step 2 yields $J_n+1 \leq i \leq I_n-1$ and $I_n+1 \leq i \leq N-1$ but $i \neq I_n$. If $k_n = \min i$ exists, then either $J_n+1 \leq k_n \leq I_n-1$, or $I_n+1 \leq k_n \leq N-1$. Follow the same reasoning described above, one can conclude that:

(a) For the case of $J_n+1 \leq k_n \leq I_n-1$, $n=1,2,\ldots$

$I_{n+1} < I_n$ (decreasing in I)

$J_{n+1} = J_n$ (no change in J)

$J_{n+1} < I_{n+1}$ (J is the lower bound of I)

(b) For the case of $I_{n+1} \leq k_n \leq N-1$, $n=1,2,\ldots$

$I_{n+1} > I_n$ (increasing in I)

$J_{n+1} > J_n$ (increasing in J)

$J_{n+1} < I_{n+1}$ (J is the lower bound of I)

SITUATION A

If at the n-th iteration, the value of k is such that $I_n + 1 \le k_n \le N-1$, then J_n, the value of J at n-th iteration, must increase at least by one to become J_{n+1}. In general, if during the process of iterations beginning from n-th iteration, the value of k falls in the interval $[I+1, N-1]$ in any $(N-J_n-1)$ iterations, then the value of J must increase at least by $(N-J_n-1)$, so that the minimum possible value for J will be $J_n + (N-J_n-1) = N-1$. Since J is the lower bound of I, i.e., $J < I$, the minimum possible value for the corresponding I will be N. Consequently at the immediate next iteration, the two largest possible sets are $\{\alpha_i\}_{I+1}^{N-1} = \{\alpha_i\}_{N+1}^{N-1}$ and $\{\alpha_i\}_{J+1}^{I-1} = \{\alpha_i\}_{N}^{I-1}$. Since the largest possible value for I at any time is N, both sets are empty so that the iteration process is terminated.

SITUATION B

If at the n-th iteration, however, the value of the k is such that $J_n + 1 \le k_n \le I_n - 1$, then the value of J does not change so that $J_{n+1} = J_n$. But I_n, the value of I at n-th iteration, must decrease at least by one to become I_{n+1}. Now the highest possible value for I_n is N. Beginning at the n-th iteration, it is possible to have a maximum of $(N-J_n-1)$ consecutive iterations (including n-th iteration) such that at each one of them the value of k falls in the interval $[J+1, I-1]$. The reason is that at all these consecutive iterations, the value of J remains the same so that $J_n = J_{n+1} = \cdots = J_{n+N-J_n-1}$. But the value of I decreases at least by one at each iteration so that at the end of the consecutive iterations, the I_{n+N-J_n-1} has a maximum possible value of $N-(N-J_n-1) = J_n + 1$. At the immediately following iteration, supposing the value of k also falls in the interval $[J+1, I-1]$. Then $J_{n+N-J_n} = \cdots = J_n$ and max. $I_{n+N-J_n} = J_n$ which is a contradiction to $J_{n+N-J_n} < I_{n+N-J_n}$. Let $c = n+N-J_n-1$. Since $\{\alpha_i\}_{J_c+1}^{\max.I_c-1} = \{\alpha_i\}_{J_n+1}^{J_n} = \phi$, the value of k for that iteration, if it exists, must fall in the interval $[I+1, N-1]$.

These two situations may alternate to yield a longest convergent process as follows. At the very beginning, $I_0 = J_0 = 1$ so that $\{\alpha_i\}_{J+1}^{I-1} = \{\alpha_i\}_2^0$ is empty and Situation A applies. Now min.$J_1 = 2$ and the longest possibility is that Situation B applies at the following $(N-\min.J_1-1) = (N-3)$ consecutive iterations. This is followed by Situation A again which yields a minimum value of 3 for J. Now Situation B applies at the next $(N-\min.J-1) = (N-4)$ consecutive iterations, etc. As shown before, Situation A cannot apply more than $(N-J_0-1) = (N-2)$ times during the entire iterative process. Thus the total number of iterations is $[1+(N-3)] + [1+(N-4)] + \cdots + [1+(N-3-\{N-2-1\})] = \sum_{i=1}^{N-2} i = (N-2)(N-1)/2$. Thus the iterative algorithm converges in at most $(N-2)(N-1)/2$ iterations. ·

V. *SUMMARY*

It was shown that for robots with a prismatic joint, such as joint 3 of the Stanford manipulator, the boom's length may be compensated for by two pseudo obstacles for every edge of the objects when the robot is, in the usual sense, represented by a point. One of the pseudo-obstacles is due to the front end of the boom, and the other is due to the rear end. An algorithm has been developed for the computation of the shortest feasible collision-free path for the robot for the case of stationary obstacles.

REFERENCES

[1] Pieper, D. C., *The Kinematics of Manipulators Under Computer Control,* ARPA Order No. 957, Stanford University, 1968.

[2] Widdoes, C., *A Heuristic Collision Avoider for the Stanford Robot Arm,* C.S. Memo 227, Stanford University, 1974.

[3] Udupa, S. M., *Collision Detection and Avoidance in Computer Controlled Manipulators,* Ph.D. Thesis, California Institute of Technology, 1977.

[4] Lozano-Perez, T. and M. A. Wesley, *An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles,* Communications of the ACM, Vol. 22, No. 10, October 1979, pp. 560-570.

[5] Lozano-Perez, T., "Automatic Planning of Manipulator Transfer Movements," IEEE Transactions on Systems, Man, and Cybernetics, Vol. 11, No. 10, October 1981, pp. 681-698.

[6] -----, *Spatial Planning: A Configuration Space Approach,* IEEE Transactions on Computers, Vol. 32, No. 2, February 1983, pp. 108-120.

[7] Brooks, R. A., *Solving the Find-Path Problem by Good Representation of Free Space, Proc. AAAI 2nd Annual National Conference* on Artificial Intelligence, August 18-20, 1982, Pittsburgh, Penn., pp. 381-386.

[8] Binford, T. O., "Visual Perception by Computer," Presented at the IEEE Systems Science and Cybernetics Conference, December 1971, Miami, Florida.

[9] Scheinman, V. D., *Design of a Computer Controlled Manipulator,* AI Memo No. 92, Artificial Intelligence Laboratory, Stanford University, June 1969.

[10] Van de Panne, C., *Methods for Linear and Quadratic Programming,* North-Holland Publishing Co., 1975.

[11] Dantzig, G. B., *Linear Programming and Extensions,* Princeton University Press, 1963.

[12] Van de Panne, C., *A Non-artificial Simplex Method for Quadratic Programming,* Report 22, International Center for Management Science, Rotterdam, 1962.

[13] Graves, R. L., "A Principal Pivoting Simplex Algorithm for Linear and Quadratic Programming," Operations Research, Vol. 15, 1967, pp. 482-494.

[14] Lemke, C. E., "Bimatrix Equilibrium Points and Mathematical Programming," Management Science, Vol. 11, 1965, pp. 681-689.

[15] ------, "On Complementary Pivot Theory," in *Mathematics of the Decision Sciences,* Part 1, edited by G. B. Dantzig and A. F. Veinott, Jr., American Mathematical Society, Providence, Rhode Island, 1968.

[16] Powell, M.J.D., "A Survey of Numerical Methods for Unconstrained Optimization," SIAM Review, Vol. 12, No. 1, January 1970, pp. 79-97.

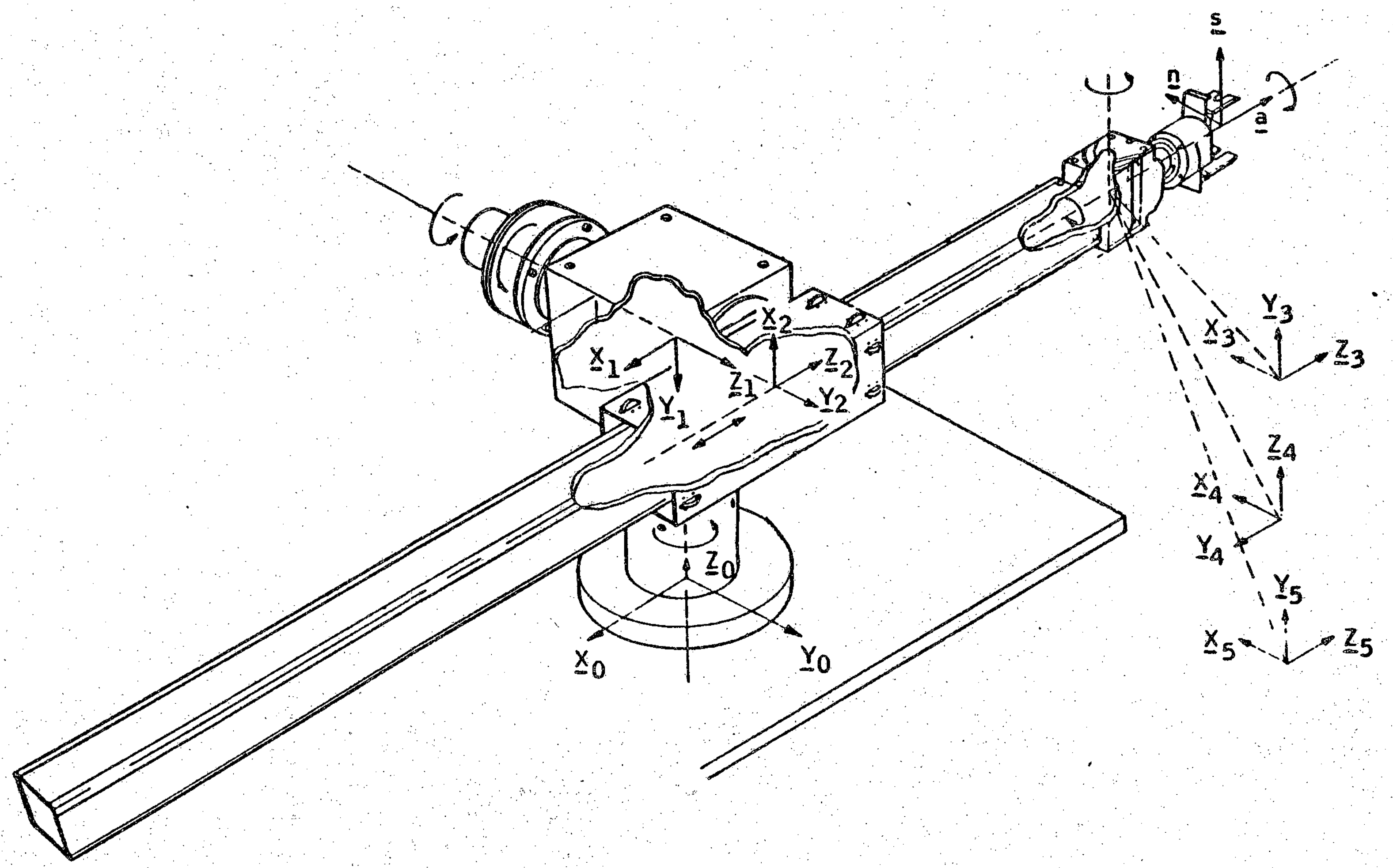


Figure 1. Stanford Manipulator.

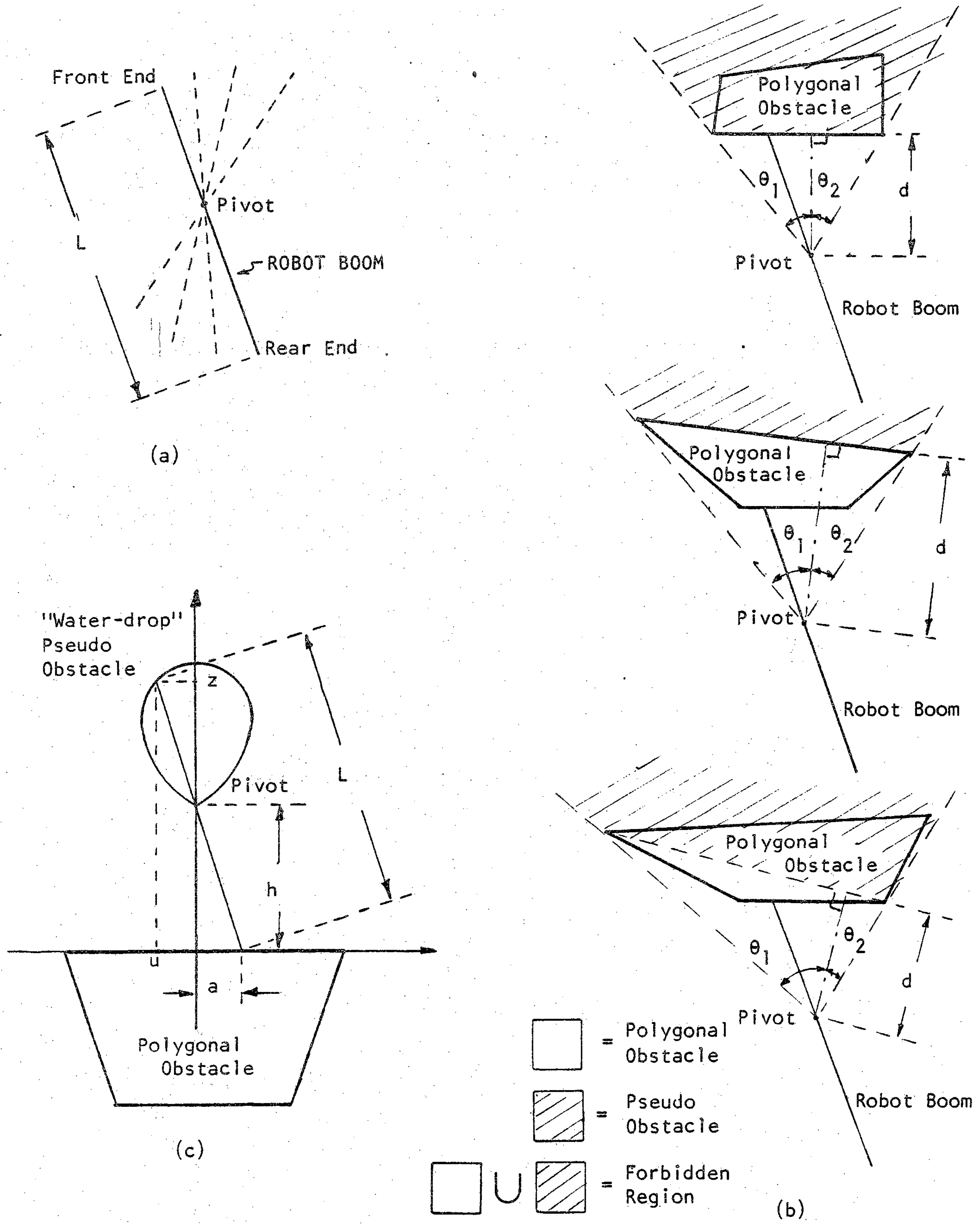


Figure 2. Pseudo Obstacles and Forbidden Regions.

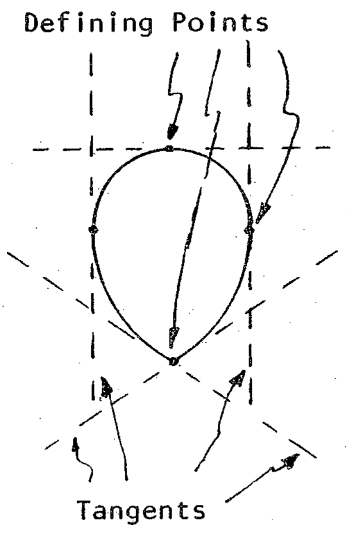


Figure 3. Defining Points Tangent of the "Water-drop".

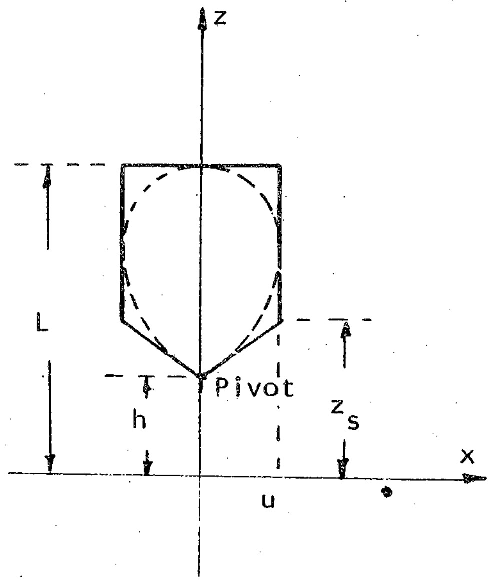


Figure 4. Enclosure of the "Water-drop".

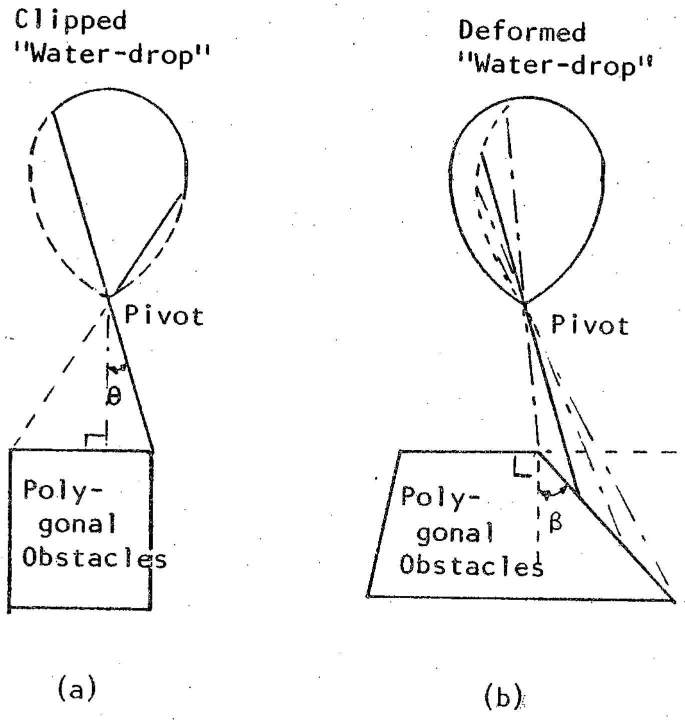
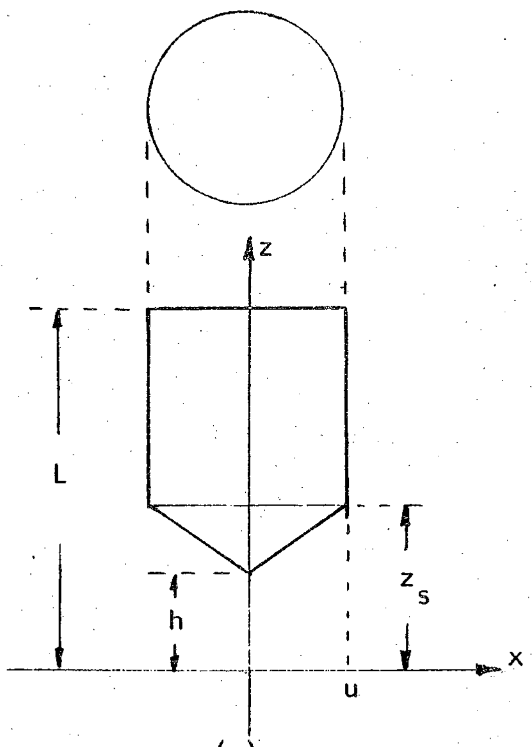
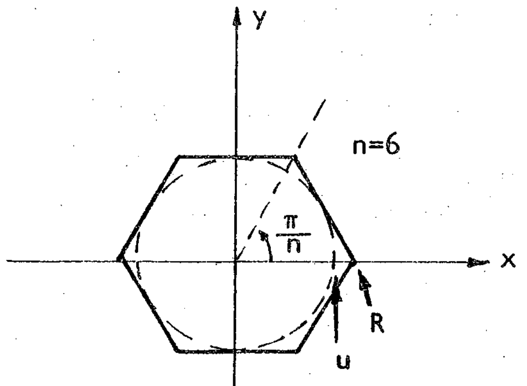


Figure 5. Clipped and Deformed "Water-drop".



(a)



(b)

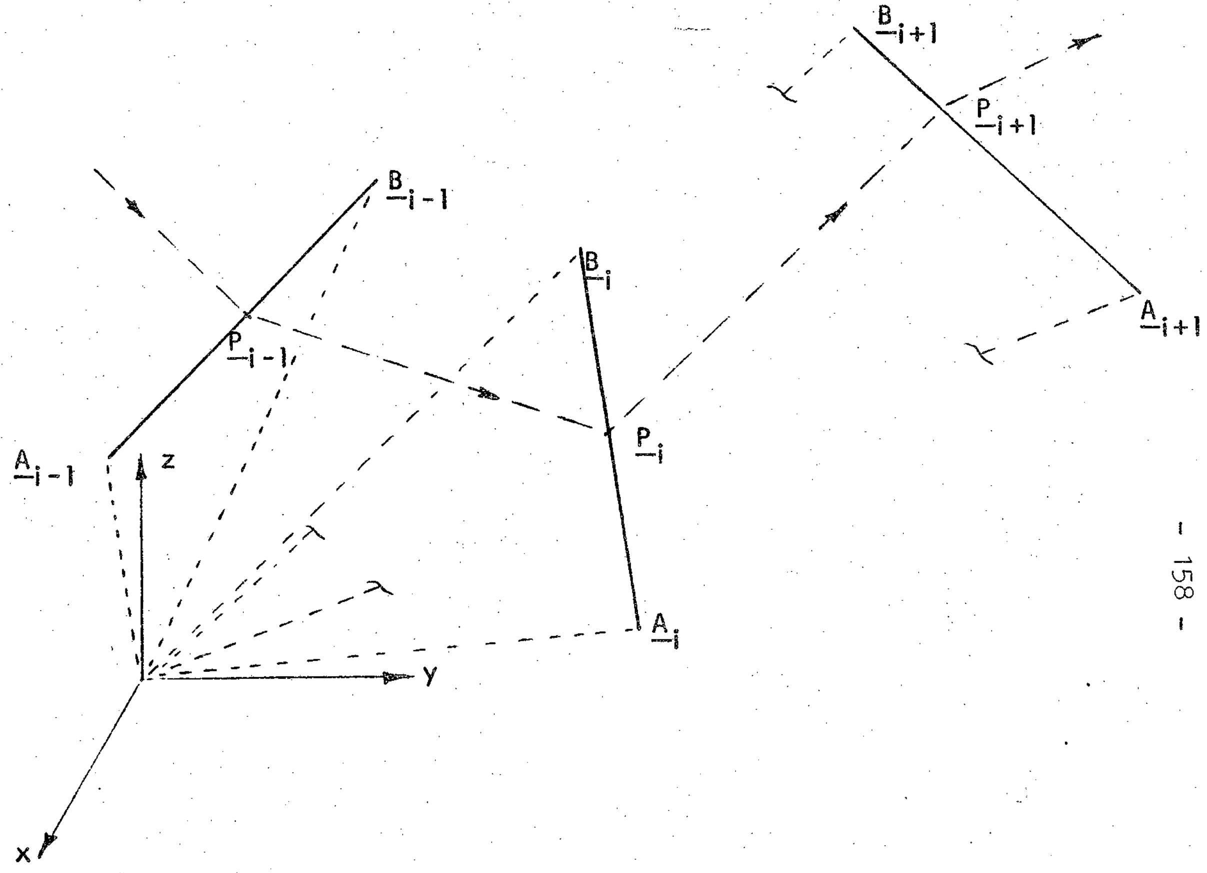


Figure 7. Collision-path Among Ordered Set of Edges.

(Revised 8/83)

REAL-TIME 3-D VISION BY OFF-SHELF SYSTEM WITH MULTI-CAMERAS FOR ROBOTIC COLLISION AVOIDANCE*

J. Y. S. Luh and J. A. Klaasen
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

A three-dimensional vision system for on-line operation that aids a collision avoidance system for an industrial robot is developed. Because of the real-time requirement, the process that locates and describes the obstacles must be fast. To satisfy the safety requirement, the obstacle model should always contain the physical obstacle entirely. This condition leads to the bounding box description of the obstacle, which is simple for the computer to process.

The image processing is performed by a Machine Intelligence Corporation VS-100 machine vision system. The control and object perception is performed by the developed software on a host Digital Equipment Corporation VAX 11/780 Computer. Also, the communication with the robot collision avoidance program occurs on the VAX 11/780.

The resultant system outputs a file of the locations and bounding descriptions for each object found. When the system is properly calibrated, the bounding descriptions always completely envelop the obstacle. The response time is data-dependent. When using two cameras and processed on UNIX time sharing mode, the average response time will be less than two seconds if eight or less objects are present. When using all three cameras, the average response time will be less than four seconds if eight or less objects are present.

*Supported by NSF Grant MEA-8119884, and Bethlehem Steel Corporation Equipment Grant-in-Aid for Education.

I. INTRODUCTION

Customarily the industrial robots are defined as computer controlled mechanical manipulators used in industrial applications [1]. In the usual robot tasks, practically all involve some manipulation requiring the travel of the end effectors from their initial positions and orientations to the specified goal positions and orientations. However there are fixtures, mechanical parts, etc. in the work-space of the root. Thus collisions between the robot and the obstacles may occur unless some guidance for traveling is provided.

A conventional approach of safe traveling is to turn off the power and stop all the actions whenever such an obstacle standing in the robot's path is detected. In doing so, no possible accident of collision will ever occur. But, by stopping the motion of the robot, its throughput and hence the productivity is reduced. Alternatively one may maneuver the robot to travel around the obstacles. The problem, however, is complicated since no a-priori knowledge about obstacles is assumed. In addition, they may appear in the robot's path unexpectedly. Consequently maneuvering an industrial robot to avoid a collision with obstacles in real time involves not only the fast obstacle detection and description, but also fast decision making.

To ensure the safety requirement, the descriptive model of the obstacle should always enclose the physical object entirely. Since the description of the obstacle must be processed through the computer, it is desirable to avoid any complicated model. As the objective is to prevent collision, a detailed description of the obstacle is not necessary. A simple model which satisfies these conditions is a bounding box as depicted in Figure 1. When the Cartesian coordinates are defined to be parallel with the edges of the box, it can be described by the lower (minimum value) and upper (maximum value) bounds of the obstacle on the three axes.

To solve the detection problem, the use of sensors is unavoidable. For practical reasons, noncontact sensors are preferred. To simplify the experimentation, passive systems that determine the range by means of multiple cameras are considered. The binary image processing is chosen to shorten the execution time. From the published literature, it is found that the vision system developed at SRI International satisfies the needs [2]. This technical approach is now commercialized. At least two manufacturers, viz. Machine Intelligence Corporation of California and Automatix of Massachusetts, have marketed vision systems which adopt and improve the SRI International's scheme.

The machine vision system is a passive system, and the camera simply detects the reflected radiation from the environment. To determine the ranges of the objects, the passive system requires multiple cameras [3]. For this project, the object perception of its 2-dimensional projection for each camera is done via the Machine Intelligence Corporation (MIC) VS-100 vision system. The sensors are General Electric TN-2200 solid-state automation cameras, which contain charge-injection-device sensors [4,5]. The pixels are spaced as squares whose centers are 0.0018 inch apart, and are practically contiguous in the 128 by 128 array. The frame time is 17,688 pixel times, which is user selectable from 0.20 milliseconds to 1.43 microseconds. Thus, frame rates could be above 2000 per minute. The spectral response of the TN-2200 reveals that the sensor is twice as sensitive to infrared signals as to violets and blues.

Within the MIC system, the contiguous regions from connectivity analysis [6] of run-length encoded data [7] are identified and organized into data structures of essential features [8]. These features are analyzed such that location, orientation, and recognition data can be communicated to an external computer. The features utilized in this project were the location of the centroids, the maximum x and y values, and the minimum x and y values of each region, where x and y are the coordinates of the 2-dimensional image of each camera. Typical processing time of the MIC system is shown in Figure 2 [9].

II. THE EXPERIMENTAL SYSTEM

Passive machine systems, unfortunately, need to correlate the pixels of each camera to the pixels of the other cameras [10,11]. Then, the intersection of the rays, that the pixels represent, is found by geometric relations. For this project, three cameras are used. The perceptions of whether the regions of each camera correspond to a real object, and the location and description of the possible object, are performed by the controlling software.

The functions to be controlled are threshold adjustment, camera parameter initialization, picture taking, and data communication. The control of the three cameras to be used is handled through the MIC system which is in turn controlled by a DEC VAX 11/780 computer. The VAX 11/780 interacts with the user and the MIC VS-100 to select a threshold and the camera parameters. The VAX 11/780 also instructs when the MIC system should take a picture, how to process it, and what results to send back. The essential response contains the location of the centroid and a description of the bounding box for each object found, in the coordinates system of the work-space. This information is output to a file which is accessable by the robot's collision avoidance program. Figure 3 shows a block diagram of the information flow of the overall scheme.

The hardware link from the VAX 11/780 to the MIC VS-100 vision system consists of a UNIBUS, DR11-C interface, two 40-pin parallel cables, and a DRV11 interface to the LSI-11 of the MIC VS-100 vision system. The VAX 11/780 uses the UNIX operating system [12]. All of the controlling software and communication interfacing has been written in the C programming language [13].

Figure 4 shows the environmental arrangement of the project. It provides a workspace of six feet high in an area of eight feet square. It is embraced by black wall-panels. There are two cameras (Camera #0 and #1) mounted horizontally and one camera (Camera #2) mounted on the ceiling. They are orthogonally mounted with each camera five feet away from the center of the work-space. The ceiling is nine feet high with standard ceiling lighting. To improve the lighting, a 75-watt floor lamp is added.

III. COMPUTATION OF 3-D LOCATION FROM CAMERA IMAGES

As seen in Figure 5, the orthogonality of the center lines of the cameras is essential. These three lines form a coordinate system (x,y,z) in such a manner that the positive x, y and z axes point towards, respectively, Cameras #0, #1 and #2; and are normal to their corresponding image coordinates (x_j, y_j), $j = 0,1,2$. If the three axes do not intersect at one point, transformation of one axis is required to form the coordinate system. In any case, system (x,y,z) is assigned as the camera coordinates. If the coordinate system of the 3-D work-space, as seen from coordinates (x,y,z), is different from (x,y,z) itself, a coordinate transformation, which involves a translation and rotations [1], is required when computing the 3-D location of the object from its camera images. For simplicity, coordinate system (x,y,z) is assumed to align with the coordinates of the work-space.

Scaling Factor of the Lens

The scaling factor is needed in computing the 3-D location. As usual the scaling factor of the lens is defined as the ratio of the object size to the object distance from the lens, which is the same as the ratio of the image size to the focal length. To determine the scaling factor experimentally, first the width of view on the plane perpendicular to the lens axis, and the distance from the lens to the plane are obtained physically with a measuring tape (see Figure 6). Then the scaling factor may be computed by the method of the least-squares-fit. For the 4.8 mm c-mount lenses used in the experiments, thirty pairs of data are read. The least-squares-fit computation yields a scaling factor of 1.264 with a standard deviation of 0.0103. Using the published data by the camera manufacturer [4], an linear interpolation results in a scaling factor of 1.2385. There is a 2% error between the two values of scaling factor. For the reported experiment, the last-squares-fit value is used.

3-D Location Formulas When Using Two Cameras

When two cameras see an object in the 3-D work-space, each has a two-dimensional image. The goal is to derive formulas that compute the location of the object in the space based on the two images. In the following, Cameras #0 and #1, which are mounted orthogonally in a horizontal plane, are considered. Formulas can be modified by exchanging appropriate image coordinate variables for combinations involving the vertically mounted camera.

Refer to Figure 5. From the object's point of view Camera #j has an image of the object with an image coordinates (x_j,y_j), $j = 0,1$. The origins of the image coordinates are user defined by positioning a cursor in each image at a pixel corresponding to a physical point which may not be the center of the image frame (microsensor). Translations may be required to shift the origins of coordinates (x_j, y_j) to rest on the axes of the work-space coordinates (x,y,z). For simplicity, let the origins of coordinates (x_0,y_0) and (x_1,y_1) be resting on x and y axes respectively. Then the correspondence between the image and the work-space coordinates are: Image coordinates ~ Work-space coordinates: $x_0 \sim y$, $y_0 \sim -z$, $x_1 \sim -x$, $y_1 \sim -z$. A point q in the work-space can be defined by a vector $\underline{q} = (q_x,q_y,q_z)$ in the coordinates of that space. Supposing the point has images in Cameras #0 and #1. These images are described in terms of pixels (p_{x0},p_{y0}) and (p_{x1},p_{y1}) in the image frames (microsensors) of Cameras #0 and #1 respectively. How can $\underline{q}$ be computed from the two images?

For $i = x,y,z$ and $j = 0,1$, let Q_{ij} be the i-th component of vector $\underline{q}$ from Camera # j's point of view. Then for Camera #0.

$$Q_{z0}/(\text{width of view}) = -p_{y0}/(\text{pixel width}) \tag{1}$$

where the negative sign comes from the correspondence between y_0 and -z as indicated in the preceding paragraph. Let d_j be the object distance from Camera #j; $j = 0,1$. Along the x-axes, the object distance from Camera #0 is $(d_0 - q_x)$. Thus, from (1),

$$Q_{z0}(d_0 - q_x)/(\text{width of view}) = Q_{z0}/(\text{scaling factor})$$

$$= -p_{y0}(d_0 - q_x)/(\text{pixel width}) \tag{2}$$

Let

$$y_j^* = p_{yj}(\text{scaling factor})/(\text{pixel width}) \tag{3}$$

where the pixel width varies with different cameras, and is 128 for GE TN-2200 in either coordinate, then one obtains from (2):

$$Q_{z0} = -y_0^*(d_0 - q_x) \tag{4}$$

Likewise,

$$Q_{z1} = -y_1^*(d_1 - q_y) \tag{5}$$

$$Q_{x1} = -x_1^*(d_1 - q_y) \tag{6}$$

$$Q_{y0} = x_0^*(d_0 - q_x) \tag{7}$$

where

$$x_j^* = p_{xj}(\text{scaling factor})/(\text{pixel width}) \tag{8}$$

Equations (4) through (7) give one relation for each of the x and y components of $\underline{q}$; but two relations for z component, one from each of the two cameras. To solve, let

$$q_x = Q_{x1}, \tag{9}$$

and

$$q_y = Q_{y0}. \tag{10}$$

Then solving equations (6), (7), (9) and (10) yields

$$q_x = x_1^*(-d_1 + x_0^* d_0)/(1 + x_0^* x_1^*) \quad \text{by Camera \#1} \tag{11}$$

$$q_y = x_0^*(d_0 + x_1^* d_1)/(1 + x_0^* x_1^*) \quad \text{by Camera \#0} \tag{12}$$

Consequently, from (4), (5), (11) and (12),

$$q_z = \begin{cases} -y_0^*(d_0 + x_1^* d_1)/(1 + x_0^* x_1^*) & \text{by Camera \#0} \\ y_1^*(-d_1 + x_0^* d_0)/(1 + x_0^* x_1^*) & \text{by Camera \#1} \end{cases} \tag{13}$$

Now $|p_{ij}|$ /(pixel width) < 1. As long as the maximum value of the ratio is less than the inverse of the scaling factor, the maximum magnitudes of x_0^* and x_1^* are less than 1 by (8), so that the denominator of (11), (12) and (13) cannot be zero. These three equations are the location formulas for the case of using Cameras #0 and #1.

3-D Location Formulas When Using Three Cameras

Although the location of the point q in the 3-D work-space can be determined by using two cameras, some of the hidden free space may not be detected, as illustrated by an experimental example in Section VI. Wasting free space is not desirable because it restricts the maneuver of the robot. This leads to the use of three cameras.

Again refer to Figure 5, an added image coordinate system (x_2, y_2) is for the vertically mounted Camera #2. The correspondence between this system and the work-space coordinates are: Image coordinates ~ Work-space coordinates: $x_2 \sim x$, $y_2 \sim -y$. Since three orthogonally mounted cameras are used, each component of point q is viewed by two cameras:

x - component viewed by Camera #1 & #2 $\rightarrow Q_{x1}$ &Q_{x2}

y - component viewed by Cameras #0 & #2 $\rightarrow Q_{y0}$ & Q_{y2}

z - component viewed by Cameras #0 & #1 $\rightarrow Q_{z0}$ & Q_{z1}

where Q_{i2} is the i-th component of vector $\underline{q}$ from Camera #2's point of view, $i = x,y$. Now the object distance from Camera #0 has two values: $(d_0 - Q_{x1})$ and $(d_0 - Q_{x2})$, which has an average value of $[d_0 - (Q_{x1} + Q_{x2})/2]$. Thus, for the case of using three cameras, equation (4) is modified as:

$$Q_{z0} = -y_0^*[d_0 - (Q_{x1} + Q_{x2})/2] \tag{14}$$

Likewise, equations (5), (6) and (7) are modified as, respectively,

$$Q_{z1} = -y_1^*\left[d_1 - (Q_{y0} + Q_{y2})/2\right] \tag{15}$$

$$Q_{x1} = -x_1^*\left[d_1 - (Q_{y0} + Q_{y2})/2\right] \tag{16}$$

$$Q_{y0} = x_0^*\left[d_0 - (Q_{x1} + Q_{x2})/2\right] \tag{17}$$

The remaining two relations are derived in a similar manner as:

$$Q_{x2} = x_2^*\left[d_2 - (Q_{z0} + Q_{z1})/2\right] \tag{18}$$

$$Q_{y2} = -y_2^*\left[d_2 - (Q_{z0} + Q_{z1})/2\right] \tag{19}$$

where d_2 is the object distance from Camera #2; x_2^* and y_2^* are defined in a similar manner as in equation (3) with appropriate pixel images in Camera #2. Equations (14) through (19) can be written in a matrix from as:

$$\begin{bmatrix} -2 & 0 & x_1^* & x_1^* & 0 & 0 \\ 0 & 2 & 0 & 0 & x_2^* & x_2^* \\ x_0^* & x_0^* & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 & y_2^* & y_2^* \\ y_0^* & y_0^* & 0 & 0 & -2 & 0 \\ 0 & 0 & y_1^* & y_1^* & 0 & -2 \end{bmatrix} \begin{bmatrix} Q_{x1} \\ Q_{x2} \\ Q_{y0} \\ Q_{y2} \\ Q_{z0} \\ Q_{z1} \end{bmatrix} = 2 \begin{bmatrix} x_1^* d_1 \\ x_2^* d_2 \\ x_0^* d_0 \\ y_2^* d_2 \\ y_0^* d_0 \\ y_1^* d_1 \end{bmatrix} \tag{20}$$

which has the following solution:

$$Q_{x1} = x_1^* \left[\frac{(x_2^* d_2 - 2d_0)(-y_0^* y_2 + 2x_0^*) + (2d_1 + y_2^* d_2)(x_2^* y_0^* + 4)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{21}$$

$$Q_{x2} = x_2^* \left[\frac{(y_1^* d_1 + 2d_2)(x_0^* x_1^* + 4) + (x_1^* d_1 + 2d_0)(-x_0^* y_1^* + 2y_0^*)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{22}$$

$$Q_{y0} = x_0^* \left[\frac{(x_2^* d_2 - 2d_0)(4 - y_1^* y_2^*) + (2d_1 + y_2^* d_2)(x_2^* y_1^* - 2x_1^*)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{23}$$

$$Q_{y2} = y_2^* \left[\frac{(y_1^* d_1 + 2d_2)(x_0^* x_1^* + 4) + (x_1^* d_1 + 2d_0)(-x_0^* y_1^* + 2y_0^*)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{24}$$

$$Q_{z0} = y_0^* \left[\frac{(x_2^* d_2 - 2d_0)(4 - y_1^* y_2^*) + (2d_1 + y_2^* d_2)(x_2^* y_1^* - 2x_1^*)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{25}$$

$$Q_{z1} = -\, y_1^* \left[\frac{(x_2^* d_2 - 2d_0)(-y_0^* y_2^* + 2x_0^*) + (2d_1 + y_2^* d_2)(x_2^* y_0^* + 4)}{8 + 2x_2^* y_0^* - 2y_1^* y_2^* + 2x_0^* x_1^* - x_1^* y_0^* y_2^* - x_0^* x_2^* y_1^*} \right] \tag{26}$$

Finally

$$q_x = \begin{cases} Q_{x1} & \text{by Camera \#1,} \\ Q_{x2} & \text{by Camera \#2,} \end{cases} \tag{27}$$

$$q_y = \begin{cases} Q_{y0} & \text{by Camera \#0,} \\ Q_{y2} & \text{by Camera \#2,} \end{cases} \tag{28}$$

$$q_z = \begin{cases} Q_{z0} & \text{by Camera \#0,} \\ Q_{z1} & \text{by Camera \#1,} \end{cases} \tag{29}$$

Under the same condition stated for the case of using two cameras, the denominators of (21) through (26) cannot be zero. These six equations together with (27) through (29) are the 3-D location formulas for the case of using three cameras.

IV. EFFECT OF NONLINEAR LENSES

The Comsicar 4.8 mm is a wide-angle nonlinear lens which is used in the cameras for the experiments to view the entire work-space at a close distance. Wide-angle lenses expand the image at the center of the lens and compress the image at the perimeter, which introduces distortions that must be corrected and calibrated.

Model for Nonlinear Lenses

The model for nonlinear lenses can be determined by least-squares-fit of experimental data. Because the image function of a physical object should be symmetrical about the center of the lens, the location of the center of the lens in the physical plane is important. This projection of the lens center in the physical plane had to be estimated for the purpose of constructing a model. First, a square object that almost filled the image was placed in front of the camera. The edges of the object were aligned to be parallel with the edges of the image frame (microsensor). The corners of the object were compressed at the corners by the nonlinearity of the lens. Then, the points at which the distortion was at a minimum along each of the four image edges were recorded. The intersection of the up-down and left-right lines through these four points was calculated. The location in the physical plane that corresponded to the pixel at the point of this intersection was used as the projection of the lens center.

As shown in Figure 7, there are four salient points in the image in Camera #j, viz. origin of the manually assigned image coordinates (x_j, y_j), centers of the lens and the image frame (microsensor), and the point of interest. They are related by the vectors

$$\underline{p}_j \doteq \begin{bmatrix} p_{xj} \\ p_{yj} \end{bmatrix}, \quad \underline{f}_j = \begin{bmatrix} f_{xj} \\ f_{yj} \end{bmatrix}, \quad \text{and} \quad \underline{\lambda}_j = \begin{bmatrix} \lambda_{xj} \\ \lambda_{yj} \end{bmatrix}$$

as indicated in Figure 7. Since the model of the distortion effects must be symmetrical about the center of the lens, no even power terms must exist in the distortion model. Let $\underline{p}_j' = \underline{p}_j - \underline{\lambda}_j$. The chosen model is of the form,

$$\text{adjusted radius} = A\,(\text{image radius})^3 + B(\text{image radius}), \tag{30}$$

where

$$\text{image radius} = \left[(p_x')^2 + (p_y')^2\right]^{1/2}, \quad \text{and}$$

$$A \text{ and } B = \text{constants yet to be determined.}$$

The distortion data for the Comsicar 4.8mm lens was obtained with a measuring tape as in the case of obtaining the scaling factor data. Light and black backgrounds with contrasting objects were both used. At first, points were only taken along an axis of an image. But, to accurately represent the extreme effects at the corners of the image, points were also taken near the corners and along the edges of the image. Again, the data was recorded over many different sessions, to allow for the effects of slightly different operating conditions to be averaged into the data. The least-squares-fit to the data of 44 points yields $A = 3.506 \times 10^{-5}$ and $B = 0.87007$.

The adjusted values of p_x' and p_y', which would be the values of p_x' and p_y' if a linear lens were used so that no distortion would exist, can now be computed as

$$\begin{bmatrix}\text{adjusted } p_x' \\ \text{adjusted } p_y'\end{bmatrix} = (\text{adjusted radius}) \begin{bmatrix}\cos\ \theta \\ \sin\ \theta\end{bmatrix} \tag{31}$$

where

$$\theta = \tan^{-1}(p_y'/p_x')$$

However, the trigonometric functions are slow, increase error due to round-off, and require double precision floating point variables. A faster computation can be achieved by using the following formula:

$$\begin{bmatrix}\text{adjusted } p_x' \\ \text{adjusted } p_y'\end{bmatrix} = \frac{\text{adjusted radius}}{\text{image radius}} \begin{bmatrix}p_x' \\ p_y'\end{bmatrix}$$

$$= \{A\left[(p_x')^2 + (p_y')^2)\right] + B\} \begin{bmatrix}p_x' \\ p_y'\end{bmatrix} \tag{32}$$

Error Due to Misalignment of Centers of Lens and Image

The distortion effect caused by the nonlinear lens yields the following observation. Points approaching the perimeter of the lens image are increasingly sensitive to errors in this location. During experimentation with GE TN-2200 camera, the location of the lens center on an image moved in a circle of radius three pixels when the lens was rotated inside the case of its mount. This occurrence demonstrates that the lens center may not coincide with the image center. If the microsensor in a camera is further away from the lens, a larger radius would result.

To illustrate the error caused by the misalignment, consider an example of using two cameras which are $d_0 = d_1 = 36$ inches away from the origin of the work-space (see Figure 5), with $q_x = q_y = q_z = 15$ inches. This value is chosen because it corresponds to the pixel close to the perimeter of the lens image, which is more sensitive to errors. With the 4.8 mm nonlinear lenses, the scaling factor for either camera is 1.264. The pixel width for GE TN-2200 camera is 128. If the lens center in each camera is perfectly aligned with the center of image frame (microsensor), i.e., if $\underline{\lambda}_j = \underline{0}$, the image pixels are $(p_{x0},p_{y0}) = (63,-63)$ and $(p_{x1},p_{y1}) = (-63,-63)$. These values may be obtained by either experiments or computation as follows. By equations (4) through (7), (9), and (10), one obtains $x_0^* = 5/7$, $x_1^* = y_0^* = y_1^* = -5/7$. Using (3) and (8) to yield adjusted $p_{x0} = 72.333$, adjusted p_{x1} = adjusted p_{y1} = adjusted $p_{y0} = -72.333$. Although these values are at the outside of the image field which do not physically exist, they are the adjusted pixels in equation (32). The solution of (32), however, yields the answer $(p_{x0},p_{y0}) = (63,-63)$ which is close to the corner (64,-64). Similarly one obtains $(p_{x1},p_{y1}) = (-63,-63)$.

Supposing these pixels are read from the experiments and one wishes to determine the 3-D location of the point q from the pixels. By (32), one obtains the adjusted pixels $(p_{x0},p_{y0}) = (72.35,-72,35)$ and $(p_{x1},p_{y1}) = (-72.35,-72.35)$. By (8), one computes $x_0^* = 0.71446$, $y_0^* = x_1^* = y_1^* = -0.71446$. Finally one uses two-camera location formulas (11), (12) and (13) to determine $q_x = q_y = 15$ inches and q_z (by Camera #0) = q_z (by Camera #1) = 15 inches. It is seen that no distortion error has occurred.

Now supposing the lens center in each camera does not align with the center of image frame (microsensor). Now the adjusted $\underline{p}_j$ is the pixel values in the image frame (microsensor) corresponding to the point of interest as if a linear lens were used so that there were no distortion caused by the lens nonlinearity. Thus, if $\underline{\lambda}_j \neq \underline{0}$ is introduced, the adjusted $\underline{p}_j\big|_{\underline{\lambda}_j\neq\underline{0}}$ now becomes (adjusted $\underline{p}_j\big|_{\underline{\lambda}_j=0} + \underline{\lambda}_j$). For $\underline{\lambda}_j = \begin{bmatrix}-2\\-2\end{bmatrix}$, the adjusted $\underline{p}_j\big|_{\underline{\lambda}_j\neq 0}$ now has the values: "adjusted $p_{x0} = 72.333 - 2 = 70.333$ and adjusted p_{y0} = adjusted p_{x1} = adjusted $p_{y1} = -72.333 - 2 = -74.333$," so that (32) yields:

$$\begin{bmatrix} 70.333 \\ -74.333 \end{bmatrix} = \left[3.506 \times 10^{-5} (p_{x0}^2 + p_{y0}^2) + 0.87007\right] \begin{bmatrix} p_{x0} \\ p_{y0} \end{bmatrix} \tag{33}$$

$$\begin{bmatrix} -74.333 \\ -74.333 \end{bmatrix} = \left[3.506 \times 10^{-5} (p_{x1}^2 + p_{y1}^2) + 0.87007\right] \begin{bmatrix} p_{x1} \\ p_{y1} \end{bmatrix} \tag{34}$$

These two equations are nonlinear but the symmetry in (34) suggests that $p_{x1} = p_{y1}$ to yield

$$70.12\,(0.01\,p_{x1})^3 + 87.007\,(0.01\,p_{x1}) + 74.333 = 0$$

On solving, one obtains, when rounded off to the nearest integer, $(p_{x1}, p_{y1}) = (-64, -64)$. However, two variables are coupled in equation (33) which makes it difficult to solve. It may be solved by iterative procedure. For this equation, the answer is, after rounded off to integers, $(p_{x0}, p_{y0}) = (62, -64)$. It is seen that when the lens centers for each camera is off set at (-2,-2), the image pixels near the perimeter are shifted from (63,-63) to (62,-64) for Camera #0 and from (-63,-63) to (-64,-64) for Camera #1. Now, using the adjusted pixels values and equations (11), (12) and (13), and following the same procedure as before, one computes $q_x = 16$ inches, $q_y = 14$ inches, q_z (by Camera #0) = 14.45 inches, and q_z (by Camera #1) = 16.09 inches. But physically $p_y = p_y = p_z = 15$ inches. Thus a maximum error of (16.09-15)/(width of review), or 4.11%, is introduced by ignoring the misalignment.

Compensation for Distortion

In reality, pixel values p_{ij} for $i = x,y$ and $j = 0,1,2$, are obtained from the MIC vision system once an object is viewed. They are modified to compensate the appropriate off-set of lens centers. Equation (32) is then applied for each j to determine the corresponding adjusted p_{ij}'s. These computed values are modified again to remove the off-set compensation. Finally, depending on either two or three cameras are used, the

3-D location formulas, equations (11) through (13), or (21) through (29) are invoked to compute the object in work-space, q_k for k = x,y,z, as viewed from different cameras.

Now $\underline{q}$ is defined in 3-D work-space (x,y,z) whose relative location with respect to the center of image frame (microsensor) is predetermined. Refer to Figure 7, $\underline{f}_j$ and $\underline{\lambda}_j$ are determined, and $(\underline{p}_j + \underline{f}_j)$ are produced by the version system. Thus from the adjusted $(\underline{p}_j + \underline{f}_j)$, the physical location of $\underline{q}$ can be determined. Now $(\underline{p}_j + \underline{f}_j) = (\underline{p}_j - \underline{\lambda}_j) + (\underline{f}_j + \underline{\lambda}_j)$ so that

$$\text{Adjusted } (\underline{p}_j + \underline{f}_j) \simeq \text{adjusted } (\underline{p}_j - \underline{\lambda}_j) + \text{adjusted } (\underline{f}_j + \underline{\lambda}_j) \qquad (35)$$

By incorporating the relation between 3-D work-space coordinates (x,y,z) and the assigned image coordinates (x_j,y_j), the adjusted $(\underline{p}_j + \underline{f}_j)$ is then used in the 3-D location formulas to determine $\underline{q}$.

Iterative Algorithm for Calibrating Location of Lens Centers

Instead of using estimation procedure described earlier, the off-set of the lens center may be determined by a combination of an experiment and an iterative computation using equation (32). The process is tedious and time consuming. Fortunately it has to be done only once for each set-up.

To start the procedure, an object is placed in the work-space whose coordinate has already been chosen. The values of q_x, q_y and q_z as well as d_o, d_1 and d_2 are then physically measured. Here it is assumed that all three cameras will be used. For the case of using only two cameras, the modification of the following procedure is straightforward.

Initially the lens center of each camera is assumed to align with its corresponding center of image frame perfectly. One then computes the values of p_{ij} for i=x,y and j=0,1,2 using the steps shown in the previous example. The corresponding adjusted p_{ij}'s are then computed via equation (32); call them $\overline{p_{ij}}$. Finally, (21) through (29) are used to compute q_k for $k=x,y,z$. Note that for the case of using three cameras, there are two computed values for each q_k.

In reality, the lens center of each camera may not align with its corresponding center of image frame (microsensor). But the cameras produce the values of p_{ij} as images, from which the adjusted p_{ij}'s are also computed via (32) and are called $\tilde{p}_{ij}$. Now compare the six computed q_k's with the three physically measured q_k's. If the difference for a q_k is larger than a prespecified value ϵ, then the corresponding $\overline{p_{ij}}$ is corrected by a value δ, i.e., the lens center is corrected by a value δ on the basis of "adjusted p_{ij}". Otherwise the correction process continues for the next q_k. The direction of correction is guided by the manner how $\overline{p_{ij}}$ is deviated from $\tilde{p}_{ij}$. When a lens center shifts in a positive (or negative) direction along an axis, the corresponding coordinate in the image of the same physical point moves in a negative (or positive) direction. When all differences are within the specified ϵ, then ϵ is divided by c_1 and δ by c_2, where c_1 and c_2 are prespecified positive constants that are greater than 1. The iterative process then repeats until δ is not greater than a positive constant DMAX. Thus the lens centers which are determined at the "adjusted" level, are precise to the value of DMAX. The value of DMAX was set to 0.0001 during initial experimentation, but was increased to 0.001 for later experiments to speed up the calibration without seriously reducing the precision. Note that although the stop condition is controlled by DMAX, the comparison is done with the physical measured values of q_k's. This permits a manual judgment on the realistically allowable errors. The initial values for ϵ and δ were determined by trial and error, and were large and conservative. Based on the functional relations defined by equations (11) through (13), (21) through (29) and (32), heuristically set $c_2^2=c_1$; and $c_1=2$ is preferably used in the experiments.

Although the convergence of the iterative procedure is not available, the process does not converge only when $\tilde{p}_{ij}$'s differ from their theoretical values by a large amount.

To avoid the entrapment in an infinite loop caused by this situation, the iterative process is terminated if the number of iterations exceeds the value IMAX. This value is set to 2×10^3.

Since the distortion is most pronounced at the edges of the image corresponding to the perimeter of the lens, points that appear near the edges will produce a more reliable result. The calibration should be performed with as many sets of points as possible. The resultant lens center off-set from each calibrated point, i.e., the difference between $\tilde{p}_{ij}$ and the final $\overline{p_{ij}}$, are averaged. This averaged value is added to all the adjusted p_{ij}'s during the operation performed later to compensate the distortion.

V. DETECTION, LOCATION AND DESCRIPTION OF OBSTACLES

Once the cameras have observed the scene and the 3-D location formulas are applied to compute the sizes of objects, the task of conducting a search for obstacles is ready to start. An initial approach is to divide the work-space into volume cells which are formed by the window grids from each camera. Each cell would either be empty or contain an obstructive object. However, if one takes the full advantage of the camera resolution and employs $128 \times 128 = 16384$ window grids for GE TN-2200 cameras, there will be 2,097,152 volume cells. Even using the window size of two pixels square, $64 \times 64 \times 64 = 262{,}144$ volume cells are required. It involves not only a large size of memory storage, but also a long processing time to describe and then to inform the robot about the obstacles. Thus a different approach for the task is desirable.

There are three separate problems within this task. The first problem is the determination if the obstacles exist. The second is the estimation of the locations of the existing obstacles. And the third involves extracting useful information to form a description of the obstacles. The constraints are time, accuracy, and a meaningful description which serves as an input to the collision avoidance program.

The method presented in this paper requires each camera processing one image, with the MIC vision system retaining information on all the regions of each camera in its memory. Also, some simultaneous processing is performed on the LSI-11 of the MIC vision system and the host DEC VAX 11/780. The host computer can write the MIC vision system a command to take and process a picture. Then, since the image processing time of the MIC vision system is often more than 100 msec., the host computer can do some work before reading the response from the MIC system. The length of saved time cannot be accurately measured. However, the total time consumed by this method, which is called the search algorithm in the following, is relatively short. The method mainly relies on sets of six numbers furnished by the MIC vision system. They are x-min, x-max, x-centroid, y-min, y-max, and y-centroid for each 'blob' found by the camera.

Determination of Existence and Location

After the pictures are taken and the features adjusted, the search algorithm cross-checks the centroids of the regions of each camera with that of the regions of the other cameras. In the following, the set of six numbers for each 'blob' mentioned above are assumed to be the adjusted values so that the distortions and off-set errors have already been compensated. Recall that the 3-D location formulas yield two values for one or all coordinate values, depending on whether two or three cameras are used. If the difference between the two centroid values, of each coordinate, is less than a prescribed constant e, an object is determined to exist. For convenience, the constant e is called the "error margin." Let w_j be the width of view of Camera #j, then the error margin is chosen to be

$$e = \begin{cases} c_3\max(w_1,w_2) & \text{if along x-direction} \\ c_3\max(w_0,w_2) & \text{if along y-direction} \\ c_3\max(w_0,w_1) & \text{if along z-direction} \end{cases} \tag{36}$$

where c_3 is a prescribed percentile fraction. The value of c_3 is determined by trial and error. For the experiments described in the paper, the value between 3% and 5% is used. A smaller search error may introduce the output with some actual objects omitted, and a larger value may introduce the output of objects that do not exist.

The location of the existing object's centroid is estimated to be the average of the calculated centroid coordinate values based on those that are furnished by the MIC system.

Extraction of Description

The description of an obstacle must be a worst case bounding description, i.e., the bounding geometry that encloses the entire obstacle. But the features of each region of the MIC VS-100 vision system restrict the bounding description. From the maximum and minimum x and y coordinate values for each region, a worst case bounding rectangular solid with sides parallel to the axis planes can be constructed. Then, the minimum, centroid, and maximum values of each three dimensional coordinate are output to a file to be read by the collision avoidance program.

First, the worst case adjusted coordinates from the distorted images are calculated. This feature adjustment processing is performed simultaneously while the MIC VS-100 is processing the next picture. The distortion effects are most apparent at the extreme radius values, i.e., corresponding to edges of the image or perimeter of the lens. Thus, which quadrant of the assigned image coordinate (x_j,y_j) for $j=0,1,2$ the centroid is in determines which pairs of minimum and maximum values should be used to calculate the adjusted worst case minimum and maximum values. Figure 8(a) indicates which pairs are used in which quadrants. This determination of the worst case values is only necessary for pictures taken through a nonlinear lens.

Now, the pictures have been taken and the features adjusted. If the centroids of a set of regions are determined to originate from the same object, the worst case three dimensional coordinates at the corners of the rectangular solid are calculated. Figure 8(b) shows the scheme used to label the bounding box points. Due to the perspective qualities of any lens, the points on a side of the box will not be on a plane. The worst case values of the calculated bounding points coordinates are used to describe the box. Figure 9(a) indicates which values from each camera are used to determine the three-dimensional coordinates of the bounding box.

Remember that the xmax value from camera #0 is a worst case value in the positive y-direction. However, the xmax value from camera #1 is a worst case value in the negative x-direction. A maximum x or y value from the cameras does not always coincide with a maximum positive value in the three-dimensional space. Thus the three-dimensional coordinates of each bounding point are determined with the inputs from each camera as in Figure 9(a). Then, the worst case coordinates of each side of the bounding volume are determined. The assumption that the cameras are basically on orthogonal axis eliminates two points on each plane as worst case candidates for the common coordinate. Figure 9(b) depicts which points are used to determine the worst case value for each coordinate. This elimination of two candidates is done to conserve cpu time.

Further simplification can be made in the two camera situation. The removal of camera #2 causes the vertical edges of the box to be parallel with the z-axis. Thus, less coordinate calculations are required.

VI. SYSTEM OPERATIONS

The overall 3-D vision system consists of three software programs in addition to the software that operates the MIC VS-100 vision system. They are the Interface Protocol, the Interprocessor Communications Device Driver, and the Monitor and real-time Search Program. These software programs as well as the real-time operation speed and accuracy of the output information are presented as follows.

Software Programs

The interface protocol is provided by the Machine Intelligence Corporation. The DRV11 and DR11-C hardware allow each machine to generate two kinds of interrupts in the other machine. It may assert CSR0 to cause an "interrupt A" (INTA) or CSR1 to cause an "interrupt B" (INTB). There are 32 data lines, 16 in each direction. To send a one-word message, it is necessary to load that word into the sending machine's output register, then to signal the other machine with INTA or INTB. In the MIC software, INTA is generally used for control and status messages, and INTB is used for data ready or data received.

The Device Driver connects the DEC VAX 11/780 to the MIC VS-100, and allows the MIC vision system to be driven by the VAX. The Device Driver written for this project did not use interrupts. The CSR0 and CSR1 hold their bits until turned off.

The Monitor and Search Program include all the functions to be processed that are described in this paper.

System Initialization

In using the 3-D vision system within a real-time robot environment, the Monitor is first used to initialize the parameters and possibly run a few test searches. Then, all the current parameters are written to a data file. As the robot needs more current information, it calls a real-time Search Program. The real-time Search Program inputs the system parameters, conducts a search for objects, outputs the results, and returns to the calling program. The Monitor and real-time Search Program cannot run simultaneously. However, if the real-time Search Program is not running, the Monitor can be reentered to adjust some parameters.

To operate the 3-D vision system, first load the MIC VS-100 machine vision system with the files on tape. Then enter the "SYSTEM SETUP" mode via the light pen control and use the "DISPLAY CAMERA IMAGE" and "SELECT CAMERA" commands to align the cameras along orthogonal axes. For example, place small objects along an axis and move the camera positions until the objects appear as one object. One may also have to adjust the threshold of each camera.

Next, enter the "SWITCHES" command via light pen control. And turn the "EXTERNAL-COMPUTER-CONTROL" switch on. Return to the MIC logo by selecting the "QUIT" commands.

On the DEC VAX 11/780, execute the vision system Monitor program. Initialize each parameter as the Monitor asks for its value. Put a small object at the origin of the camera coordinate frame. This will be used to select the corresponding location within each image. The distance values will have to be measured by hand. Use the unit of measure that one desires the output to be in. And enter what lens is on each camera.

Next is the optional lens center calibration. Choose some points to calibrate the lens center locations. One must place objects at known locations in the camera coordinates. Then center the cursor over the corresponding locations in each image.

Now that the system has been initialized, try a few sample searches. Two features added to the Monitor allow the user to control the amount of noise while decreasing the threshold level (increasing the sensor sensitivity). The user can adjust the minimum area of a region that is considered a 'blob'. This can be used to have the MIC VS-100 eliminate multi-pixel noise, or to allow the VS-100 to be sensitive to very small objects.

Adjust this parameter if desired. Also, the user can establish a window around an 'active' region of the image. All data outside this region is ignored by the MIC VS-100. This allows a noisy edge of the image to be removed from any processing. The use of active windows also reduces the amount of time used by the MIC VS-100 to process a noisy picture. Hit the 'd' command to store any new window values.

After exiting the Monitor, the Search Program can be called to conduct a search for objects. Unless the VS-100 has been turned off, the parameter file can be used to initialize the system parameters if the Monitor is reentered.

Speed of Real-Time Operation

There are three independent parameters that affect the speed of the search algorithm. The first is the time to process an image within the MIC VS-100 vision system. This time is data-dependent. It can range from 100 milliseconds to one second. Both larger numbers of pixels turned on and more individual regions found within an image contribute to longer image processing times. The second component is the time to transfer data between the LSI-11 of the MIC VS-100 and the host DEC VAX 11-780. However, in this project, this interface has been refined to the point that the time involved can be ignored. The third parameter is the cpu time used by the host DEC VAX 11/780. This is also data-dependent. The larger the number of regions found per camera, the more time the nested loops that check all possibilities consume. And the more objects found, the more worst case boundary calculations that have to be performed. Thus, as more objects enter the work-space, the longer the time before the search routine will output the file of objects found.

Also, the search using three cameras is longer than the search using two cameras. This is because the three-camera search has an extra image to process, many more possible combinations of regions to check, and three-dimensional location formulas that are more time-consuming. Figure 10 shows the VAX 11/780 cpu times and the total elapsed times of the search programs. The data contains values for both the two camera and the three camera search algorithms. Also, the data spans values from no objects to eight objects in the work-space. As a reference, the objects were arranged such that when n objects were found, each image contained n regions. Sometimes, a camera will see regions that do not correspond to objects in the work-space. And sometimes a camera will see one region that corresponds to more than one object behind another. This problem may be eliminated by using an additional camera as demonstrated by an illustrative example shown later.

Precision of Output

The precision of the program's results depends upon two items: the resolution of the image sensors used, and the accuracy of the initializations performed. The resolution of the sensors affects both the accuracy of the initialization procedure and the accuracy of the features calculated during an object search.

During the initialization of the system parameters while using the Monitor, a cursor is used to choose the pixel in each image which represents the origin of the assigned image coordinates. A small object is used to represent this point in space. Figure 11 depicts four situations where the resolution of the object's image will introduce initialization errors. In each situation, the actual centroid location is not one of the pixel centers which must be chosen. The integer value representing a specific pixel will introduce error.

Another problem of this scheme is that the bounding regions may overlap. Figure 12 depicts a situation where this overlap may occur. To describe a larger bounding box would introduce much more wasted space. And to describe the single complex region contained within both boxes would slow the response of the search and greatly increase the complexity of the computation of the output.

Non-Existing Objects

To restore the 3-D environment based on the images from two cameras with the scheme presented in the paper, false scene including non-existing objects may result for some environmental arrangement. As an illustration, consider a scene of four objects as shown in Figure 4. For the purpose of explanation, they are labelled A, B, C, and D as shown in Figure 13. Their physical locations with respect to two horizontal cameras are so arranged that Camera #0 only sees A, C, and D with B hidden behind C; while Camera #1 only sees D, C, and B with A hidden behind C. The Search Program found five objects as indicated in Figure 14. Figure 15 shows the output giving the locations and descriptions of the five objects in reference to the work-space coordinates defined in Figure 16. It gives one more object than physically in existence. Note that in Figure 15, the output data are arranged in the following format:

x-min	x-centroid	x-max
y-min	y-centroid	y-max
z-min	z-centroid	z-max

A brief explanation of the situation is that Camera #0 sees three objects A, C, and D along three rays $\overline{0A}$, $\overline{0C}$, and $\overline{0D}$, respectively, where $\overline{0A}$ is the ray from Camera #0 to object A, etc. Likewise, Camera #1 sees three objects D, C, and B along three rays $\overline{1D}$, $\overline{1C}$, and $\overline{1B}$, respectively. Now, ray $\overline{0D}$ intersects $\overline{1D}$ at D. Ray $\overline{0C}$ intersects $\overline{1C}$ at C and $\overline{1B}$ at B. However, Ray $\overline{0A}$ intersects $\overline{1C}$ at A but intersects $\overline{1B}$ at an imaginary point which yields a non-existing object. The phenomenon of the non-existing object, however, may be eliminated if the image of the vertical Camera #2 is incorporated. As shown in Figure 17, Camera #2 found four regions and the output gives the locations and descriptions of the four objects.

VII. CONCLUSIONS

A 3-D vision system using the Machine Intelligence Corporation VS-100 system has been developed for robotic collision avoidance. As shown in Figure 10, the search program will take less than two seconds to determine eight objects or less while using two cameras, and less than four seconds while using three cameras. However, the total elapsed time is data-dependent. The program could return in one second if no objects are present. Also, note that the DEC VAX 11/780 was in UNIX "normal mode", with three other users in time sharing, during the collection of the data. If the search program is called when the DEC VAX 11/780 is in "real-time mode" with no system loads or other processes running, the response time will be reduced.

The use of three cameras is preferred since otherwise non-existing objects may be found by the program. However, the perception error of detecting objects that do not exist is more favorable than not to detect objects that do exist, for the purpose of collision avoidance. The bounding description will often waste the space surrounding an object. But, for the same purpose, the inclusion of extra space in the boundary is favorable to not including a part of an object. Also, the user of the output must be aware that the output descriptions may overlap in the three-dimensional space.

Again, the accuracy of the scheme is dependent on the accuracy of the initializations performed and the resolution of the sensor. The user affects the accuracy of the scheme by the accuracy of the lens models used, the orthogonality of the camera set-up, the accuracy of the distance measurements, and the accuracy of the cursor positions chosen during the system initialization. As for any vision system, choosing the correct threshold for each camera and properly adjusting the lighting are also important.

ACKNOWLEDGEMENT

The authors wish to acknowledge that George Goble wrote the Interprocessor Communications Device Driver program.

REFERENCES

1. Luh, J. Y. S., "An Anatomy of Industrial Robots and Their Controls," IEEE Transactions on Automatic Control, Vol. 28, No. 2, February 1983, pp. 133-153.

2. Nitzan, D., C. Rosen, et al., *Machine Intelligence Research Applied to Industrial Automation,* SRI International 9th Report, Menlo Park, California, August 1979.

3. Nevatia, R., *Machine Perception,* Prentice Hall, 1982, pp. 158-167.

4. *TN-2200/2201 Solid-State Automation Camera, Literature EHM-12408/5000,* Optoelectronic Systems Operations, General Electric, Syracuse, New York.

5. Ballard, D. H. and C. M. Brown, *Computer Vision,* Prentice Hall, 1982, p. 50.

6. Rosenfeld, A., "Connectivity in Digital Pictures," Journal of ACM, Vol. 17, January 1970, pp. 146-170.

7. Agin, G. J., "Computer Vision Systems for Industrial Inspection and Assembly," Computer (IEEE Computer Society), Vol. 13, May 1980, pp. 11-20.

8. *VS-100 Machine Vision System Reference Manual, Version 1.33,* Machine Intelligence Corporation, Sunnyvale, California, 1980.

9. Gleason, G. J. and G. J. Agin, "The Vision Module Sets Its Sight on Sensor-Controlled Manipulation and Inspection," Robotics Today, Winter 1980-1981, pp. 36-40.

10. Yakimovsky, T. and R. Cunningham, "A System for Extracting Three-Dimensional Measurements from a Stereo Pair of TV Cameras," Computer Graphics and Image Processing (Journal), Vol. 7, No. 2, Academic Press, April 1978, pp. 195-210.

11. Thompson, A., "Camera Geometry for Robot Vision," Robotics Age, Vol. 3, No. 2, March/April 1981, pp. 20-27.

12. *UNIX Time-Sharing System: UNIX Programmer's Manual, 7th Edition,* Bell Telephone Laboratories, Inc., 1979.

13. Kernigham, B. W. and D. M. Ritchie, *The C Programming Language,* Prentice Hall, 1978.

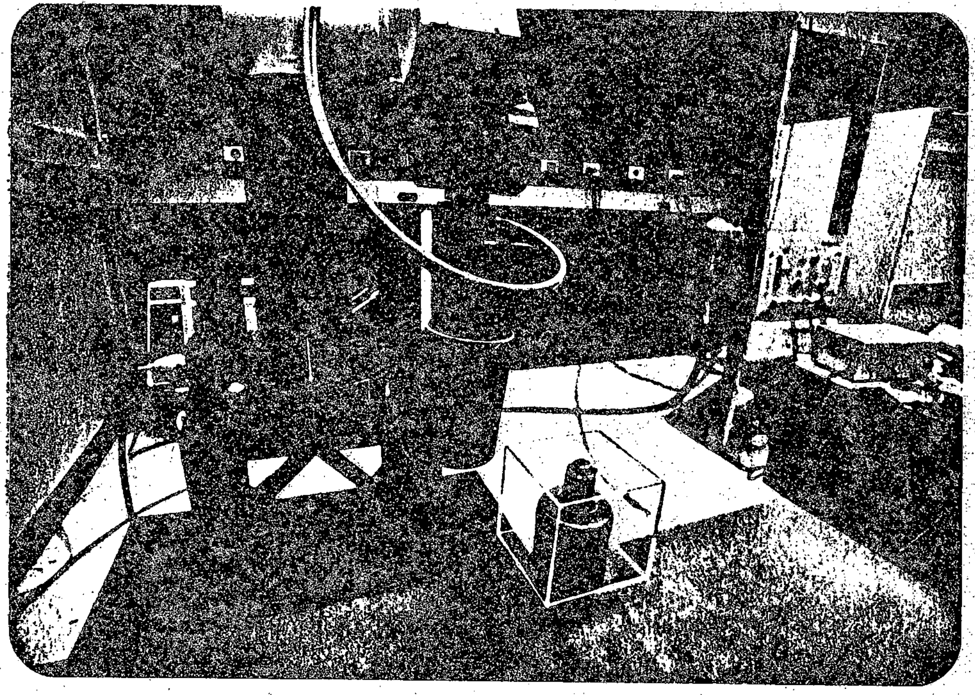


Figure 1. Bonding Box Representation of Obstacle.

OPERATION PROCESSING TIMES	
General Operation Processing Time	Measured Processing Time
Read in the image and process it	
• Using software run length encoding 484 ms + 0.97 ms/segment	583 ms
• Using hardware run length encoding 267 ms + 0.53 ms/segment	321 ms
Connectivity analysis 66 ms + 1.8 ms/segment + 3.3 ms/blob	270 ms
Perimeter accumulation 0.86 ms/segment + 2.3 ms/blob	102 ms
Accumulation of second moments 2.3 ms/segment	235 ms
Perimeter and radius calculation 5.2 ms/segment	500 ms
Note: Blob = connected component	

Figure 2. Operation Processing Times for MIC VS-100 Vision System.

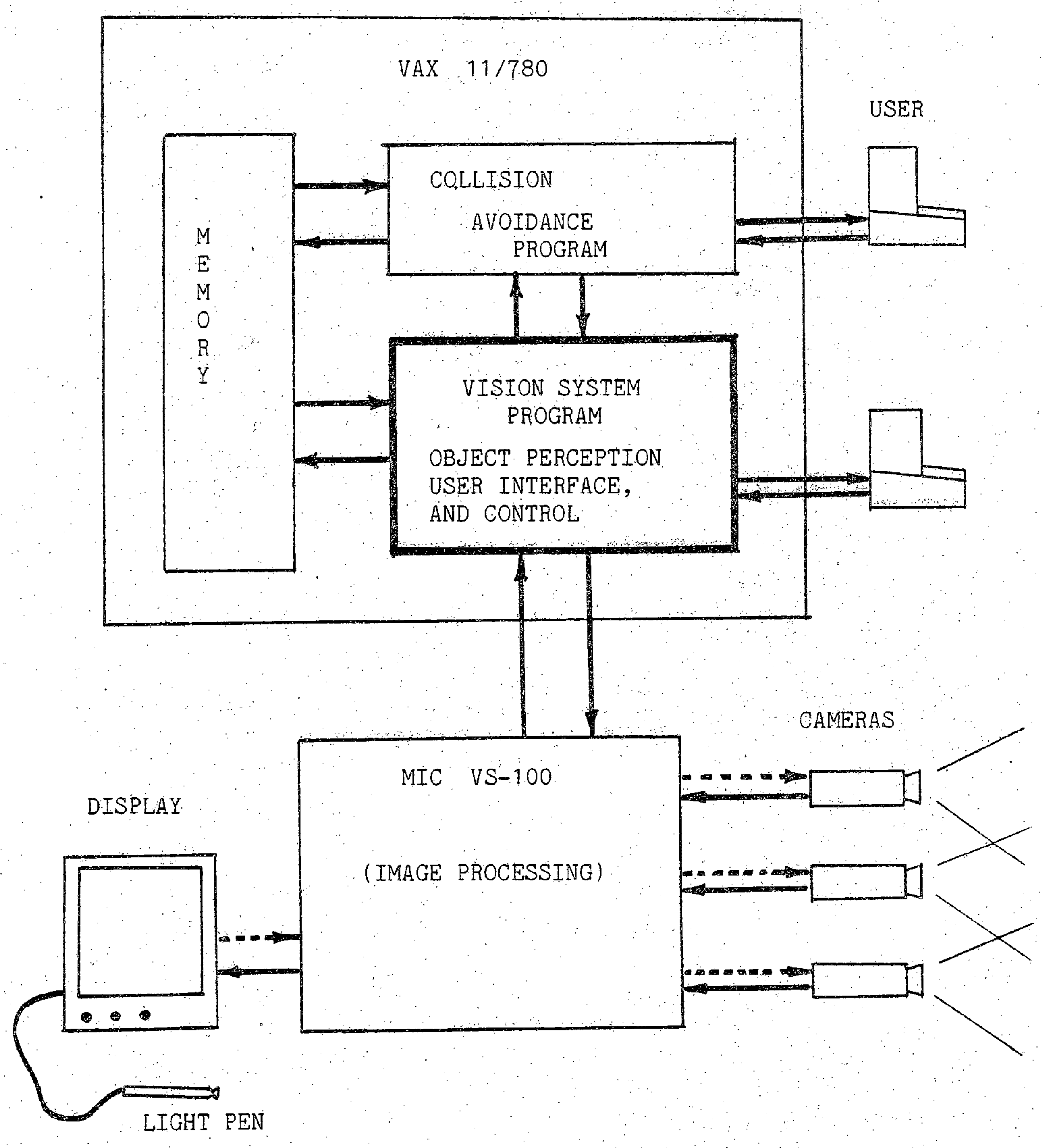


Figure 3. Schematic Diagram for the Overall Vision System

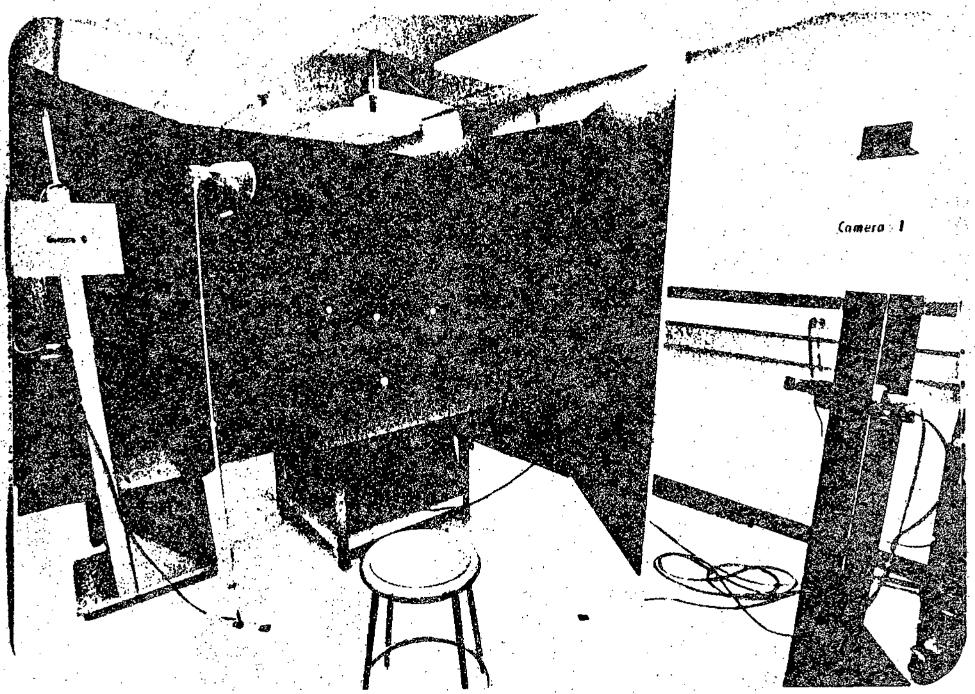


Figure 4. Environmental Arrangement for Experiments.

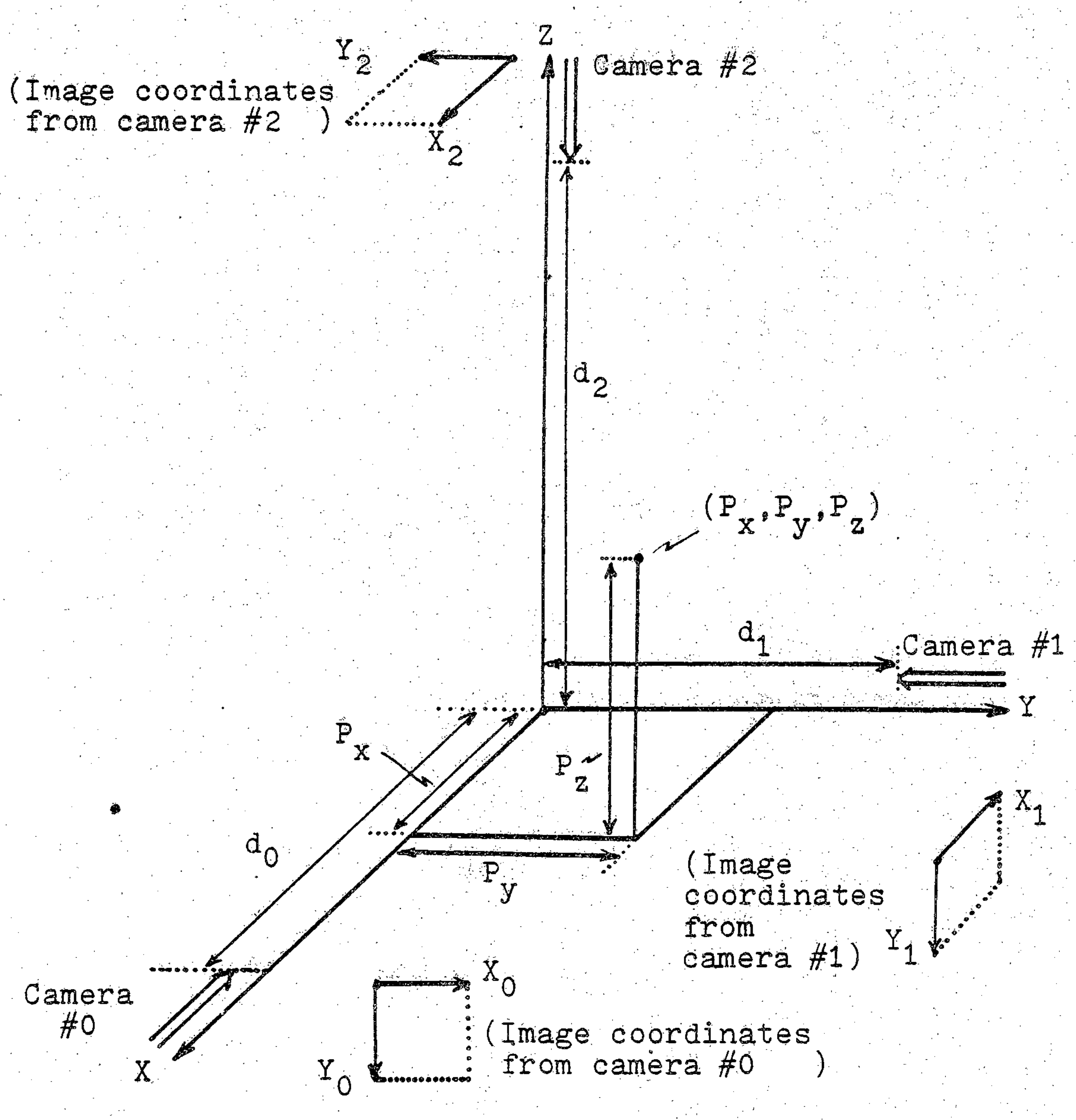


Figure 5

Camera Orientations

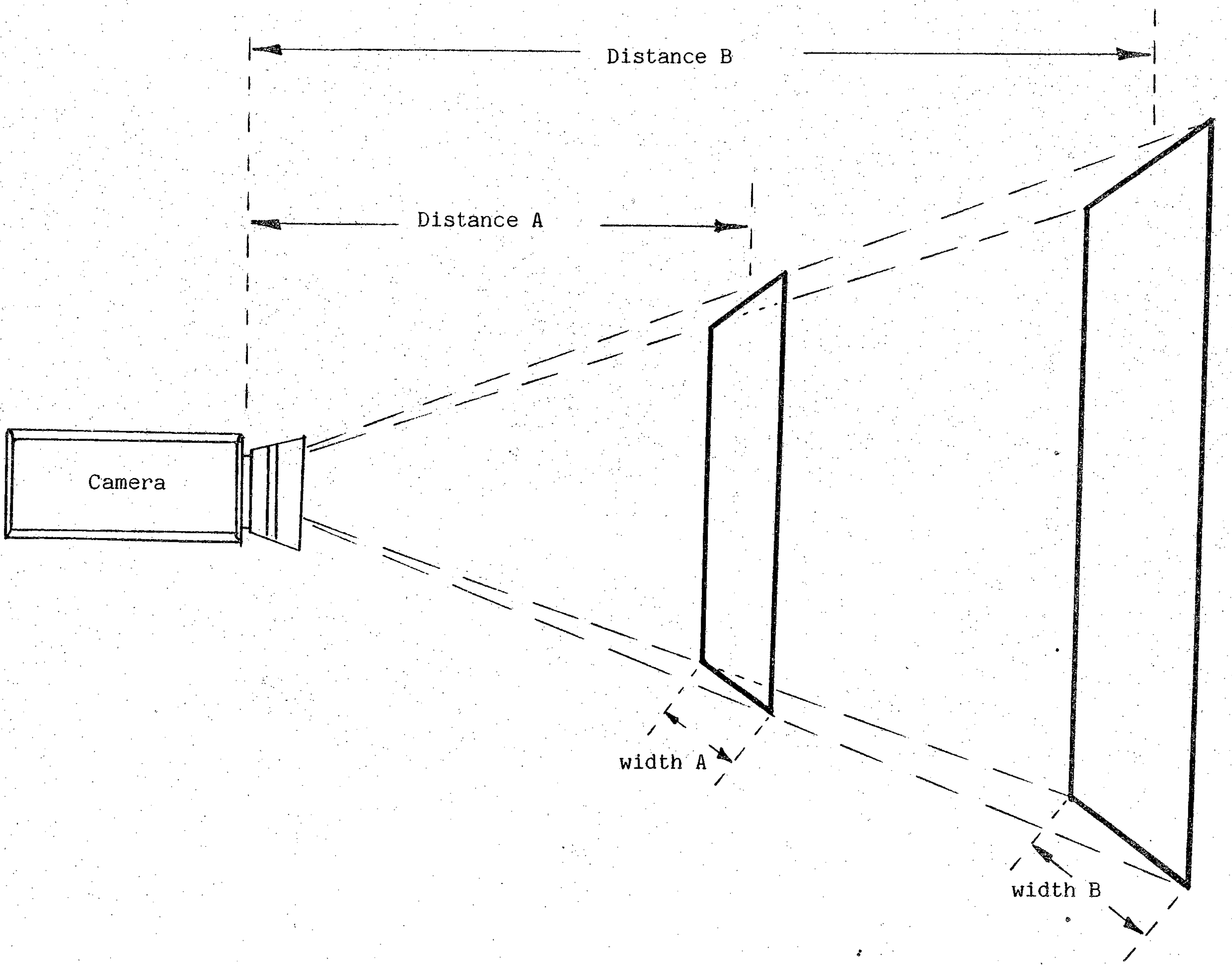


Figure 6. Measurements for Determining Camera Scaling Factor

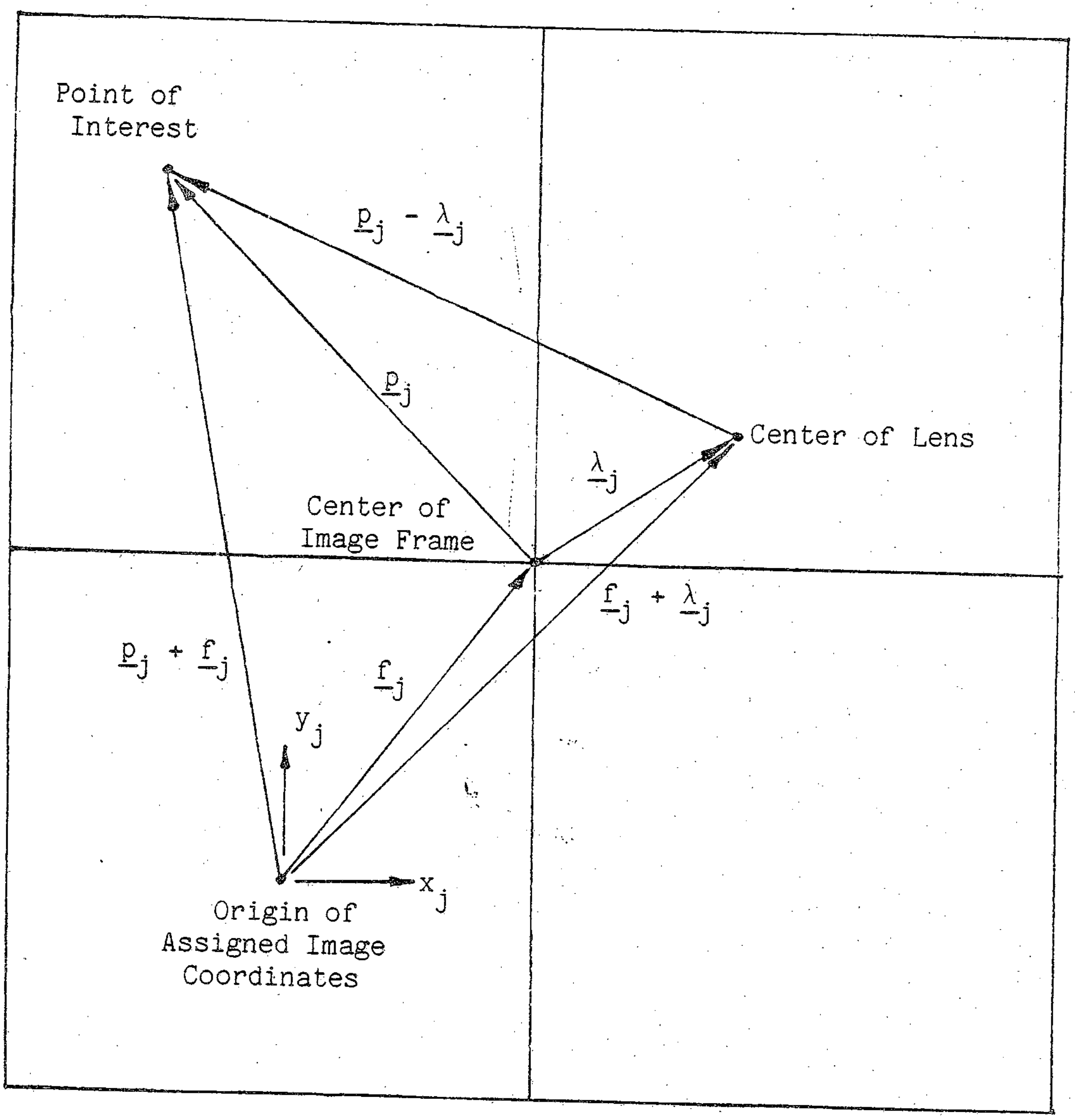

Point of Interest
$\underline{p}_j - \underline{\lambda}_j$
$\underline{p}_j$
Center of Lens
$\underline{\lambda}_j$
Center of Image Frame
$\underline{f}_j + \underline{\lambda}_j$
$\underline{p}_j + \underline{f}_j$
$\underline{f}_j$
y_j
x_j
Origin of Assigned Image Coordinates

Figure 7. Four Salient Points in Image in Camera #j

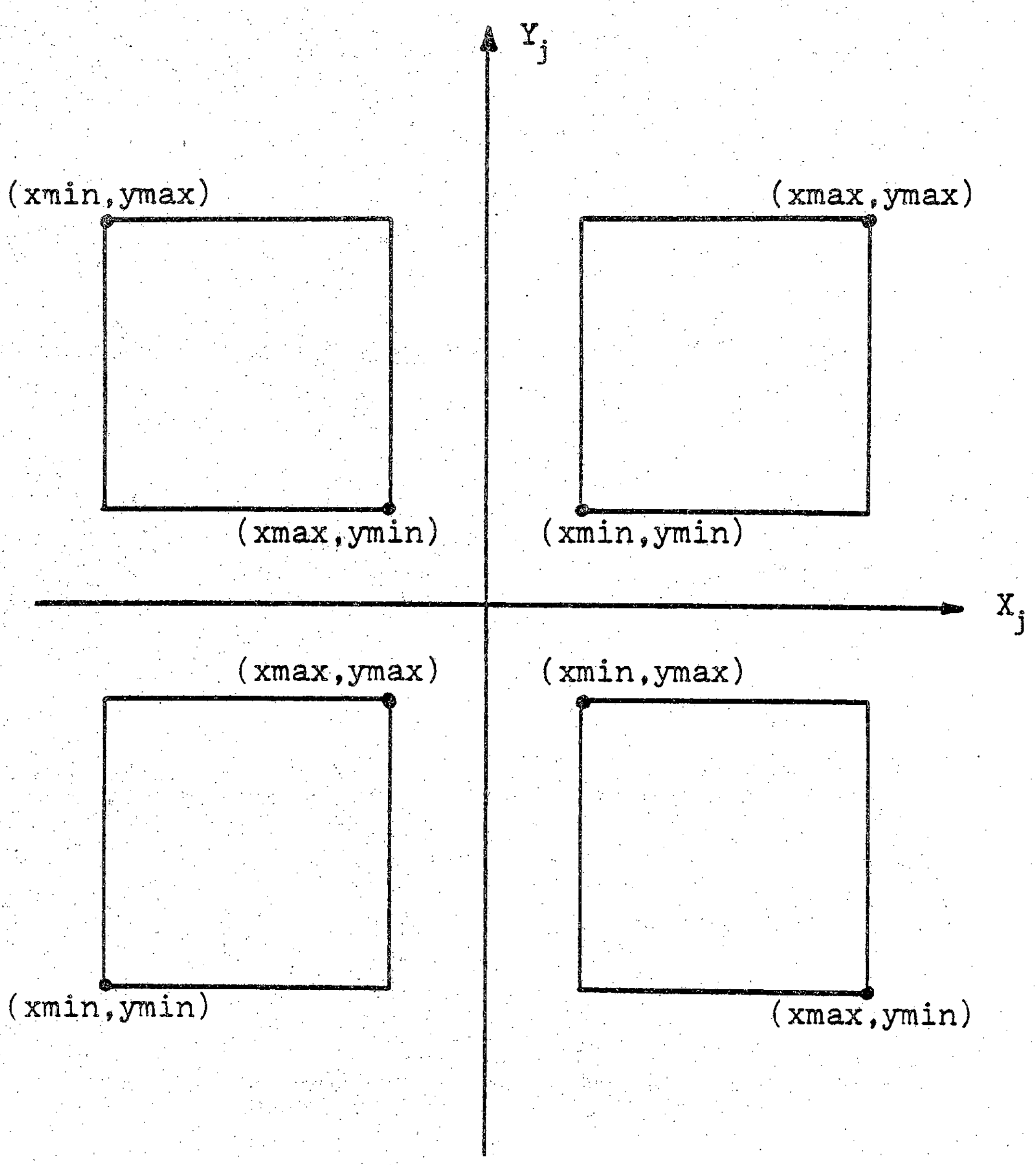


Figure 8(a)

Worst Case Pairs within a Distorted Image

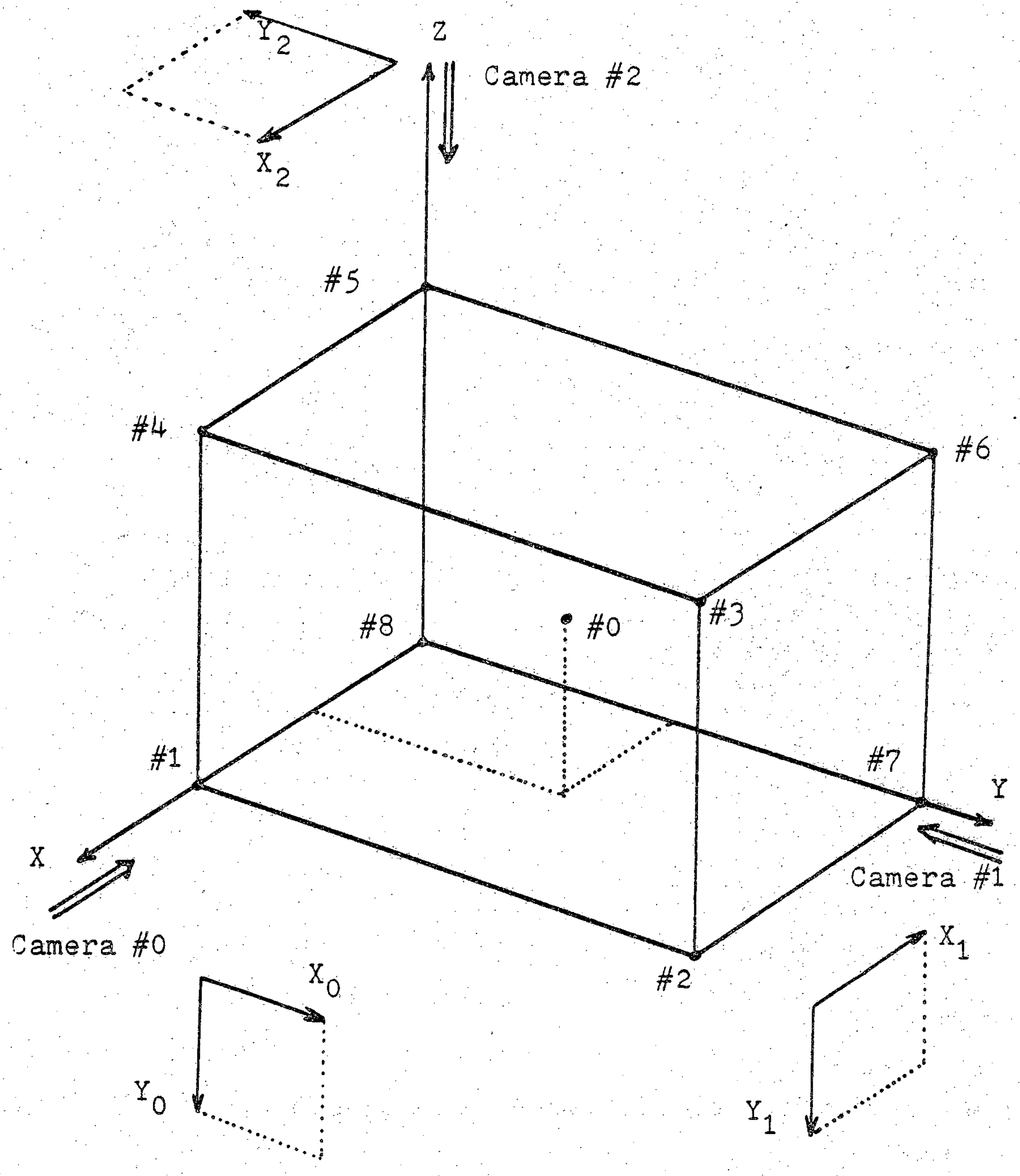


Figure 8(b)

The Bounding Box

Bounding point number (see Fig. 8)	Camera #0 features		Camera #1 features		Camera #2 features	
	x_0	y_0	x_1	y_1	x_2	y_2
0	xcnt	ycnt	xcnt	ycnt	xcnt	ycnt
1	xmin	ymax	xmin	ymax	xmax	ymax
2	xmax	ymax	xmin	ymax	xmax	ymin
3	xmax	ymin	xmin	ymin	xmax	ymin
4	xmin	ymin	xmin	ymin	xmax	ymax
5	xmin	ymin	xmax	ymin	xmin	ymax
6	xmax	ymin	xmax	ymin	xmin	ymin
7	xmax	ymax	xmax	ymax	xmin	ymin
8	xmin	ymax	xmax	ymax	xmin	ymax

(a) Bounding Box Information

Worst Case Coordinate of the bounding box	Candidate Points from the bounding box (see Figure 8)
xmin	6 and 8
xmax	1 and 3
ymin	4 and 8
ymax	3 and 7
zmin	2 and 8
zmax	3 and 5

(b) Bounding Box Coordinates

Figure 9. Construction of Bounding Box

Number of Objects	Vax cpu time		Total elapsed time	
	2 Cameras	3 Cameras	2 Cameras	3 Cameras
0	0.0417	0.0485	0.743	1.163
1	0.0500	0.0750	0.959	1.381
2	0.0733	0.1284	1.155	1.675
3	0.0883	0.1750	1.295	1.832
4	0.1150	0.2600	1.400	1.979
5	0.1583	0.3800	1.543	2.301
6	0.1683	0.5383	1.694	2.995
7	0.1933	0.7500	1.718	3.157
8	0.2017	1.0016	1.907	3.342

Notes: These values are the averages of ten samples. The data was obtained when there were three other users on the Vax, causing a load average of 1.5 to 4.5. The Vax cpu time data is only accurate to 1/60th of a second. The total elapsed time data was measured with a stop watch, after 10 calls to the search routine, and measured to 1/100th of a second.

Figure 10. System Execution Times

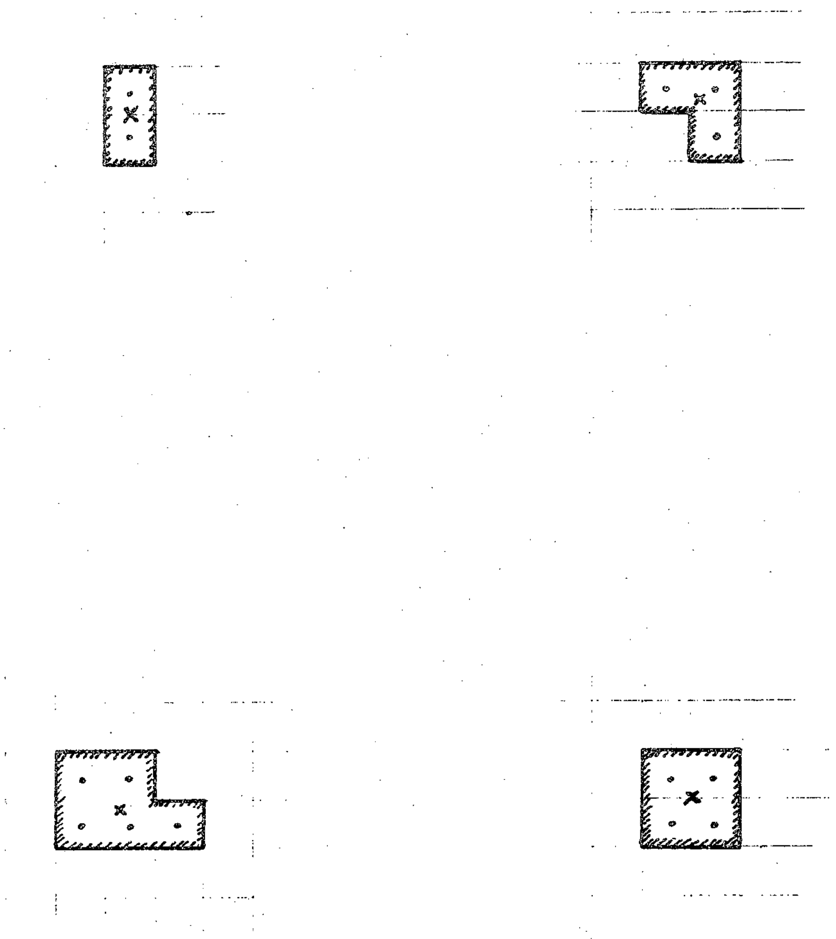


Figure 11

Pixel Selection Errors

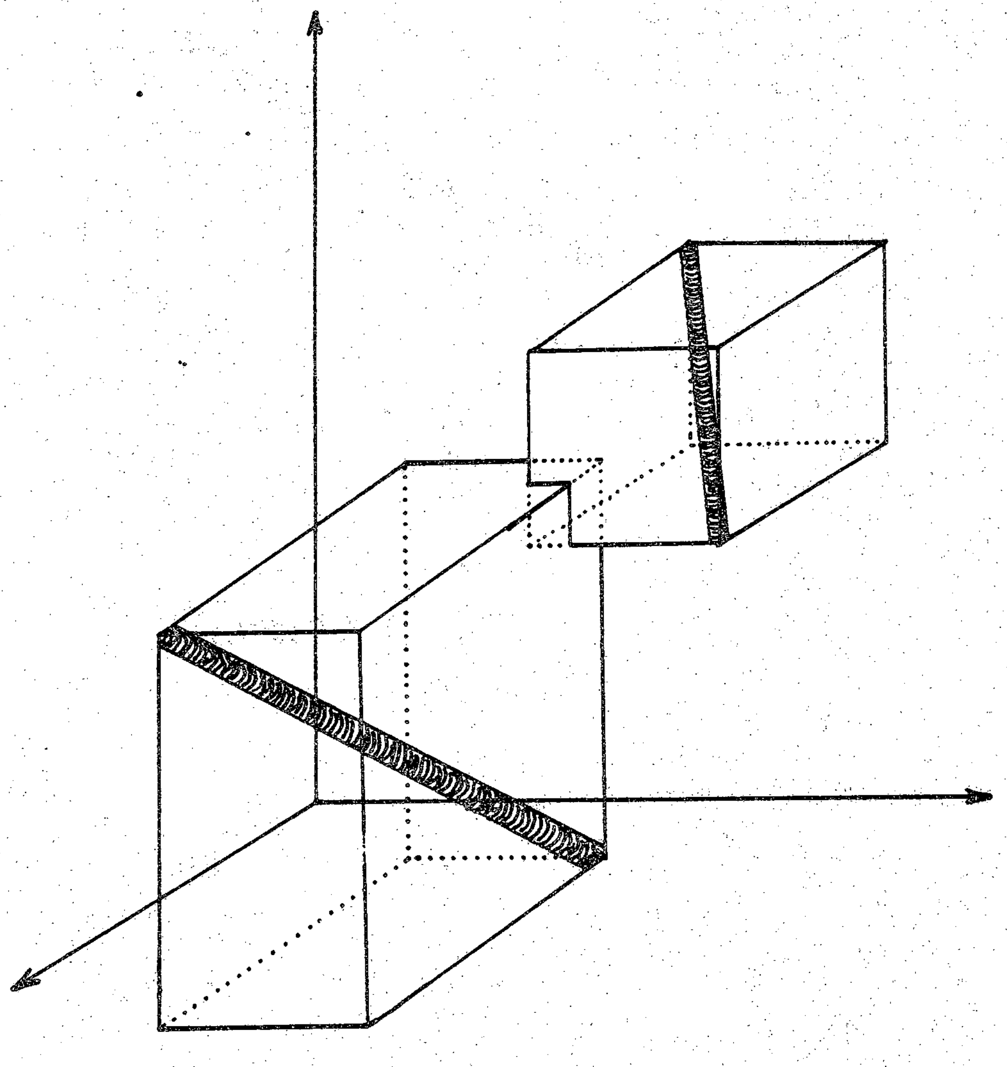


Figure 12

Boundary Description Overlap

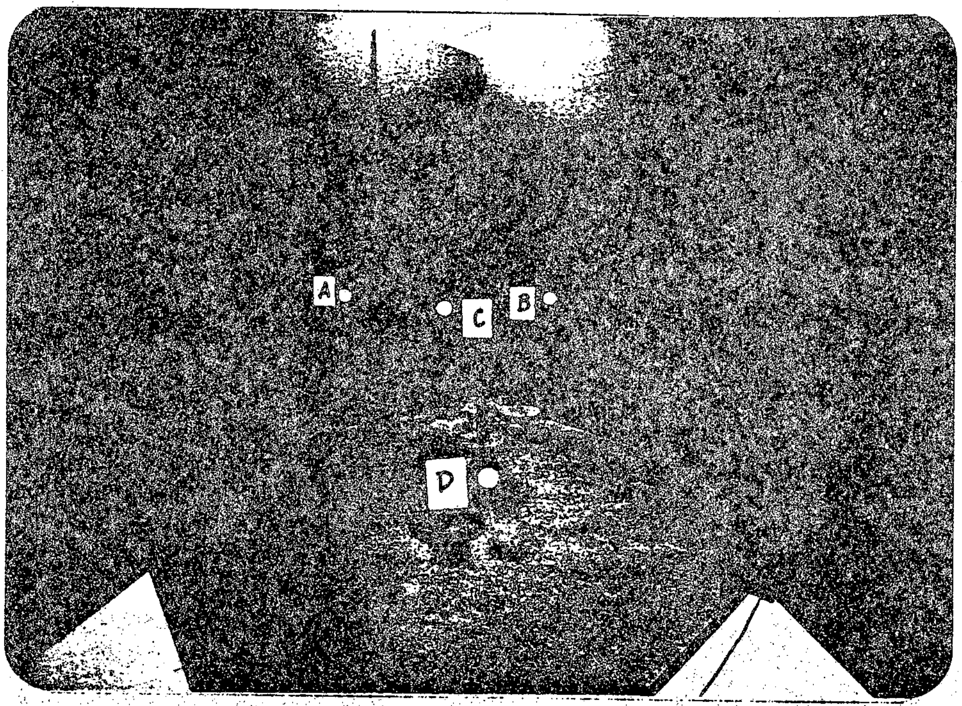
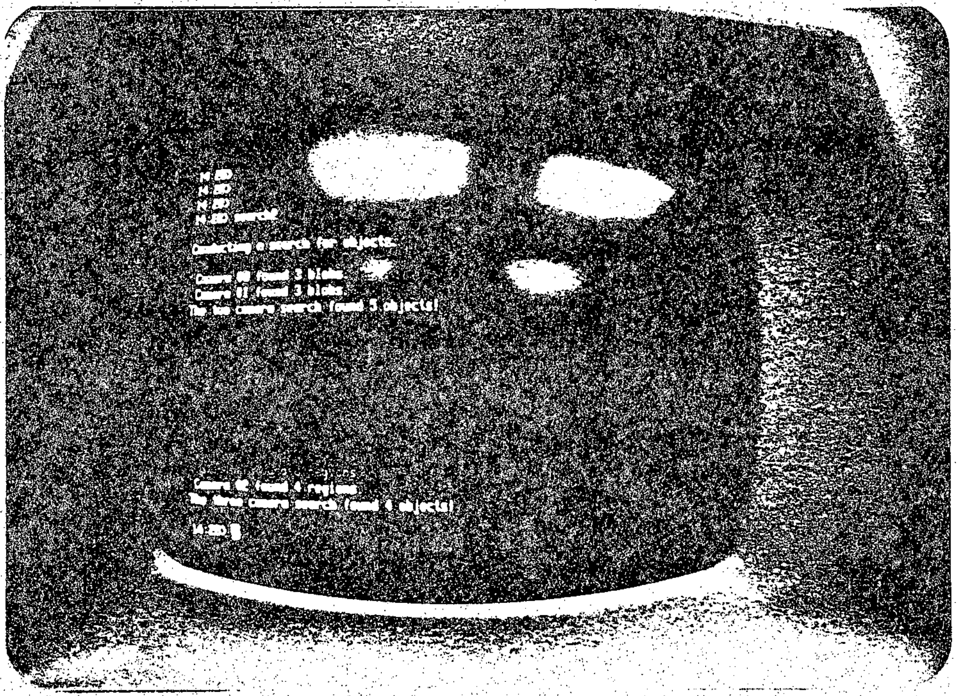


Figure 13. Labels for the Four Physical Objects



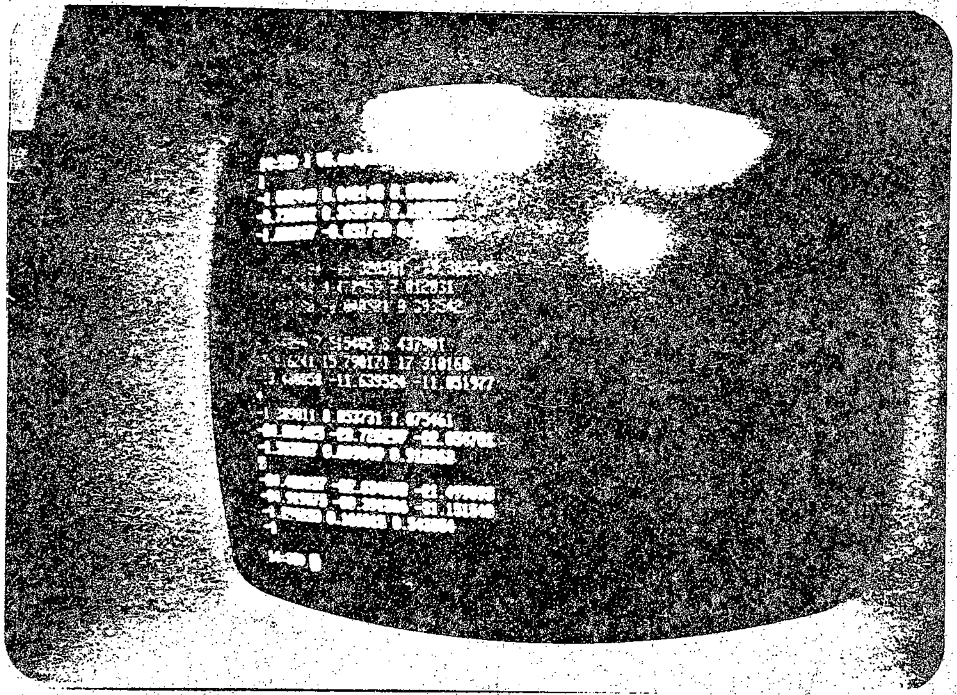
```
M-80
M-80
M-80
M-80 search
Conducting a search for objects.

Camera 00 found 3 blobs.
Camera 01 found 3 blobs.
The two camera search found 5 objects!

Camera 02 found 4 regions.
The three camera search found 4 objects!

M-80
```

Figure 14. Indicating Five Objects Are Found by the Search Program



```
[illegible]
1
[illegible]
[illegible]
[illegible]

[illegible] -15.326591 -15.382475
[illegible] 4.472953 2.812031
[illegible] -7.648921 9.593542

[illegible] 7.515405 8.437901
[illegible] 624] 15.790171 17.318168
-7.68858 -11.539524 -11.851977

-1.390811 0.853731 1.675461
[illegible] -22.78[illegible] [illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

Figure 15. Description of Bounding Boxes for Five Objects Found

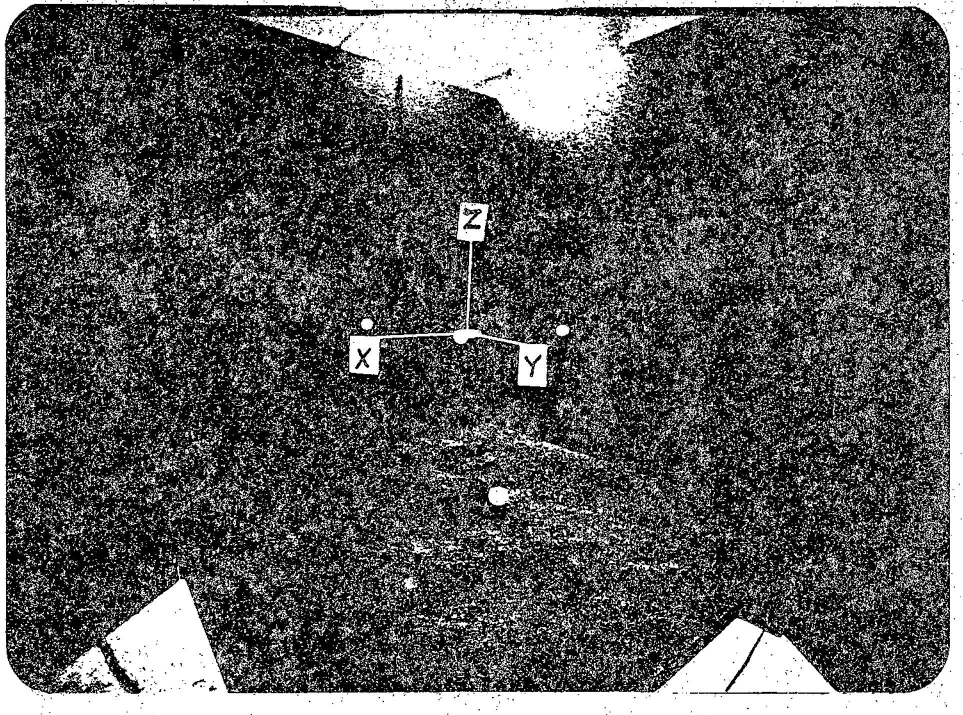
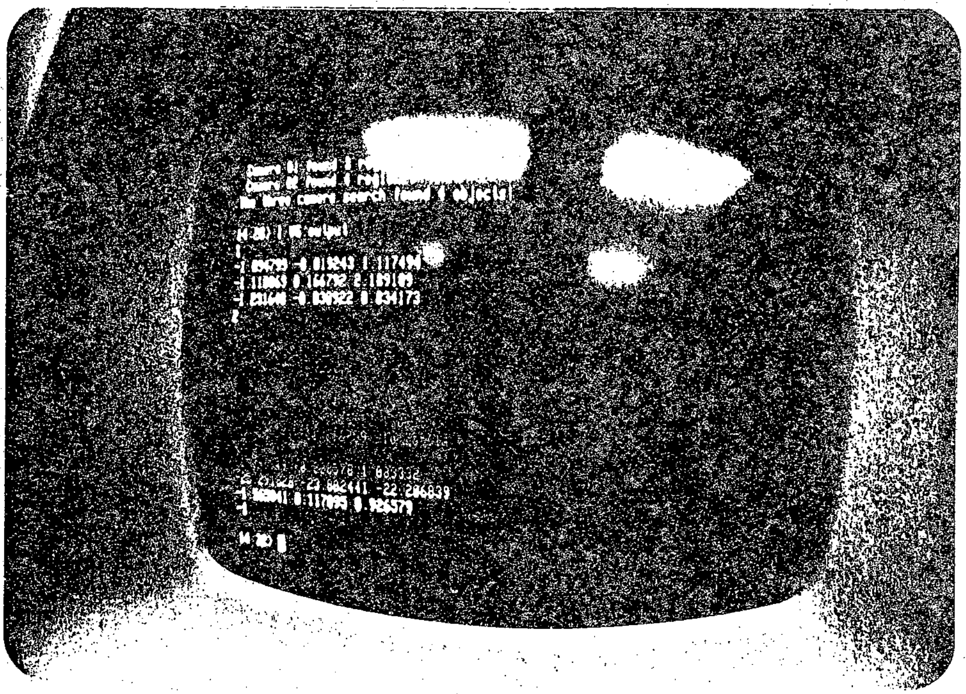


Figure 16. Work-space Coordinates and the Four Objects



```
Camera #1 found 4 r[illegible]
Camera #2 found 4 reg[illegible]
The three camera search found 4 object(s)

(4-20) 1 ME output
(
-1.09479 -0.019243 1.117498
-1.11063 0.144792 2.189109
-1.23164 -0.030922 0.834173
2

[illegible] 0.34978 1.683332
-25.47123 -23.082441 -22.286839
-1.908041 0.117095 0.926579
-1

(4-20) █
```

Figure 17. Description of Bounding Boxes for Four Objects Found

LAGRANGIAN FORMULATION OF ROBOT DYNAMICS WITH DUAL-NUMBER TRANSFORMATION FOR COMPUTATIONAL SIMPLIFICATION *

J.Y.S. Luh and Y.L. Gu
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

Among a variety of formulations of robot dynamics, the Lagrangian equation yields an insight of the robot behavior but suffers from the excessive computational complexity. The dual-number transformation is capable of transforming velocities from one coordinate frame to another. By incorporating dual-number transformation into the Lagrangian equation, it is possible to apply the differential operator directly so that the computation of the joint torques/forces of the robot is simplified.

I. INTRODUCTION

The industrial robots are computer controlled mechanical manipulators used in industrial applications. They have serial link mechanisms whose dynamic behavior can be described by equations in Lagrangian formulation as:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i , \quad i = 1,2,\dots,n . \tag{1}$$

where q_i = generalized coordinates

$L = L(q_1,\dots,q_n,\dot{q}_1,\dots,\dot{q}_n)$ = Lagrangian ,

*Supported by NSF Grant MEA-8119884.

τ_i = generalized forcing function .

The generalized coordinate q_i represents the displacement of joint i. The Lagrangian is also defined as L = K - P where K and P are, respectively, the kinetic energy and potential energy of the system. By applying the Lagrangian equation to a robot with n joints (or (n+1) links), one obtains [1,2]:

$$\tau_i = \sum_{j=1}^{n} D_{ij}\ddot{q}_j + J_{ai}\ddot{q}_i + \sum_{j=1}^{n} D_{ijj}(\dot{q}_j)^2 + \sum_{\substack{j=1 \\ j\neq k}}^{n} \sum_{k=1}^{n} D_{ijk}\dot{q}_j\dot{q}_k + D_i \tag{2}$$

where

$$D_{ij} = \sum_{p=\max(i,j)}^{n} \mathrm{Tr}[U_{pj}\, J_p (U_{pi})']\tag{3}$$

$$D_{ijk} = \sum_{p=\max(i,j,k)}^{n} \mathrm{Tr}[U_{pjk}\, J_p (U_{pi})']\tag{4}$$

$$D_i = - \sum_{p=i}^{n} m_p \hat{g}'\, U_{pi}\, \hat{r}_p\tag{5}$$

Tr =	trace operator,
( )' =	transpose of ( ),
τ_i =	input generalized force for joint i,
m_p =	mass of link p,
$\hat{r}_p$ =	a vector describing the center of mass of link p with respect to p-th coordinate system,
$\hat{g}'$ =	[0, 0, 9.8 m/sec^2, 0] is a gravitational acceleration vector at a sea level base,

$\mathbf{J}_p =$ inertia matrix for link p,

$$\mathbf{U}_{pj} = \frac{\partial \mathbf{T}_o^p}{\partial q_j} = \begin{cases} (\mathbf{T}_o^{j-1})\, \mathbf{Q}_j\, (\mathbf{T}_{j-1}^p), & \text{for } p \geq j\,, \\ 0 & \text{,otherwise}\,, \end{cases} \tag{6}$$

$$\mathbf{U}_{pjk} = \frac{\partial^2 \mathbf{T}_o^p}{\partial q_j \partial q_k} = \begin{cases} (\mathbf{T}_o^{j-1})\, \mathbf{Q}_j\, (\mathbf{T}_{j-1}^{k-1})\, \mathbf{Q}_k\, (\mathbf{T}_{k-1}^p)\,, & \text{for } p \geq k \geq j\,, \\ (\mathbf{T}_o^{k-1})\, \mathbf{Q}_k\, (\mathbf{T}_{k-1}^{j-1})\, \mathbf{Q}_j\, (\mathbf{T}_{j-1}^p)\,, & \text{for } p \geq j \geq k\,, \\ 0 & \text{, otherwise,} \end{cases} \tag{7}$$

$$\mathbf{Q}_j = \begin{cases} \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & \text{if joint j is rotational,} \\ \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}, & \text{if joint j is translational,} \end{cases} \tag{8}$$

$\mathbf{T}_j^k =$ 4 x 4 matrix which transforms any vector expressed in k-th coordinate system to the same vector expressed in j-th coordinate system.

$q_k =$ generalized coordinate (i.e., joint displacement).

For each joint i, the required torque or force is divided into five groups as shown in (2). The first group represents the contribution from inertias of all the joints. The second term represents the inertia torque of the actuator of joint i. The third and fourth groups in (2) are the contributions from, respectively, the centrifugal term and the Coriolis force, while the last term is resulted from the gravitational acceleration. Whether equation (2) is utilized to solve forward dynamics problem for analysis and simulation (i.e., solve for q_j's and their time-derivatives for given τ_i's), or to solve inverse dynamics problem for control of robots (i.e., solve for τ_i's for desired q_j's and their derivatives), one must compute the coefficients D_{ij}, D_{ijk} and D_i that are defined by

(3), (4) and (5) respectively. The computation of these terms is, unfortunately, very complicated and time consuming. It involves an evaluation of thousands of trigonometrical terms [3]. Obviously it is not a simple computational task especially when the position-dependent and orientation-dependent parameters change as the robot moves. Therefore it warrants the effort of searching for methods of simplifying the computation.

Efficient algorithms for computing τ_i have been developed by various authors during the past three years. Luh, Walker and Paul [4] computed the joint forces/torques based on the Newton-Euler formulation. Walker and Orin [5] extended the approach to compute the joint accelerations which were then used in the simulation of the robot control scheme. Hollerbach [6] developed recursive algorithms based on the Lagrangian formulation which were shown to be equivalent to the Newton-Euler method [7]. Recently Kane and Levinson [8] used specialized formulation for specific robots, while Featherstone [9] approached the problem differently by using articulated-body inertias. All the methods mentioned above are very efficient in producing numerical solutions. However, they will yield very little insight views of the dynamical behavior of the robot. To analyze the dynamics of the robot for full understanding and aiding in designing new robots, it is desirable to simplify the computation of the coefficients D_{ij}, D_{ijk} and D_i and then deal with the differential equation (2) directly.

There are three known approaches of simplification, viz. geometric/numeric, composite, and differential transformation. Bejczy's geometric/numeric evaluation [10,11] deals with the nature of joints whether it is revolute or prismatic. Thus the 4 by 4 homogeneous transformation matrices $\mathbf{T}_j^k$ in (6), (7) and (8) can be simplified in advance. Since many elements in the matrices are zeros, the resulting expressions for D_i, D_{ij} and D_{ijk} are less complicated [10,11]. The composite technique by Luh and Lin [12] involves the comparison of all the terms in Newton-Euler formulation of the dynamic equation [4] in a computer. Some of the terms may be eliminated under various criteria. The remaining terms are then rearranged in a Lagrangian formulation. The upshot is a computer output of a simplified equation in symbolic form. Paul's differential transformation [2] which converts $\partial T_o^p/\partial q_j$, the partial derivative of the homogeneous transformation matrices, into the matrix product of the transformation and a differential matrix which reduces D_{ij} to a much simpler form. However, the term D_{ijk} contains a second order partial derivative $\partial^2 T_o^p/(\partial q_j \, \partial q_k)$ which was not simplified until recently by Bejczy and Lee [13]. Their approach is to apply the differential operator used by Paul, successively at the appropriate link-to-link coordinate transformations. An alternative approach is to adopt the dual-number algebra and screw calculus in the analysis instead of the homogeneous transformation.

In screw calculus [14,15], a vector may be represented by either six real numbers, or three dual numbers. The associated coordinate transformation matrices perform line transformations, which is different from the point transformation by homogeneous transformation. In robotics, this approach has been investigated by Pennock and Yang [16], and Featherstone [9]. As shown by Rooney [17], the dual-number representation is most concise, while the real 6 by 6 matrix representation contains redundant components since not all conditions that form the matrix are independent. The size of the 6 by 6 matrix gives an intuitive impression of excessive computational burden. Yet the dynamical analyses are done by the real 6 by 6 matrix representation in [9] and [16] because it is not feasible to express the inertia directly in dual-numbers.

This paper presents a method of expressing the kinetic energy of the system in terms of dual-number transformations so that the analysis of the dynamics using dual-number algebra is possible. The method is different from the momentum approach by Yang [18]. Because of the property of line transformation, the dual-number transformation may deal with dual-velocity vectors. Thus the differential transformation in the kinetic energy term yields only the first order partial derivatives in D_{ijk} so that Paul's simplification approach [2] applies. Although there is no first order partial derivatives in D_{ij} in the dual-number representation, the computation of D_{ij} is still simpler than that by Paul's simplified representation [2]. The computational efficiency of the dual-number representation is exhibited by comparing the numbers of required multiplications and additions for computing the joint torques/forces τ_i for all n joints, with those numbers required when the direct homogeneous transformation [1], and Paul's simplified homogeneous transformation [2] methods are applied.

II. MOVING RIGID BODY AND KINETIC ENERGY

Consider a moving rigid body L_j with its center of mass at point G_j, and an arbitrary point at O_j upon which a coordinate system $(\mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j)$ is attached as shown in Figure 1. The radius vector from O_j to G_j is denoted by $\mathbf{c}_j$. Let $\mathbf{v}_j$ and ω_j be, respectively, the linear and angular velocity of coordinate system $(\mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j)$ with reference to the base coordinates $(\mathbf{x}_o, \mathbf{y}_o, \mathbf{z}_o)$. Then the linear velocity of the center of mass G_j is [16,18]:

$$\mathbf{v}_{Gj} = \mathbf{v}_j + \omega_j \times \mathbf{c}_j \tag{9}$$

where $\times$ denotes the cross-product. The kinetic energy of the moving body L_j is

$$K_j = \frac{1}{2}(m_j \mathbf{v}'_{Gj}\mathbf{v}_{Gj} + \omega'_j \mathbf{J}_{Gj}\omega_j) \tag{10}$$

where m_j is the mass of the body L_j, $\mathbf{J}_{Gj}$ the 3 by 3 inertia matrix of the body about its center of mass in $(\mathbf{x}_o, \mathbf{y}_o, \mathbf{z}_o)$, and $(\ )'$ the transpose of $(\ )$. Let

$$\mathbf{c}_j = \begin{bmatrix} c_{xj} \\ c_{yj} \\ c_{zj} \end{bmatrix} \quad \text{and} \quad \mathbf{C}_j = \begin{bmatrix} 0 & -c_{zj} & c_{yj} \\ c_{zj} & 0 & -c_{xj} \\ -c_{yj} & c_{xj} & 0 \end{bmatrix} = -\mathbf{C}'_j \tag{11}$$

be the antisymmetric matrix whose components are the combinations of those of $\mathbf{c}_j$. Then

$$C_j' \omega_j = \omega_j \times c_j \tag{12}$$

Combine (9), (10) and (12) to yield

$$K_j = \frac{1}{2} [m_j(\mathbf{v}_j' \mathbf{v}_j + 2\omega_j' C_j \mathbf{v}_j) + \omega_j' (J_{Gj} + m_j C_j C_j')\omega_j] \tag{13}$$

In Appendix A, it is shown that

$$\mathbf{J}_j = \mathbf{J}_{Gj} + m_j C_j C_j' \tag{14}$$

where $\mathbf{J}_j$ is the inertia matrix of the same body L_j about the origin of coordinates ($\mathbf{x}_j$, $\mathbf{y}_j$, $\mathbf{z}_j$) in ($\mathbf{x}_o$, $\mathbf{y}_o$, $\mathbf{z}_o$). Thus (13) can be written as

$$K_j = \frac{1}{2}[m_j(\mathbf{v}_j' \mathbf{v}_j + 2\omega_j' C_j \mathbf{v}_j) + \omega_j' \mathbf{J}_j \omega_j] \tag{15}$$

for simplicity. When the numerical values of $\mathbf{J}_j$ are needed for computation, however, it is more convenient to compute them from (14) since $\mathbf{J}_{Gj}$ can be obtained directly.

III. KINETIC ENERGY OF THE ROBOT

Consider an industrial robot having (n+1) links among which link 0 is bolted on the platform. For j = 1,2,...,n, let link j be represented by the rigid body L_j described in Section II. Then equation (15) represents the kinetic energy of link j. The total kinetic energy of the robot can be expressed as:

$$K = \frac{1}{2} \sum_{j=1}^{n} [m_j(\mathbf{v}_j' \mathbf{v}_j + 2\omega_j' C_j \mathbf{v}_j) + \omega_j' \mathbf{J}_j \omega_j] \tag{16}$$

Let $\mathbf{A}_{j-1}^{j}$ be a 3 by 3 rotation matrix which projects any vector with reference to ($\mathbf{x}_j$, $\mathbf{y}_j$, $\mathbf{z}_j$) coordinate system onto ($\mathbf{x}_{j-1}$, $\mathbf{y}_{j-1}$, $\mathbf{z}_{j-1}$) system. Since $\mathbf{A}_{j-1}^{j}$ is an orthonormal matrix, then

$$(\mathbf{A}_{j-1}^{j})^{-1} = (\mathbf{A}_{j-1}^{j})' = (\mathbf{A}_{j}^{j-1})$$

In addition,

$$A_o^j = A_o^1 A_1^2 \dots A_{j-1}^j$$

Thus equation (16) may be written as:

$$K = \frac{1}{2} \sum_{j=1}^{n} [m_j\{(A_j^o v_j)'(A_j^o v_j) + 2(A_j \omega_j)'(A_j^o C_j A_o^j)(A_j^o v_j)\}$$

$$+ (A_j^o \omega_j)'(A_j^o J_j A_o^j)(A_j^o \omega_j)] \tag{17}$$

To simplify the notation, let

$$v_j^* = A_j^o v_j\,, \qquad \omega_j^* = A_j^o \omega_j \tag{18}$$

be the linear and angular velocities of link j, respectively, projected onto their own coordinate system (x_j, y_j, z_j). Likewise, let

$$C_j^* = A_j^o C_j A_o^j\,, \qquad J_j^* = A_j^o J_j A_o^j\,, \tag{19}$$

then C_j^* and J_j^* are, respectively, the radius and inertia matrices of link j about their center of mass referred to their own coordinates (x_j, y_j, z_j). As a result, equation (17) can be written as:

$$K = \frac{1}{2} \sum_{j=1}^{n} [m_j\{(v_j^*)' v_j^* + 2(\omega_j^*)' C_j^* v_j^*\} + (\omega_j^*)' J_j^* \omega_j^*] \tag{20}$$

IV. KINETIC ENERGY IN TERMS OF DUAL-NUMBER TRANSFORMATION

The kinetic energy of the robot given in (20) can also be expressed in terms of dual-number transformation. It is intended to show that the dual-number representation of the robot dynamics leads to a simplified computation of the coefficients D_{ij} and D_{ijk} of the Lagrangian dynamical equation (2).

Dual-Number Transformation

The dual-number algebra has been extended to the vector and matrix calculus [14,15]. The basic operations on the dual-vectors and dual-matrices are similar to those in the complex variable algebra. In this paper, physical quantities such as displacements, velocities, etc. are represented by dual-vectors while the coördinates transformations are represented by dual-matrices. The dual-vectors are line vectors [17], which is different from the usual vector of a point. The dual-matrices are line transformations which transform lines in the space to other lines. The line transformation is efficient in spatial geometry and kinematics [17] since lines such as axes of rotation and paths of traveling arise naturally.

According to the Principle of Transference [19], the dual-number algebraic operations are the same as those of ordinary real number algebra by replacing real numbers by dual numbers. The dual-number transformation (or matrix) is defined based on the following observation. Consider the standard defining relationship between coordinate frmaes of two adjacent links of an industrial robot, as shown in Figure 2, given in [2, p. 53]:

"rotate about $\mathbf{z}_{n-1}$ (axis), an angle θ_n;

translate along $\mathbf{z}_{n-1}$, a distance Δ_n;

translate along rotated $\mathbf{x}_{n-1} = \mathbf{x}_n$ (axis), a length a_n;

rotate about $\mathbf{x}_n$, the twist angle α_n."

The homogeneous transformation that describes these four steps is

$$\mathbf{T}_{n-1}^{n} = \begin{bmatrix} c\theta_n & -s\theta_n & 0 & 0 \\ s\theta_n & c\theta_n & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_n \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha_n & -s\alpha_n & 0 \\ 0 & s\alpha_n & c\alpha_n & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

where $c\theta_n = \cos\theta_n$, $s\theta_n = \sin\theta_n$, etc. for abbreviation. Note that the first two matrices represent a rotation about and a translation along the same axis, hence they are commutable. The transformation T_{n-1}^{n} transforms any 4-dimensional vector with reference to the n-th link coordinate frame to the n-1st link coordinate frame. To apply the dual-number algebra to the robot kinematics, one defines a dual-displacement scalar

$$\hat{\theta}_n = \theta_n + \epsilon\Delta_n \, . \tag{22}$$

in which, as mentioned above, θ_n and Δ_n are displacements with reference to the same axis; and ϵ is the dual unit having the property $\epsilon^2 = 0$ [14-19]. In a way, $\hat{\theta}_n$ describes the displacements of a "screw" after it is turned, which gives rise to the name of "calculus of screws". Likewise one defines

$$\hat{\alpha}_n = \alpha_n + \epsilon a_n \tag{23}$$

The dual-number transformation that describes the four steps is given by [16,19]:

$$\hat{A}_{n-1}^{n} = \begin{bmatrix} c\hat{\theta}_n & -s\hat{\theta}_n & 0 \\ s\hat{\theta}_n & c\hat{\theta}_n & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\hat{\alpha}_n & -s\hat{\alpha}_n \\ 0 & s\hat{\alpha}_n & c\hat{\alpha}_n \end{bmatrix} = \begin{bmatrix} c\hat{\theta}_n & -s\hat{\theta}_n c\hat{\alpha}_n & s\hat{\theta}_n s\hat{\alpha}_n \\ s\hat{\theta}_n & c\hat{\theta}_n c\hat{\alpha}_n & -c\hat{\theta}_n s\hat{\alpha}_n \\ 0 & s\hat{\alpha}_n & c\hat{\alpha}_n \end{bmatrix} \tag{24}$$

It is seen that $\hat{A}_{n-1}^{n}$ is orthonormal so that

$$(\hat{A}_{n-1}^{n})^{-1} = (\hat{A}_{n-1}^{n})' = \hat{A}_{n}^{n-1} \tag{25}$$

For computational purposes, one may use the trigonometric identities sin $(x+y)$=sin x cos y + cos x sin y and cos $(x+y)$ = cos x cos y - sin x sin y, the dual-number property $\epsilon^2=0$, and the Taylor's expansion [14,19] to obtain

$$s\hat{\theta}_n = s\theta_n + \epsilon\Delta_n c\theta_n \, , \quad c\hat{\theta}_n = c\theta_n - \epsilon\Delta_n s\theta_n \, ,$$

and (26)

$$s\hat{\alpha}_n = s\alpha_n + \epsilon a_n c\alpha_n \, , \quad c\hat{\alpha}_n = c\alpha_n - \epsilon a_n s\alpha_n \, ,$$

By substituting (22), (23) and (26) into (24), and then performing some algebraic manipulations, one obtains

$$\hat{\mathbf{A}}_{n-1}^{n} = \mathbf{R}_{n-1}^{n} + \epsilon\mathbf{S}_{n-1}^{n} \tag{27}$$

where

$$\mathbf{R}_{n-1}^{n} = \begin{bmatrix} c\theta & -s\theta c\alpha & s\theta s\alpha \\ s\theta & c\theta c\alpha & -c\theta s\alpha \\ 0 & s\alpha & c\alpha \end{bmatrix}, \ \mathbf{S}_{n-1}^{n} = \begin{bmatrix} -\Delta s\theta & -\Delta c\theta c\alpha + as\theta s\alpha & \Delta c\theta s\alpha + as\theta c\alpha \\ \Delta c\theta & -\Delta s\theta c\alpha - ac\theta s\alpha & \Delta s\theta s\alpha - ac\theta c\alpha \\ 0 & ac\alpha & -as\alpha \end{bmatrix} \tag{28}$$

in which θ, α, Δ and a stand for θ_n, α_n, Δ_n and a_n, respectively, for simplicity. Obvious $\mathbf{R}_{n-1}^{n}$ is a pure rotation operator which is orthonormal so that $(\mathbf{R}_{n-1}^{n})^{-1} = (\mathbf{R}_{n-1}^{n})'$, while $\mathbf{S}_{n-1}^{n}$ is a singular matrix since its determinant equals zero.

Properties of Dual-Number Transformation

Since $\hat{\mathbf{A}}_{n-1}^{n}$ describes the same "four steps" as $\mathbf{T}_{n-1}^{n}$ does, then $\hat{\mathbf{A}}_{i}^{j}$ is a dual-number transformation which transforms any dual-vector with reference to $(\mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j)$ coordinate frame to the same dual-vector with reference to $(\mathbf{x}_i, \mathbf{y}_i, \mathbf{z}_i)$ coordinate frame. By (27) and (28),

$$\hat{\mathbf{A}}_{i}^{j} = \mathbf{R}_{i}^{j} + \epsilon\mathbf{S}_{i}^{j} \tag{29}$$

and the transition property holds for $\hat{\mathbf{A}}_{i}^{j}$ such that

$$(\hat{\mathbf{A}}_{j}^{i})^{-1} = \hat{\mathbf{A}}_{i}^{j} = \hat{\mathbf{A}}_{i}^{i+1}\,\hat{\mathbf{A}}_{i+1}^{i+2} \dots \hat{\mathbf{A}}_{j-1}^{j} \tag{30}$$

Combining (25) and (27) yields an identity matrix:

$$\mathbf{I} = [(\mathbf{R}_{i}^{j})' + \epsilon(\mathbf{S}_{i}^{j})']\,[\mathbf{R}_{i}^{j} + \epsilon\mathbf{S}_{i}^{j}]$$

$$= (\mathbf{R}_{i}^{j})'(\mathbf{R}_{i}^{j}) + \epsilon[(\mathbf{S}_{i}^{j})'(\mathbf{R}_{i}^{j}) + (\mathbf{R}_{i}^{j})'(\mathbf{S}_{i}^{j})]$$

which leads to

$$(\mathbf{R}_{i}^{j})'(\mathbf{R}_{i}^{j}) = \mathbf{I}\,, \tag{32}$$

and

$$(\mathbf{S}_i^j)'(\mathbf{R}_i^j) + (\mathbf{R}_i^j)'(\mathbf{S}_i^j) = \mathbf{O} \tag{33}$$

Equation (32) implies that $\mathbf{R}_i^j$ is orthonormal which is in agreement with (28). Since $(\hat{\mathbf{A}}_j^i)' = \hat{\mathbf{A}}_i^j$ and $(\mathbf{R}_j^i)' = \mathbf{R}_i^j$, then $(\mathbf{S}_j^i)' = \mathbf{S}_i^j$. But $(\mathbf{S}_j^i)' \neq (\mathbf{S}_j^i)^{-1}$ since $\mathbf{S}_j^i$ is singular and $\mathbf{S}_j^i \neq \mathbf{S}_j^k\mathbf{S}_k^i$ in general. However, by equation (33), one obtains:

$$\mathbf{S}_i^j = (\mathbf{S}_j^i)' = -(\mathbf{R}_j^i)'\mathbf{S}_j^i(\mathbf{R}_j^i)' \tag{34}$$

Since $\mathbf{R}_i^i = \mathbf{I} = \hat{\mathbf{A}}_i^i = \mathbf{R}_i^i + \epsilon\mathbf{S}_i^i$, then

$$\mathbf{S}_i^i = \mathbf{O} \tag{35}$$

which is expected since $\mathbf{S}_{n-1}^n$ is singular. Finally, $(\mathbf{R}_i^j + \epsilon\mathbf{S}_i^j) = \hat{\mathbf{A}}_i^j = \hat{\mathbf{A}}_i^k\hat{\mathbf{A}}_k^j$ $=(\mathbf{R}_i^k+\epsilon\mathbf{S}_i^k)(\mathbf{R}_k^j+\epsilon\mathbf{S}_k^j) = \mathbf{R}_i^k\mathbf{R}_k^j + \epsilon(\mathbf{R}_i^k\mathbf{S}_k^j+\mathbf{S}_i^k\mathbf{R}_k^j)$ so that:

$$\mathbf{R}_i^j = \mathbf{R}_i^k\mathbf{R}_k^j \quad \text{and} \quad \mathbf{S}_i^j = \mathbf{R}_i^k\mathbf{S}_k^j + \mathbf{S}_i^k\mathbf{R}_k^j \tag{36}$$

Equations (29) through (36) summarize the essential properties of the dual-transformation.

Kinetic Energy of the Robot

The dual-displacement $\hat{\mathbf{q}}_j = \boldsymbol{\theta}_j+\epsilon\boldsymbol{\Delta}_j$ is a dual-vector where $\boldsymbol{\theta}_j$ is the angular displacement vector about a specific axis while $\boldsymbol{\Delta}_j$ is the linear displacement vector along the same axis. In industrial robots, link j either rotates about z_{j-1}-axis, or travels along that axis. The magnitudes of the displacements are measured from their own reference points. Thus the dual-displacement of link j with reference to $(\mathbf{x}_{j-1}, \mathbf{y}_{j-1}, \mathbf{z}_{j-1})$ is:

$$\hat{\mathbf{q}}_j = (\theta_j+\epsilon\Delta_j)\begin{bmatrix}0\\0\\1\end{bmatrix} \tag{37}$$

Especially,

$$\mathring{\hat{\mathbf{q}}}_j = \begin{cases} [0 \;\; 0 \;\; \theta_j]' & \text{for revolute joint ,} \\ [0 \;\; 0 \;\; \epsilon\Delta_j]' & \text{for prismatic joint ,} \end{cases} \tag{38}$$

where θ_j and Δ_j are the third components of vectors $\boldsymbol{\theta}_j$ and $\boldsymbol{\Delta}_j$, respectively. Let

$$\hat{\mathbf{v}}_j = \omega_j^* + \epsilon \mathbf{v}_j^* \,, \;\; j = 1,2,\ldots,n, \tag{39}$$

be the dual-velocity of link j with reference to coordinates $(\mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j)$. It relates to $\mathring{\hat{\mathbf{q}}}_j$ by [16]:

$$\hat{\mathbf{v}}_j = \hat{\mathbf{A}}_j^{j-1} \mathring{\hat{\mathbf{q}}}_j + \hat{\mathbf{A}}_j^{j-1} \hat{\mathbf{v}}_{j-1} \tag{40}$$

with $\hat{\mathbf{v}}_o = \mathbf{O}$ since link O is stationary. The first term in (40) comes from the dual-velocity $\mathring{\hat{\mathbf{q}}}_j$ with reference to z_{j-1}-axis, while the second term is the contribution of the dual-velocity of link (j-1) with reference to coordinates $(\mathbf{x}_{j-1}, \mathbf{y}_{j-1}, \mathbf{z}_{j-1})$. Apply the recursive relation (40) repeatedly, one obtains

$$\hat{\mathbf{v}}_j = \sum_{i=1}^{j} \hat{\mathbf{A}}_j^{i-1} \mathring{\hat{\mathbf{q}}}_i = \sum_{i=1}^{j} (\hat{\mathbf{A}}_{i-1}^{j})' \mathring{\hat{\mathbf{q}}}_i \tag{41}$$

Substituting (27) and (37) into (41) yields

$$\hat{\mathbf{v}}_j = \sum_{i=1}^{j} (\mathbf{R}_{i-1}^{j})' \, \dot{\theta}_i \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \epsilon \sum_{i=1}^{j} [(\mathbf{S}_{i-1}^{j})' \dot{\theta}_i + (\mathbf{R}_{i-1}^{j})' \dot{\Delta}_i] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{42}$$

Compare (42) with (39) to obtain

$$\omega_j^* = \sum_{i=1}^{j} (\mathbf{R}_{i-1}^{j})' \, \dot{\theta}_i \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \mathbf{v}^* = \sum_{i=1}^{j} [(\mathbf{S}_{i-1}^{j})' \dot{\theta}_i + (\mathbf{R}_{i-1}^{j})' \dot{\Delta}_i] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{43}$$

By combining (20) and (43), the total kinetic energy of the robot becomes

$$K = \frac{1}{2} \sum_{j=1}^{n} \{ m_j \, [0 \; 0 \; 1] \, (\sum_{i=1}^{j} [\mathbf{S}_{i-1}^{j} \dot{\theta}_i + \mathbf{R}_{i-1}^{j} \dot{\Delta}_i] \sum_{k=1}^{j} [(\mathbf{S}_{k-1}^{j})' \dot{\theta}_k + (\mathbf{R}_{k-1}^{j})' \dot{\Delta}_k]) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$+ 2m_j[0\ 0\ 1]\left(\sum_{i=1}^{j} \mathbf{R}_{i-1}^{j}\dot{\theta}_i\mathbf{C}_j^{*} \sum_{k=1}^{j} [(\mathbf{S}_{k-1}^{j})'\dot{\theta}_k + (\mathbf{R}_{k-1}^{j})'\dot{\Delta}_k]\right) \begin{bmatrix}0\\0\\1\end{bmatrix}$$

$$+ [0\ 0\ 1]\left(\sum_{i=1}^{j} \mathbf{R}_{i-1}^{j}\dot{\theta}_i\mathbf{J}_i^{*} \sum_{k=1}^{j} (\mathbf{R}_{k-1}^{j})'\dot{\theta}_k\right) \begin{bmatrix}0\\0\\1\end{bmatrix} \Big\} \tag{44}$$

For i, j=1,2,3, let $\mathbb{D} = [D_{ij}]$ be an arbitrary **3** by **3** matrix; and let

$$\mathrm{Tr}_3(\mathbb{D}) = [0\ 0\ 1]\mathbb{D} \begin{bmatrix}0\\0\\1\end{bmatrix} = D_{33} = \mathrm{Tr}_3(\mathbb{D}') \tag{45}$$

Consequently, (44) can be written as

$$K = \frac{1}{2}\sum_{j=1}^{n}\sum_{i=1}^{j}\sum_{k=1}^{j} [E_{jik}\dot{\theta}_i\dot{\theta}_k + F_{jik}\dot{\theta}_i\dot{\Delta}_k + G_{jik}\dot{\Delta}_i\dot{\Delta}_k] \tag{46}$$

where

$$E_{jik} = \mathrm{Tr}_3[\mathbf{R}_{i-1}^{j}\mathbf{J}_j^{*}(\mathbf{R}_{k-1}^{j})'] + 2m_j\mathrm{Tr}_3[\mathbf{R}_{i-1}^{j}\mathbf{C}_j^{*}(\mathbf{S}_{k-1}^{j})'] + m_j\mathrm{Tr}_3[\mathbf{S}_{i-1}^{j}(\mathbf{S}_{k-1}^{j})']$$

$$F_{jik} = 2m_j\{\mathrm{Tr}_3[\mathbf{R}_{i-1}^{j}\mathbf{C}_j^{*}(\mathbf{R}_{k-1}^{j})'] + \mathrm{Tr}_3[\mathbf{S}_{i-1}^{j}(\mathbf{R}_{k-1}^{j})']\}$$

$$G_{jik} = m_j\mathrm{Tr}_3[\mathbf{R}_{i-1}^{k-1}] \tag{47}$$

Equations (46) and (47) indicate the main difference between the expressions of kinetic energy in terms of dual-number transformation and homogeneous transformation. The homogeneous transformation $\mathbf{T}_o^i$ transforms point vectors. If a point is described with reference to ($\mathbf{x}_i$, $\mathbf{y}_i$, $\mathbf{z}_i$), then the position of the point in base coordinates is the point vector pre-multiplied by $\mathbf{T}_o^i$. The velocity of the point in base coordinates will contain terms of $\sum_{j=1}^{i} (\partial\mathbf{T}_o^i/\partial q_j)(dq_i/dt)$. Consequently the kinetic energy of the robot is [2, p. 167]:

$$K = \frac{1}{2} \sum_{i=1}^{n} \mathrm{Tr}[\sum_{j=1}^{i} \sum_{k=1}^{i} (\partial T_o^i/\partial q_j) J_i (\partial T_o^i/\partial q_j)' q_i q_k] \tag{48}$$

which includes the partial derivatives of T_o^i. When K is substituted into equation (1) to form the Lagrangian equation, its time derivative yields terms of second order partial derivatives $\partial^2 T_o^i/(\partial q_k \partial q_j)$, which causes difficulties in simplifying the computation. The dual-number transformation, however, performs line transforms [16, 17] which transforms velocity vector described with reference to one coordinate system directly to another coordinate system. Thus no derivatives of the dual-number transformation appear in the expression for the kinetic energy, as indicated in equations (46) and (47). As shown in the next section, when the kinetic energy K is placed in the Lagrangian equations, the time derivative of K yields only the first order partial derivatives of the dual-number transformation so that Paul's simplification procedure [2] applies.

It should be emphasized that because of the line transformation property of the dual-number transformation, using it in the terms of potential energy does not have any advantages.

V. LAGRANGIAN EQUATION IN TERMS OF DUAL-NUMBER TRANSFORMATION

The potential energy of the robot is given by

$$P = \sum_{j=1}^{n} m_j \mathbf{g}' \mathbf{r}_{jo} \tag{49}$$

where m_j is the total mass of link j, $\mathbf{r}_{jo}$ is the position vector describing the center of mass of link j with reference to base coordinates, and $\mathbf{g}'$ is the gravitational acceleration vector at a sea level base and equal to [0, 0, 9.8 m/sec^2]. The Lagrangian equation is given by (1) in which L = K - P, q_i is the generalized coordinate, and τ_i is the generalized forcing function. For the robots,

$$q_i = \begin{cases} \theta_i\ , & \text{if link i is revolute ,} \\ \Delta_i\ , & \text{if link i is prismatic ,} \end{cases} \tag{50}$$

is the displacement of link i, and τ_i represents either the torque or the force depending on whether link i is revolute or prismatic.

Revolute Link

If link ℓ is revolute, then $q_\ell = \theta_\ell$. Now

$$\frac{\partial L}{\partial \theta_\ell} = \frac{1}{2} \sum_{j=\ell}^{n} \sum_{i=1}^{j} \sum_{k=1}^{j} \left(\frac{\partial E_{jik}}{\partial \theta_\ell} \dot{\theta}_i \dot{\theta}_k + \frac{\partial F_{jik}}{\partial \theta_\ell} \dot{\theta}_i \dot{\Delta}_k + \frac{\partial G_{jik}}{\partial \theta_\ell} \dot{\Delta}_i \dot{\Delta}_k \right)$$

$$- \sum_{j=\ell}^{n} m_j g' \frac{\partial}{\partial \theta_\ell} r_{jo} \tag{51}$$

and

$$\frac{\partial L}{\partial \dot{\theta}_\ell} = \frac{1}{2} \sum_{j=\ell}^{n} \sum_{i=1}^{j} [(E_{ji\ell} + E_{j\ell i})\dot{\theta}_i + F_{j\ell i}\dot{\Delta}_i] \tag{52}$$

Note that in (51) and (52), the first summations start from $j=\ell$ instead of 1 because E_{jik}, F_{jik}, G_{jik} are independent of θ_ℓ while $\dot{\theta}_j$ is independent of $\dot{\theta}_\ell$, for $j < \ell$. Let

$$\Phi_{j\ell i} = E_{ji\ell} + E_{j\ell i} \tag{53}$$

Then, by (11), (45) and (47),

$$\Phi_{j\ell i} = 2\{Tr_3[R^j_{i-1} J_j^* (R^j_{\ell-1})' + m_j Tr_3[R^j_{i-1} C_j^* (S^j_{\ell-1})' - S^j_{i-1} C_j^* (R^j_{\ell-1})']$$

$$+ m_j Tr_3[S^j_{i-1} (S^j_{\ell-1})']\} \tag{54}$$

Substitute (53) into (52) and then take the time derivative of $\dfrac{\partial L}{\partial \dot{\theta}_\ell}$ to yield

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_\ell}\right) = \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}[\Phi_{j\ell i}\ddot{\theta}_i + F_{j\ell i}\ddot{\Delta}_i]$$

$$+ \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}\sum_{k=1}^{j}[\frac{\partial \Phi_{j\ell i}}{\partial \theta_k}\dot{\theta}_i\dot{\theta}_k + (\frac{\partial \Phi_{j\ell i}}{\partial \Delta_k} + \frac{\partial F_{j\ell k}}{\partial \theta_i})\dot{\theta}_i\dot{\Delta}_k + \frac{\partial F_{j\ell i}}{\partial \Delta_k}\dot{\Delta}_i\dot{\Delta}_k] \quad (55)$$

Finally, by (1), (51) and (55), the external torque of revolute link ℓ becomes

$$\tau_\ell = \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}[\Phi_{j\ell i}\ddot{\theta}_i + F_{j\ell i}\ddot{\Delta}_i]$$

$$+ \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}\sum_{k=1}^{j}[(\frac{\partial \Phi_{j\ell i}}{\partial \theta_k} - \frac{\partial E_{jik}}{\partial \theta_\ell})\dot{\theta}_i\dot{\theta}_k + (\frac{\partial \Phi_{j\ell i}}{\partial \Delta_k} + \frac{\partial F_{j\ell k}}{\partial \theta_i} - \frac{\partial F_{jik}}{\partial \theta_\ell})\dot{\theta}_i\dot{\Delta}_k$$

$$+ (\frac{\partial F_{j\ell i}}{\partial \Delta_k} - \frac{\partial G_{jik}}{\partial \theta_\ell})\dot{\Delta}_i\dot{\Delta}_k] + \sum_{j=\ell}^{n} m_j g' \frac{\partial}{\partial \theta_\ell}\mathbf{r}_{jo} \quad (56)$$

If all the n links are revolute, such as Unimate PUMA 560 robots, then Δ_i is zero for all i so that (56) reduces to

$$\tau_\ell = \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}\Phi_{j\ell i}\ddot{\theta}_i + \frac{1}{2}\sum_{j=\ell}^{n}\sum_{i=1}^{j}\sum_{k=1}^{j}(\frac{\partial \Phi_{j\ell i}}{\partial \theta_k} - \frac{\partial E_{jik}}{\partial \theta_\ell})\dot{\theta}_i\dot{\theta}_k + \sum_{j=\ell}^{n} m_j g' \frac{\partial}{\partial \theta_\ell}\mathbf{r}_{jo} \quad (57)$$

Since in (57), the computation is independent of the order of summation so that (57) can be rewritten as

$$\tau_\ell = \sum_{i=1}^{n} D_{\ell i}\ddot{\theta}_i + \sum_{i=1}^{n}\sum_{k=1}^{n} D_{\ell ik}\dot{\theta}_i\dot{\theta}_k + D_\ell \quad (58)$$

where

$$D_{\ell i} = \frac{1}{2}\sum_{j=\max(i,\ell)}^{n}\Phi_{j\ell i} \quad (59)$$

$$D_{\ell ik} = \frac{1}{2} \sum_{j=\max(i,\ell)}^{n} \left(\frac{\partial \Phi_{j\ell i}}{\partial \theta_k} - \frac{\partial E_{jik}}{\partial \theta_\ell}\right) \tag{60}$$

$$D_\ell = \sum_{j=\ell}^{n} m_j g' \frac{\partial}{\partial \theta_\ell} r_{jo} \tag{61}$$

for robots having revolute links only.

Prismatic Link

If link ℓ is prismatic, then $q_\ell = \Delta_\ell$; and the Lagrangian formulation of the external force for that link is

$$\tau_i = \frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\Delta}_\ell}\right) - \frac{\partial L}{\partial \Delta_\ell}\,. \tag{62}$$

The explicit expression for (62) can be derived in a similar manner as that for the revolute link. Thus the explicit expression is the same as (56) except that the roles of θ_k and Δ_k are interchanged. Thus,

$$\begin{aligned} \tau_\ell &= \frac{1}{2} \sum_{j=\ell}^{n} \sum_{i=1}^{j} [\Psi_{j\ell i}\ddot{\Delta}_i + F_{ji\ell}\ddot{\theta}_i] \\ &+ \frac{1}{2} \sum_{j=\ell}^{n} \sum_{i=1}^{j} \sum_{k=1}^{j} [(\frac{\partial \Psi_{j\ell i}}{\partial \Delta_k} - \frac{\partial G_{jik}}{\partial \Delta_\ell})\dot{\Delta}_i\dot{\Delta}_k \\ &+ (\frac{\partial \Psi_{j\ell k}}{\partial \theta_i} + \frac{\partial F_{ji\ell}}{\partial \Delta_k} - \frac{\partial F_{jik}}{\partial \Delta_\ell})\dot{\Delta}_k\dot{\theta}_i \\ &+ (\frac{\partial F_{ji\ell}}{\partial \theta_k} - \frac{\partial E_{jik}}{\partial \Delta_\ell})\dot{\theta}_i\dot{\theta}_k] + \sum_{j=\ell}^{n} m_j g' \frac{\partial}{\partial \Delta_\ell} r_{jo} \end{aligned} \tag{63}$$

gives the external force for the prismatic link ℓ, where

$$\Psi_{j\ell i} = G_{ji\ell} + G_{j\ell i}$$

$$= 2m_j Tr_3(R_{i-1}^{j-1}) \tag{64}$$

VI. DIFFERENTIAL OPERATORS OF DUAL-NUMBER TRANSFORMATION FOR SIMPLIFICATION

In [2, p. 172] differential operators of homogeneous transformations are introduced to simply the computation of coefficients in the Lagrangian equation. Likewise, differential operators of dual-number transformation in dual-vector space may also be introduced for the same purpose. Let dq_k be a differential displacement (rotation or translation) of link k. Let $\hat{\partial}_j^k$ represent the dual-number differential coordinate transformation of (x_k, y_k, z_k) with reference to (x_j, y_j, z_j). Then the total differential change of dual-number transformation $\hat{A}_i^j$ is

$$d\hat{A}_i^j = \sum_{k=i+1}^{j} \hat{A}_i^j \hat{\partial}_j^k dq_k \quad \text{for } i < j\,. \tag{65}$$

On the other hand,

$$d\hat{A}_i^j = \sum_{k=i+1}^{j} (\partial \hat{A}_i^j / \partial q_k) dq_k \tag{66}$$

Combine (65) and (66) to yield

$$\partial \hat{A}_i^j / \partial q_k = \hat{A}_i^j \hat{\partial}_j^k \quad \text{for } i < k \leq j \tag{67}$$

Since q_k is the displacement of link k with reference to $(x_{k-1}, y_{k-1}, z_{k-1})$, i.e., either about or along the z_{k-1}-axis, then $(\partial \hat{A}_i^j / \partial q_k)$ represents the differential change of $\hat{A}_i^j$ with reference to $(x_{k-1}, y_{k-1}, z_{k-1})$. But $\hat{A}_i^j = \hat{A}_i^{k-1} \hat{A}_{k-1}^j$, and moving links k does not affect link i for $i < k$. Thus

$$\partial A_i^j / \partial q_k = \hat{A}_i^{k-1} \hat{\partial}_{k-1}^k \hat{A}_{k-1}^j \quad \text{for } i < k \leq j \tag{68}$$

By (67) and (68) one obtains

$$\hat{\partial}_j^k = (\hat{A}_i^j)^{-1}\hat{A}_i^{k-1}\hat{\partial}_{k-1}^k\hat{A}_{k-1}^j$$

$$= \hat{A}_j^{k-1}\hat{\partial}_{k-1}^k\hat{A}_{k-1}^j \quad \text{for } i < k \leq j \tag{69}$$

Equation (68) may be used to determine the numerical value of $\hat{\partial}_{k-1}^k$ by setting i = k-1 and j = k. That is

$$\hat{\partial}_{k-1}^k = (\partial\hat{A}_{k-1}^k/\partial q_k)(\hat{A}_{k-1}^k)' \tag{70}$$

since $\hat{A}_{k-1}^k$ is orthonormal and $\hat{A}_{k-1}^{k-1} = \mathbb{I}$. Now (24) and (26) give

$$\hat{A}_{k-1}^k = \Theta_k\Omega_k \tag{71}$$

where

$$\Theta_k = \begin{bmatrix} c\hat{\theta}_k & -s\hat{\theta}_k & 0 \\ s\hat{\theta}_k & c\hat{\theta}_k & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} c\theta_k-\epsilon\Delta_k s\theta_k & -s\theta_k-\epsilon\Delta_k c\theta_k & 0 \\ s\theta_k+\epsilon\Delta_k c\theta_k & c\theta_k-\epsilon\Delta_k s\theta_k & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{72}$$

and

$$\Omega_k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\hat{\alpha}_k & -s\hat{\alpha}_k \\ 0 & s\hat{\alpha}_k & c\hat{\alpha}_k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\alpha_k-\epsilon a_k s\alpha_k & -s\alpha_k-\epsilon a_k c\alpha_k \\ 0 & s\alpha_k+\epsilon a_k c\alpha_k & c\alpha_k-\epsilon a_k s\alpha_k \end{bmatrix} \tag{73}$$

Note that both Θ_k and Ω_k are orthonormal. If link k is revolute, then $q_k = \theta_k$ so that (70) becomes

$$\hat{\partial}_{k-1}^k = (\partial\Theta_k/\partial\theta_k)\Omega_k(\Theta_k\Omega_k)'$$

$$= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{74}$$

If link k is prismatic, then $q_k = \Delta_k$ so that

$$\hat{\partial}^k_{k-1} = (\partial\Theta_k/\partial\Delta_k)\Omega_k(\Theta_k\Omega_k)'$$

$$= \epsilon \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{75}$$

For convenience, let

$$\hat{\partial}^k_j = \delta^k_j + \epsilon \mathbf{d}^k_j \tag{76}$$

and

$$\boldsymbol{\delta} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{77}$$

Then

$$\delta^k_{k-1} = \begin{cases} \boldsymbol{\delta} \text{ , if link k is revolute ,} \\ \mathbf{0} \text{ , if link k is prismatic ,} \end{cases} \tag{78}$$

$$\mathbf{d}^k_{k-1} = \begin{cases} \mathbf{0} \text{ , if link k is revolute ,} \\ \boldsymbol{\delta} \text{ , if link k is prismatic ,} \end{cases} \tag{79}$$

Now substituting (29) and (76) into (69) yields

$$\delta^k_j + \epsilon \mathbf{d}^k_j = (\mathbf{R}^{k-1}_j + \epsilon \mathbf{S}^{k-1}_j)(\delta^k_{k-1} + \epsilon \mathbf{d}^k_{k-1})(\mathbf{R}^j_{k-1} + \epsilon \mathbf{S}^j_{k-1}) \tag{80}$$

$$= \mathbf{R}^{k-1}_j \delta^k_{k-1} \mathbf{R}^j_{k-1} + \epsilon(\mathbf{R}^{k-1}_j \mathbf{d}^k_{k-1} \mathbf{R}^j_{k-1} + \mathbf{S}^{k-1}_j \delta^k_{k-1} \mathbf{R}^j_{k-1}$$

$$+ \mathbf{R}^{k-1}_j \delta^k_{k-1} \mathbf{S}^j_{k-1}) \text{ for } k \le j \text{ .} \tag{81}$$

By (78) and (79), equation (81) for k ≤ j becomes

$$\delta^k_j = \begin{cases} \mathbf{R}^{k-1}_j \boldsymbol{\delta} \mathbf{R}^j_{k-1} \text{ ,} & \text{, if link k is revolute ,} \\ \mathbf{0} & \text{, if link k is prismtic ;} \end{cases} \tag{82}$$

and

$$\mathbf{d}_j^k = \begin{cases} \mathbf{S}_j^{k-1}\delta\mathbf{R}_{k-1}^j + \mathbf{R}_j^{k-1}\delta\mathbf{S}_{k-1}^j & \text{, if link k is revolute ,} \\ \mathbf{R}_j^{k-1}\delta\mathbf{R}_{k-1}^j & \text{, if link k is prismtic ;} \end{cases} \tag{83}$$

Finally, substitute (29) and (76) into (67) to obtain

$$(\partial\mathbf{R}_i^j/\partial q_k) + \epsilon(\partial\mathbf{S}_i^j/\partial q_k) = \mathbf{R}_i^j\delta_j^k + \epsilon(\mathbf{R}_i^j\mathbf{d}_j^k + \mathbf{S}_i^j\delta_j^k) \text{ for } i < k \le j . \tag{84}$$

Consequently

$$\partial\mathbf{R}_i^j/\partial q_k = \begin{cases} \mathbf{R}_i^j\delta_j^k & \text{, for } i < k \le j \text{ ,} \\ \mathbf{0} & \text{, otherwise ;} \end{cases} \tag{85}$$

and

$$\partial\mathbf{S}_i^j/\partial q_k = \begin{cases} \mathbf{R}_i^j\mathbf{d}_j^k + \mathbf{S}_i^j\delta_j^k & \text{, for } i < k \le j \text{ ,} \\ \mathbf{0} & \text{, otherwise ;} \end{cases} \tag{86}$$

Since q_k equals either θ_k or Δ_k, depending on whether link k is either revolute or prismatic, then by (36), (82) (83), (85) and (86), one obtains, if link k is revolute:

$$\partial\mathbf{R}_i^j/\partial\theta_k = \begin{cases} \mathbf{R}_i^{k-1}\delta\mathbf{R}_{k-1}^j & \text{, for } i < k \le j \text{ ,} \\ \mathbf{0} & \text{, otherwise ;} \end{cases} \tag{87}$$

$$\partial\mathbf{S}_i^j/\partial\theta_k = \begin{cases} \mathbf{S}_i^{k-1}\delta\mathbf{R}_{k-1}^j + \mathbf{R}_i^{k-1}\delta\mathbf{S}_{k-1}^j & \text{, for } i < k \le j \text{ ,} \\ \mathbf{0} & \text{, otherwise ;} \end{cases} \tag{88}$$

and if link k is prismatic:

$$\partial\mathbf{R}_i^j/\partial\Delta_k = \mathbf{0} \tag{89}$$

$$\partial S_i^j/\partial \Delta_k = \begin{cases} R_i^{k-1}\delta R_{k-1}^j & \text{, for } i < k \leq j \text{ ,} \\ 0 & \text{, otherwise ;} \end{cases} \tag{90}$$

where δ is given in (77).

VII. COEFFICIENTS OF LAGRANGIAN EQUATION IN EXPLICIT FORMS

The Lagrangian equation for robot dynamics are given by equations (56), or (58) or (63). Using either one of the equations to compute the external torque/force, one must evaluate all the coefficients which contain the first order partial derivatives of E_{jik}, F_{jik}, G_{jik}, $\Phi_{j\ell i}$ and $\Psi_{j\ell i}$. By (11), (34), (45), (47), (53), (64) and (87) through (90), these partial derivatives for $\max(i,k) \leq \ell \leq j$ have explicit expressions as follows:

$$\begin{aligned} \partial E_{jik}/\partial \theta_\ell = {} & \mathrm{Tr}_3[R_{i-1}^{\ell-1}\delta R_{\ell-1}^j J_j^* R_j^{k-1}] \\ & + \mathrm{Tr}_3[R_{k-1}^{\ell-1}\delta R_{\ell-1}^j J_j^* R_j^{i-1}] \\ & + 2m_j\{\mathrm{Tr}_3[R_{i-1}^{\ell-1}\delta R_{\ell-1}^j C_j^* S_j^{k-1}] \\ & \quad - \mathrm{Tr}_3[R_{k-1}^{\ell-1}\delta S_{\ell-1}^j C_j^* R_j^{i-1}] \\ & \quad - \mathrm{Tr}_3[S_{k-1}^{\ell-1}\delta R_{\ell-1}^j C_j^* R_j^{i-1}]\} \\ & + m_j\{\mathrm{Tr}_3[R_{i-1}^{\ell-1}\delta S_{\ell-1}^j S_j^{k-1}] + \mathrm{Tr}_3[S_{i-1}^{\ell-1}\delta R_{\ell-1}^j S_j^{k-1}] \\ & \quad + \mathrm{Tr}_3[R_{k-1}^{\ell-1}\delta S_{\ell-1}^j S_j^{i-1}] + \mathrm{Tr}_3[S_{k-1}^{\ell-1}\delta R_{\ell-1}^j S_j^{i-1}]\} \end{aligned} \tag{91}$$

$$\begin{aligned} \partial \Phi_{jik}/\partial \theta_\ell = {} & 2\,\mathrm{Tr}_3[R_{i-1}^{\ell-1}\delta R_{\ell-1}^j(J_j^* R_j^{k-1} + m_j C_j^* S_j^{k-1})] \\ & + 2\,\mathrm{Tr}_3[R_{k-1}^{\ell-1}\delta R_{\ell-1}^j(J_j^* R_j^{i-1} + m_j C_j^* S_j^{i-1})] \end{aligned}$$

$$+ 2\,m_j \mathrm{Tr}_3[(\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{S}_{\ell-1}^{j} + \mathbf{S}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j})(\mathbf{S}_j^{i-1} - \mathbf{C}_j^{*}\mathbf{R}_j^{i-1})]$$

$$+ 2\,m_j \mathrm{Tr}_3[(\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{S}_{\ell-1}^{j} + \mathbf{S}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j})(\mathbf{S}_j^{k-1} - \mathbf{C}_j^{*}\mathbf{R}_j^{k-1})] \qquad (92)$$

$$\partial F_{jik}/\partial\theta_\ell = 2\,m_j\{\mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{C}_j^{*}\mathbf{R}_j^{k-1}] - \mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{C}_j^{*}\mathbf{R}_j^{i-1}]$$

$$+ \mathrm{Tr}_3[\mathbf{S}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{k-1}] + \mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{S}_{\ell-1}^{j}\mathbf{R}_j^{k-1}]$$

$$+ \mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{S}_j^{i-1}]\} \qquad (93)$$

$$\partial G_{jik}/\partial\theta_\ell = m_j \mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{k-1}] \qquad (94)$$

$$\partial \Psi_{jik}/\partial\theta_\ell = 2\partial \mathbf{G}_{jik}/\partial\theta_\ell \qquad (95)$$

$$\partial E_{jik}/\partial\Delta_\ell = -2m_j \mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{C}_j^{*}\mathbf{R}_j^{i-1}]$$

$$+ m_j\{\mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{S}_j^{k-1}] + \mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{S}_j^{i-1}]\} \qquad (96)$$

$$\partial \Phi_{jik}/\partial\Delta_\ell = -2m_j\{\mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{C}_j^{*}\mathbf{R}_j^{i-1}] + \mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{C}_j^{*}\mathbf{R}_j^{k-1}]\}$$

$$+ 2m_j\{\mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{S}_j^{k-1}] + \mathrm{Tr}_3[\mathbf{R}_{k-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{j}\mathbf{S}_j^{i-1}]\} \qquad (97)$$

$$\partial F_{jik}/\partial\Delta_\ell = 2m_j \mathrm{Tr}_3[\mathbf{R}_{i-1}^{\ell-1}\delta\mathbf{R}_{\ell-1}^{k-1}] \qquad (98)$$

$$\partial G_{jik}/\partial\Delta_\ell = 0\,, \quad \text{and} \qquad (99)$$

$$\partial \Psi_{jik}/\partial\Delta_\ell = 0 \qquad (100)$$

The gravitational term in the Lagrangian equations (56) and (57) involve the partial derivative $\partial\mathbf{r}_{j0}/\partial\theta_\ell$, and in (63) involve $\partial\mathbf{r}_{j0}/\partial\Delta_\ell$. These derivatives are evaluated directly with $\mathbf{r}_{j0}$.

VII. ILLUSTRATIVE EXAMPLE

The Stanford manipulator shown in Figure 3 is adopted as an example to illustrate the use of formulas derived in the paper. The link parameters of the manipulator are given in Table 1. It has seven links and six joints so that n=6. By (28), one obtains:

$$\mathbf{R}_0^1 = \begin{bmatrix} c\theta_1 & 0 & -s\theta_1 \\ s\theta_1 & 0 & c\theta_1 \\ 0 & -1 & 0 \end{bmatrix}, \mathbf{R}_1^2 = \begin{bmatrix} c\theta_2 & 0 & s\theta_2 \\ s\theta_2 & 0 & -c\theta_2 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{R}_2^3 = \mathbf{I},$$

$$\mathbf{R}_3^4 = \begin{bmatrix} c\theta_4 & 0 & -s\theta_4 \\ s\theta_4 & 0 & c\theta_4 \\ 0 & -1 & 0 \end{bmatrix}, \mathbf{R}_4^5 = \begin{bmatrix} c\theta_5 & 0 & s\theta_5 \\ s\theta_5 & 0 & -c\theta_5 \\ 0 & 1 & 0 \end{bmatrix}, \mathbf{R}_5^6 = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 \\ s\theta_6 & c\theta_6 & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

$$\mathbf{S}_0^1 = \ell_1 \begin{bmatrix} -s\theta_1 & 0 & -c\theta_1 \\ c\theta_1 & 0 & -s\theta_1 \\ 0 & 0 & 0 \end{bmatrix}; \mathbf{S}_1^2 = \ell_2 \begin{bmatrix} -s\theta_2 & 0 & c\theta_2 \\ c\theta_2 & 0 & s\theta_2 \\ 0 & 0 & 0 \end{bmatrix}; \mathbf{S}_2^3 = \Delta_2 \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}; \mathbf{S}_3^4 = \mathbf{S}_4^5 = \mathbf{S}_5^6 = \mathbf{0}.$$

Using (36) one also obtains

$$\mathbf{R}_4^6 = \begin{bmatrix} c\theta_5 c\theta_6 & -c\theta_5 s\theta_6 & s\theta_5 \\ s\theta_5 c\theta_6 & -s\theta_5 s\theta_6 & -c\theta_5 \\ s\theta_6 & s\theta_6 & 0 \end{bmatrix}, \quad \mathbf{S}_4^6 = \mathbf{0}$$

$$\mathbf{R}_3^6 = \begin{bmatrix} c\theta_4 c\theta_5 c\theta_6 - s\theta_4 s\theta_6 & -c\theta_4 c\theta_5 s\theta_6 - s\theta_4 c\theta_6 & c\theta_4 s\theta_5 \\ s\theta_4 c\theta_5 c\theta_6 + c\theta_4 s\theta_6 & -s\theta_4 c\theta_5 s\theta_6 + c\theta_4 c\theta_6 & s\theta_4 s\theta_5 \\ -s\theta_5 c\theta_6 & s\theta_5 s\theta_6 & c\theta_5 \end{bmatrix}, \quad \mathbf{S}_3^6 = \mathbf{0}$$

$$\mathbf{R}_2^6 = \mathbf{R}_3^6, \ \mathbf{S}_2^6 = \Delta_3 \begin{bmatrix} -s\theta_4 c\theta_5 c\theta_6 - c\theta_4 s\theta_6 & s\theta_4 c\theta_5 s\theta_6 - c\theta_4 c\theta_6 & -s\theta_4 s\theta_5 \\ c\theta_4 c\theta_5 c\theta_6 - s\theta_4 s\theta_6 & -c\theta_4 c\theta_5 s\theta_6 - s\theta_4 c\theta_6 & c\theta_4 s\theta_5 \\ 0 & 0 & 0 \end{bmatrix}$$

$\mathbf{R}_1^6$ and $\mathbf{S}_1^6$ can be obtained in a similar manner but are omitted here because their lengthy expressions. Now applying (36) further to yield

$$R_2^5 = \begin{bmatrix} c\theta_4 c\theta_5 & -s\theta_4 & c\theta_4 s\theta_5 \\ s\theta_4 c\theta_5 & c\theta_4 & s\theta_4 s\theta_5 \\ -s\theta_5 & 0 & c\theta_5 \end{bmatrix}, \quad R_2^4 = \begin{bmatrix} c\theta_4 & 0 & -s\theta_4 \\ s\theta_4 & 0 & c\theta_4 \\ 0 & -1 & 0 \end{bmatrix}, \quad R_3^5 = R_2^5 ,$$

$$S_2^5 = \Delta_3 \begin{bmatrix} -s\theta_4 c\theta_5 & -c\theta_4 & -s\theta_4 s\theta_5 \\ c\theta_4 c\theta_5 & -s\theta_4 & c\theta_4 s\theta_5 \\ 0 & 0 & 0 \end{bmatrix}, \quad S_2^4 = \Delta_3 \begin{bmatrix} -s\theta_4 & 0 & -c\theta_4 \\ c\theta_4 & 0 & -s\theta_4 \\ 0 & 0 & 0 \end{bmatrix}, \quad S_3^5 = 0 .$$

Sample Computation of $D_{\ell i}$

From the given link parameters shown in Table 1, it is seen that for $i \geq 4$, $\Delta_i = 0$ so that $\dot{\Delta}_i = \ddot{\Delta}_i = 0$. Thus for $\ell = i = 4$, the first summation term in (56) is the same as the first summation term in (58) which gives $D_{44} = \frac{1}{2} \sum_{j=4}^{6} \Phi_{j44}$. To compute Φ_{j44}, one must first compute

$$J_j^* = A_j^o(J_{Gj} + m_j C_j C_j')A_o^j = A_j^o J_{Gj} A_o^j + m_j[(A_j^o C_j A_o^j)(A_j^o C_j' A_o^j)]$$

by (14) and (19). If the off-diagonal terms in the inertia matrices are ignored for simplicity, then one obtains

$$A_j^o J_{Gj} A_o^j = m_j \begin{bmatrix} k_{jxx}^2 & 0 & 0 \\ 0 & k_{jyy}^2 & 0 \\ 0 & 0 & k_{jzz}^2 \end{bmatrix}$$

for $j = 1,2,\ldots,6$, where k_{jxx} is the radius of gyration "xx" of link j about the origin of coordinates (x_j, y_j, z_j), etc. Using (11) and the given data of center of mass in Table 1, one computes:

$$A_i^o C_i A_o^i = \pm b_i \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad A_j^o C_j A_o^j = \pm b_j \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

for i = 1,2,4, and j = 3,5,6, with "+" sign for b_1, b_4, b_5 and b_6, and "-" sign for b_2 and b_3. Since $S_3^4 = S_3^5 = S_3^6 = 0$, then (54) yields $\Phi_{444} = 2\mathrm{Tr}_3[R_3^4 A_4^o(J_{G4} +$

$m_4C_4C_4')A_o^4R_4^3] = 2m_4k_{4yy}^2$. Likewise, $\Phi_{544} = 2m_5[(k_{5xx}^2 + b_5^2)s^2\theta_5 + k_{5zz}^2c^2\theta_5]$ and $\Phi_{644} = 2m_6[(k_{6xx}^2c^2\theta_6 + k_{6yy}s^2\theta_6 + b_6^2)s^2\theta_5 + k_{6zz}^2c^2\theta_5]$. Consequently $D_{44} = m_4k_{4yy}^2 + m_5[(k_{5xx}^2 + b_5^2)s^2\theta_5 + k_{5zz}^2c^2\theta_5] + m_6[(k_{6xx}^2c^2\theta_6 + k_{6yy}^2s^2\theta_6 + b_6^2)s^2\theta_5 + k_{6zz}^2c^2\theta_5]$.

Since link 3 is prismatic, D_{33} must be computed from (63). Since $\ell = i = 3$, and $\ddot{\theta}_3 = 0$, then $D_{33} = \frac{1}{2}\sum_{j=3}^{6} \Psi_{j33} = \sum_{j=3}^{6} m_j Tr_3(R_2^2) = \sum_{j=3}^{6} m_j$ by (64).

Sample Computation of $D_{\ell ik}$

For $\ell = i = k = 4$, one obtains from (53), (56) and (91), $D_{444} = \frac{1}{2}\sum_{j=4}^{6} \partial(\Phi_{j44} - E_{j44})/\partial\theta_4 = \frac{1}{2}\sum_{j=4}^{6} \partial E_{j44}/\partial\theta_4$. Now $\partial E_{444}/\partial\theta_4 = 2Tr_3[\delta R_3^4 J_4^* R_4^3]$. Since δ has all zeros in its third row, then $\partial E_{444}/\partial\theta_4 = 0$. For the same reason, $\partial E_{j44}/\partial\theta_4 = 0$ for j = 5,6. Thus $D_{444} = 0$.

For $\ell = 5$ and $i = k = 4$, one uses the same equations to obtain $D_{544} = \frac{1}{2}\sum_{j=5}^{6} (\partial\Phi_{j54}/\partial\theta_4 - \partial E_{j44}/\partial\theta_5)$. Now $\partial\Phi_{554}/\partial\theta_4 = 2\{Tr_3[R_4^3\delta R_3^5 J_5^* R_5^3] + Tr_3[R_3^3\delta R_3^5 J_5^* R_5^4]\} = 0$, and $\partial\Phi_{654} = 2Tr_3[R_4^3\delta R_3^6 J_6^* R_6^3] = 0$, so that $D_{544} = \frac{-1}{2}\sum_{j=5}^{6} \partial E_{j44}/\partial\theta_5 = -\{Tr_3[R_3^4\delta R_4^5 J_5^* R_5^3] + Tr_3[R_3^4\delta R_4^5 J_6^* R_6^3]\} = -[m_5(k_{5xx}^2 + b_5^2 - k_{5zz}^2)s\theta_5c\theta_5 + m_6\{(k_{6xx}^2c^2\theta_6 + k_{6yy}^2s^2\theta_6 + b_6^2 - k_{6zz}^2)s\theta_5c\theta_5\}]$.

Sample Computation of D_ℓ

Since $g' = [0\ \ 0\ \ g]$ is the gravitational acceleration, then $g'r_{jo} = gh_j$ where h_j is the z component of the position vector describing the center of mass of link j in coordinates $(x_o,\ y_o,\ z_o)$. From Figure 3, it is seen that $h_1 = \ell_1 - b_1$, $h_2 = \ell_1$, $h_3 = \ell_1 + (\Delta_3 - b_3)c\theta_2$, $h_4 = \ell_1 + (\Delta_3 - b_4)C\theta_2$, $h_5 = \ell_1 + (\Delta_3 + b_5c\theta_5)c\theta_2 - b_5s\theta_5c\theta_4s\theta_2$, and $h_6 = \ell_1 + (\Delta_3 + b_6c\theta_5)c\theta_2 - b_6s\theta_5c\theta_4s\theta_2$. Thus $\partial h_4/\partial\theta_4 = 0$, $\partial h_5/\partial\theta_4 = b_5s\theta_5s\theta_4s\theta_2$, and $\partial h_6/\partial\theta_4 = b_6s\theta_5s\theta_4s\theta_2$ so that $D_4 = \sum_{j=4}^{6} m_j g'\partial r_{jo}/\partial\theta_4 =$

$\sum_{j=4}^{6} m_j g \partial h_j/\partial\theta_4 = g(m_5b_5+m_6b_6)s\theta_5 s\theta_4 s\theta_2$, $D_5 = g(m_5b_5+m_6b_6)\ (s\theta_5 c\theta_2 + c\theta_5 c\theta_4 s\theta_2)$, and $D_6 = 0.$

Since $\partial h_3/\partial\Delta_3 = \partial h_4/\partial\Delta_3 = \partial h_5/\partial\Delta_3 = \partial h_6/\partial\Delta_3 = c\theta_2$, then for the prismtic joint 3, $D_3 = (m_3 + m_4 + m_5 + m_6)gc\theta_2$.

VIII. COMPUTATIONAL COMPLEXITY AND COMPARISON

The complexity of computation is represented by the numbers of multiplications and additions required to compute all the coefficients $D_{\ell i}$ and $D_{\ell ik}$.

Method of Direct Homogeneous Transformation

The method of direct homogeneous transformation refers to the computation of the coefficients directly using (3), (4) and (5).

(a) $D_{\ell i}$

Since $\mathrm{Tr}[U_{jk}J_jU'_{ji}] = \mathrm{Tr}[T_o^{k-1}Q_kT_{k-1}^{j}J_j\ (T_{i-1}^{j})'(Q_i)'(T_o^{i-1})']$ and

$$T_m^n = \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ 0 & 0 & 0 & 1 \end{bmatrix},\quad Q_m = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},\quad J_j = \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$$

where x represents the numerical value of that element in the matrix, and it is not necessarily zero, then the number of multiplications required in computing one trace operator is $3\cdot 2\cdot 4 + 3\cdot 4\cdot 4 + 3\cdot 4\cdot 2 + 3\cdot 2 = 102$ and the number of additions is $3\cdot 1\cdot 4 + 3\cdot 3\cdot 4 + 3\cdot 3\cdot 2 + 3\cdot 1 + 2 = 71$.

(b) $D_{\ell ik}$

Since $\mathrm{Tr}[U_{jkm}J_jU'_{ji}] = \mathrm{Tr}[T_o^{k-1}Q_kT_{k-1}^{m-1}Q_mT_{m-1}^{j}J_j(T_{i-1}^{j})'Q'_k(T_o^{i-1})']$, the number of multiplications in computing one trace operator is $3\cdot 2\cdot 2 + 3\cdot 2\cdot 4 + 3\cdot 4\cdot 4 + 3\cdot 4\cdot 2 + 3\cdot 2 = 114$, and the number of additions is $3\cdot 1\cdot 2 + 3\cdot 1\cdot 4 + 3\cdot 3\cdot 4 + 3\cdot 3\cdot 2 + 3\cdot 1 + 2 = 77$.

(c) Total Number

To compute all the input torques/forces for the robot having n moving links, one must evaluate N_1 trace operators for $D_{\ell i}$ and N_2 trace operators for $D_{\ell ik}$, where [4]

$$N_1 = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=1}^{j} 1 = n(n+1)(2n+1)/6, \text{ and } N_2 = \sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=1}^{j}\sum_{m=1}^{j} 1 = n^2(n+1)^2/4.$$

Consequently, the total numbers of multiplications and additions are:

$$\text{Number of multiplications} = 102N_1 + 114N_2 = 28\frac{1}{2}n^4 + 91n^3 + 79\frac{1}{2}n^2 + 19n$$

$$\text{Number of additions} = 71N_1 + 77N_2 = 19\frac{1}{2}n^4 + 62\frac{1}{6}n^3 + 54\frac{3}{4}n^2 + 11\frac{5}{6}n$$

Method of Differential Operator Simplification

Paul's differential operator simplification method applies to $D_{\ell i}$ only.

(a) $D_{\ell i}$

The simplification formula for D_{ij} is given by equation (6.79) of [2, p. 174]. Thus, instead of evaluating the trace operator, one must compute

$$m_p[({}^p\delta_i)' K_p({}^p\delta_i) + ({}^p d_i)'({}^p d_j) + ({}^p\bar{r}_p)'({}^p d_i \times {}^p\delta_j + {}^p d_j \times {}^p\delta_i)]$$

Now each cross product involves six multiplications and three additions, thus to compute one such expression, one requires to perform $(6 \cdot 2 + 1 \cdot 3 \cdot 3 + 3 + 3 + 6 + 6 + 3) = 42$ multiplications and $(3 \cdot 2 + 1 \cdot 2 \cdot 3 + 2 + 2 + 3 + 3 + 3 + 2 + 2) = 29$ additions.

(b) $D_{\ell ik}$

The parameters $D_{\ell ik}$ is not simplified, and hence it requires 114 multiplications and 77 additions to compute one trace operator.

(c) Total Number

For n input torques/forces, one must compute N_1 dot-product-cross-product expressions and N_2 trace operators. Thus:

Number of multiplications $= 42N_1 + 114N_2 = 28\frac{1}{2}n^4 + 71n^3 + 49\frac{1}{2}n^2 + 7n$

Number of additions $= 29N_1 + 77N_2 = 19\frac{1}{4}n^4 + 48n\frac{1}{6}n^3 + 33\frac{3}{4}n^2 + 4\frac{5}{6}n$

Method of Dual-Number Transformation

For the purpose of comparison, the expression having largest number of computational operations is used.

(a) $D_{\ell i}$

In (56) and (63), since either $\ddot{\theta}_i = 0$, or $\ddot{\Delta}_i = 0$, and $\Phi_{j\ell i}$ involves more computations than $F_{j\ell i}$ does, then use $\Phi_{j\ell i}$ for evaluation. Now $\Phi_{j\ell i}$ is given by (54) in which matrix C_j^* has zeros along its principal diagonal. Thus to compute $\Phi_{j\ell i}$, it involves $(1 \cdot 3 \cdot 3 + 3) + 3 + (1 \cdot 2 \cdot 3 + 3) \cdot 2 = 33$ multiplications and $(1 \cdot 2 \cdot 3 + 2) + 2 + (1 \cdot 1 \cdot 3 + 2) \cdot 2 + 3 = 23$ additions.

(b) $D_{\ell ik}$

By the similar reason, $\partial\Phi_{jik}/\partial\theta_\ell$ is used for evaluation. That partial derivative is given by (92). In (92), $\boldsymbol{\delta}$ is defined by (77) which simplifies the computation. Thus in computing the Tr_3 operator, each of $\mathbf{R}\boldsymbol{\delta}\mathbf{R}$ and $\mathbf{R}\boldsymbol{\delta}\mathbf{S}$ requires only the multiplication of the third row of $\mathbf{R}\boldsymbol{\delta}$ with $\mathbf{R}$ or $\mathbf{S}$, i.e., $1 \cdot 2 \cdot 3 = 6$ multiplications. Likewise, each of $\mathbf{CS}$ and $\mathbf{CR}$ requires $3 \cdot 2 \cdot 1 = 6$ multiplications. Hence to compute one $\partial\Phi_{jik}/\partial\theta_\ell$ one must compute $(1 \cdot 2 \cdot 3 + 3 \cdot 3 \cdot 1 + 3 \cdot 2 \cdot 1 + 3) \cdot 2 + (1 \cdot 2 \cdot 3 \cdot 2 + 3 \cdot 2 \cdot 1 + 3) \cdot 2 = 90$ multiplications and $(1 \cdot 1 \cdot 3 + 3 \cdot 2 \cdot 1 + 3 \cdot 1 \cdot 1 + 3 + 2) \cdot 2 + 3 + (1 \cdot 1 \cdot 3 \cdot 2 + 3 + 3 \cdot 1 \cdot 1 + 3 + 2) \cdot 2 = 71$ additions.

(c) Total Number

For computing all n input torques/force, one is required to perform $33N_1 + 90N_2 = 22\frac{1}{2}n^4 + 56n^3 + 39n^2 + 5\frac{1}{2}n$ multiplications and

$$23N_1 + 71N_2 = 17\frac{3}{4}n^4 + 43\frac{1}{6}n^3 + 29\frac{1}{4}n^2 + 3\frac{5}{6}n \text{ addition.}$$

Table 2 summarizes the computational complexity of the three methods for comparison. It is seen that the dual-number approach requires less computations.

IX. CONCLUSION

The Lagrangian formulation of dynamical equations for robots is inherently complicated. The homogeneous transformation is a point transformation which operates on the displacements. Using the homogeneous transformation in the Lagrangian equation thus introduces second order partial derivatives. The dual-number transformation, however, is a line transformation which is extended to dual-vectors to represent velocities. When the dual-number transformation is adopted in the Lagrangian equation, only the first order partial derivatives will appear. Thus the differential operator can be applied directly. As a result, the magnitude of computational complexity is reduced.

REFERENCES

1. Bejczy, A. K., *Robot Arm Dynamics and Control,* Technical Memorandum 33-669, Jet Propulsion Laboratory, February 1974.

2. Paul, R. P., *Robot Manipulators: Mathematics, Programming and Control,* MIT Press, 1981.

3. Luh, J. Y. S., "Conventional Controller Design for Industrial Robots - A Tutorial," IEEE Transactions on Systems, Man and Cybernetics, Vol. 13, No. 3, May/June 1983, pp. 298-316.

4. Luh, J. Y. S., M. W. Walker and R. P. C. Paul, "On-Line Computational Scheme for Mechanical Manipulators," ASME Transactions, Journal of Dynamic Systems, Measurement and Control, Vol. 102, No. 2, June 1980, pp. 69-76.

5. Walker, M. W. and D. E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," ibid, Vol. 104, No. 3, September 1982, pp. 205-211.

6. Hollerbach, J. M., "A Recursive Lagrangian Formulation of Manipulator Dynamics and a Cooperative Study of Dynamics Formulation Complexity," IEEE Transactions on Systems, Man and Cybernetics, Vol. 10, No. 11, November 1980, pp. 730-736.

7. Silver, W. M., "On the Equivalence of Lagrangian and Newton-Euler Dynamics for Manipulators," International Journal of Robotics Research, Vol. 1, No. 2, Summer 1982, pp. 60-70.

8. Kane, T. R. and D. A. Levinson, "The Use of Kane's Dynamical Equations in Robotics," International Journal of Robotics Research, Vol. 2, No. 3, Fall 1983, pp. 3-21.

9. Featherstone, R., "The Calculation of Robot Dynamics Using Articulated-Body Inertias," International Journal of Robotics Research, Vol. 2, No. 1, Spring 1983, pp. 13-30.

10. Bejczy, A. K. and R. P. Paul, "Simplified Robot Arm Dynamics for Control", Proceedings of 20th IEEE Conference on Decision and Control, December 16-18, 1981, San Diego, California, pp. 261-262.

11. Bejczy, A. K., "Dynamic Analysis for Robot Arm Control," Proceedings of 1983 American Control Conference, June 22-24, 1983, San Francisco, California, pp. 503-504.

12. Luh, J. Y. S. and C. S. Lin, "Automatic Generation of Dynamic Equations for Mechanical Manipulators," Proceedings of Joint Automatic Control Conference, June 17-19, 1981, Charlottesville, Virginia, pp. TA-2D.

13. Bejczy, A. K. and S. Lee, "Robot Arm Dynamic Model Reduction for Control," Proceedings of 22nd IEEE Conference on Decision and Control, December 14-16, 1983, San Antonio, Texas, pp. 1466-1476.

14. Brand, L., *Vector and Tensor Analysis,* Wiley and Sons, 1948, chapter 2.

15. Dimentberg, F. M., *The Screw Calculus and Its Applications in Mechanics,* Izdatel'stvo "Nauka", Moskva 1965, English Translation by Foreign Technology Division, WP-AFB Ohio, Part No. 680 993, April 1968.

16. Pennock, G. R. and A. T. Yang, "Dynamic Analysis of a Multi-Rigid-Body Open-Chain System," ASME Transactions, Journa of Mechanisms, Transmission, and Automation Design, Vol. 105, No. 1, March 1983, pp. 28-34.

17. Rooney, J., "A Comparison of Representations of General Spatial Screw Displacement," Environment and Planning (England), Series B, Vol. 5, 1978, pp. 45-88.

18. Yang, A. T., "Inertia Force Analysis of Spatial Mechanisms," ASME Transactions, Journal of Engineering for Industry, Vol. 93, No. 1, February 1971, pp. 27-33.

19. Yang, A. T., "Calculus of Screws," in *Basic of Design Theory,* Edited by W. R. Spillers, North-Holland Publishing Co./American Elsevier Publishing Co., 1974, pp. 266-281.

20. Goldstein, H., *Classical Mechanics,* Addison-Wesley, 1959, p. 144.

APPENDIX A - INERTIA ABOUT THE ORIGIN OF COORDINATES

To compute the inertia matrix $\mathbf{J}_j$ of the rigid body L_j about the origin O_j of coordinates (x_j, y_j, z_j) in (x_o, y_o, z_o), let Δm_i be the mass of the i-th particle of the body, and $\mathbf{r}_{ji}$ and S_{ji} the radius vectors from the origin O_j and the center of mass G_j, respectively, to Δm_i as shown in Figure 1. Then it is well known [20] that the angular momentum referred to O_j is

$$\mathbf{h}_j = \sum_i \Delta m_i \, [\mathbf{r}_{ji} \times (\omega_j \times \mathbf{r}_{ji})]$$

$$= \sum_i \Delta m_i \, [(\mathbf{r}_{ji} \cdot \mathbf{r}_{ji})\omega_j - (\mathbf{r}_{ji} \cdot \omega_j)\mathbf{r}_{ji}] \tag{A.1}$$

where "$\cdot$" is the dot product. Since $(\mathbf{r}_{ji} \cdot \mathbf{r}_{ji}) = (\mathbf{r}_{ji})'\mathbf{r}_{ji}$ and $(\mathbf{r}_{ji} \cdot \omega_j)\mathbf{r}_{ji} = [\mathbf{r}_{ji}(\mathbf{r}_{ji})']\omega_j$ where $()' =$ transpose of (), (A.1) can be written as

$$\mathbf{h}_j = \sum_i \Delta m_i \, [(\mathbf{r}_{ji})'\mathbf{r}_{ji}\mathbf{I} - \mathbf{r}_{ji}(\mathbf{r}_{ji})']\omega_j \tag{A.2}$$

in which $\mathbf{I}$ is the identity matrix, and $\mathbf{r}_{ji}(\mathbf{r}_{ji})'$ is the outer product matrix. But $\mathbf{h}_j = \mathbf{J}_j\omega_j$, hence

$$\mathbf{J}_j = \sum_i \Delta m_i \, [(\mathbf{r}_{ji})'\mathbf{r}_{ji}\mathbf{I} - \mathbf{r}_{ji}(\mathbf{r}_{ji})'] \tag{A.3}$$

which is symmetric since $\mathbf{r}_{ji}(\mathbf{r}_{ji})'$ is symmetric. From Figure 1, $\mathbf{r}_{ji} = \mathbf{s}_{ji} + \mathbf{c}_j$, so that equation (A.3) can be written as:

$$\mathbf{J}_j = \sum_i \Delta m_i \, [(\mathbf{s}_{ji} + \mathbf{c}_j)'(\mathbf{s}_{ji} + \mathbf{c}_j)\mathbf{I} - (\mathbf{s}_{ji} + \mathbf{c}_j)(\mathbf{s}_{ji} + \mathbf{c}_j)'] \tag{A.4}$$

But G_j is the center of mass so that $\sum_i \Delta m_i \mathbf{s}_{ji} = 0$. Hence equation (A.4) reduces to

$$\mathbf{J}_j = \sum_i \Delta m_i \, [(\mathbf{s}_{ji})'\mathbf{s}_{ji}\mathbf{I} + \mathbf{c}_j'\mathbf{c}_j\mathbf{I} - \mathbf{s}_{ji}(\mathbf{s}_{ji})' - \mathbf{c}_j\mathbf{c}_j'] \tag{A.5}$$

Since the origin O_j is arbitrarily assigned, equation (A.3) derived above is satisfied at the center of mass G_j, i.e.

$$\mathbf{J}_{Gj} = \sum_i \Delta m_i \left[(\mathbf{s}_{ji})' \mathbf{s}_{ji}\mathbf{I} - \mathbf{s}_{ji}(\mathbf{s}_{ji})'\right] \tag{A.6}$$

Combine (A.5) and (A.6) to yield

$$\mathbf{J}_j = \mathbf{J}_{Gj} + m(\mathbf{c}_j' \mathbf{c}_j\mathbf{I} - \mathbf{c}_j\mathbf{c}_j') \tag{A.7}$$

Refer to the definition of $\mathbf{c}_j$ and $\mathbf{C}_j$, it is seen that

$$\mathbf{c}_j' \mathbf{c}_j\mathbf{I} = (c_{xj}^2 + c_{yj}^2 + c_{zj}^2)\mathbf{I}\text{ , and } \mathbf{c}_j\mathbf{c}_j' = \begin{bmatrix} c_{xj}^2 & c_{xj}c_{yj} & c_{xj}c_{zj} \\ c_{yj}c_{xj} & c_{yj}^2 & c_{yj}c_{zj} \\ c_{zj}c_{xj} & c_{zj}c_{yj} & c_{zj}^2 \end{bmatrix} \tag{A.8}$$

Thus $\mathbf{c}_j' \mathbf{c}_j\mathbf{I} - \mathbf{c}_j\mathbf{c}_j' = \mathbf{C}_j\mathbf{C}_j' = \mathbf{C}_j'\mathbf{C}_j$ and hence (A.7) becomes

$$\mathbf{J}_j = \mathbf{J}_{Gj} + m_j\mathbf{C}_j\mathbf{C}_j' \tag{A.9}$$

Table 1. Link Parameters of Stanford Manipulator

Link i	θ_i	Δ_i	α_i	a_i	$\cos\alpha_i$	$\sin\alpha_i$	Center of Mass in (x_i, y_i, z_i)
1	θ_1	ℓ_1	$-\pi/2$	0	0	-1	$(0, b_1, 0)$
2	θ_2	ℓ_2	$\pi/2$	0	0	1	$(0, -b_2, 0)$
3	0	Δ_3	0	0	1	0	$(0, 0, -b_3)$
4	θ_4	0	$-\pi/2$	0	0	-1	$(0, b_4, 0)$
5	θ_5	0	$\pi/2$	0	0	1	$(0, 0, b_5)$
6	θ_6	0	0	0	1	0	$(0, 0, b_6)$

Table 2. Comparison of Computational Complexity

METHOD	NUMBER OF MULTIPLICATIONS*	NUMBER OF ADDITIONS*
Homogeneous Transformation	$28\frac{1}{2}n^4+91n^3+79n^2+19n$	$19\frac{1}{2}n^4+62\frac{1}{6}n^3+54\frac{3}{4}n^2+11\frac{5}{6}n$
Differential Simplification	$28\frac{1}{2}n^4+71n^3+49\frac{1}{2}n^2+7n$	$19\frac{1}{4}n^4+48\frac{1}{6}n^3+33\frac{3}{4}n^2+4\frac{5}{6}n$
Dual-Number	$22\frac{1}{2}n^4+56n^3+39n^2+5\frac{1}{2}n$	$17\frac{3}{4}n^4+43\frac{1}{6}n^3+29\frac{1}{4}n^2+3\frac{5}{6}n$

*Computing all D_{ij} and D_{ijk} for n joints.

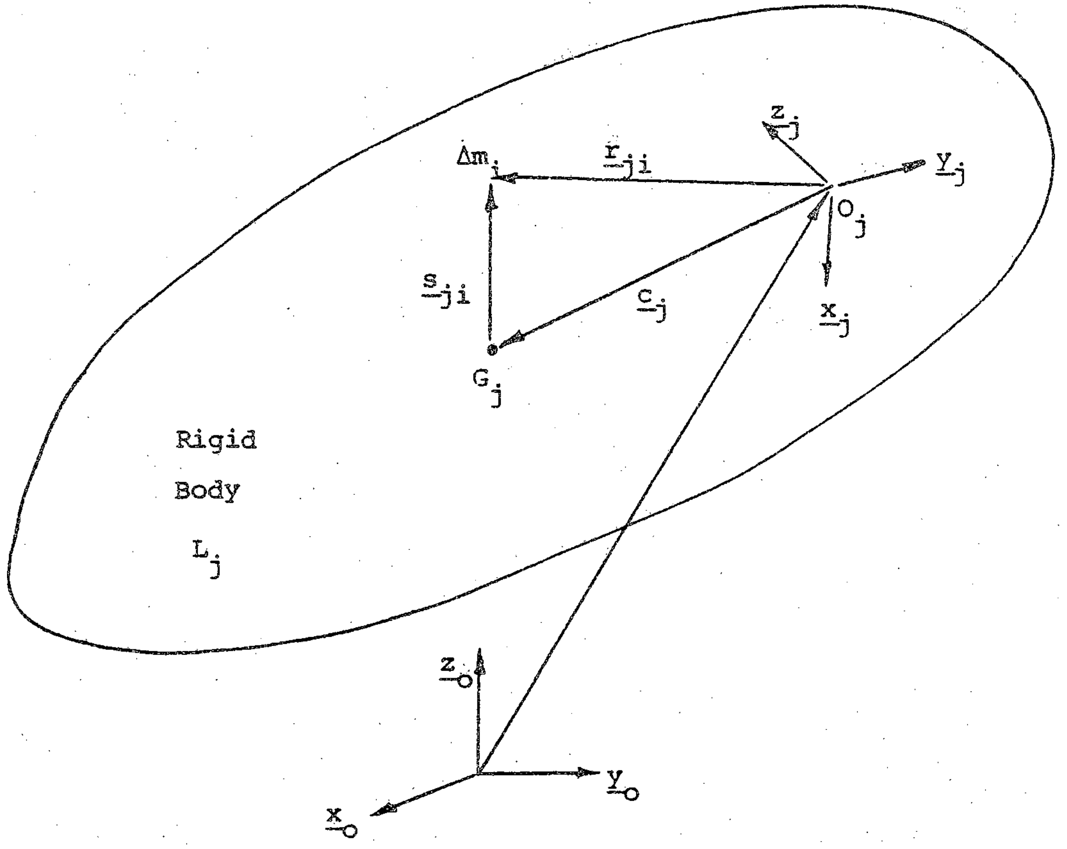


Figure 1. Moving Rigid Body

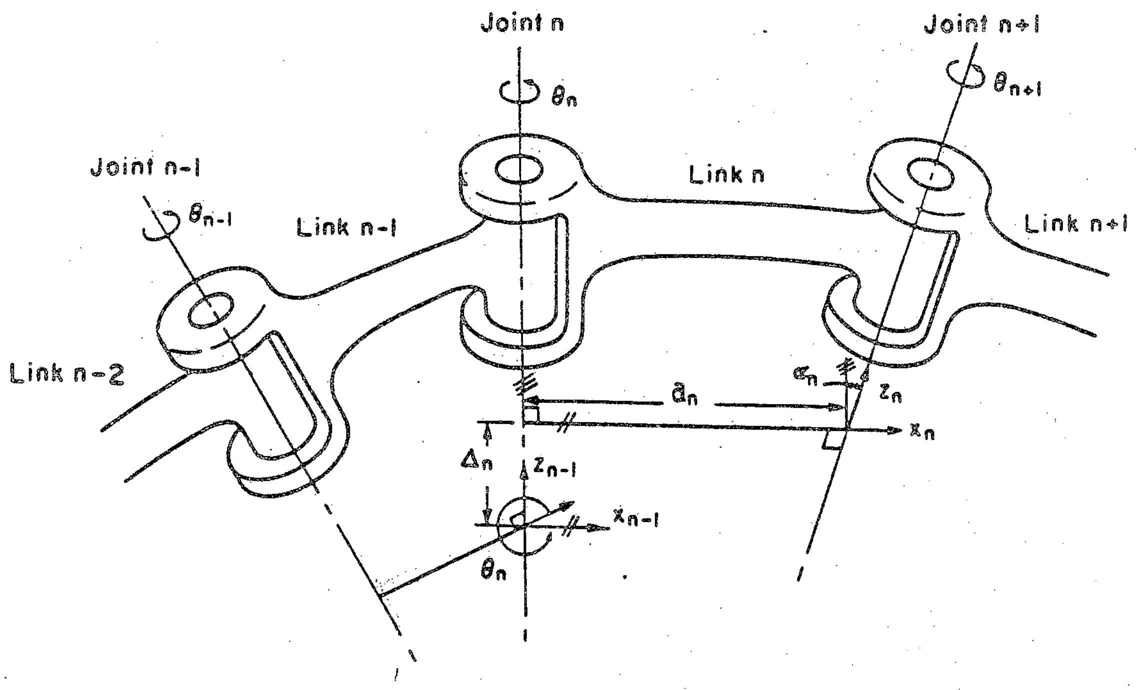


Figure 2. Link Parameters θ, Δ, α and a

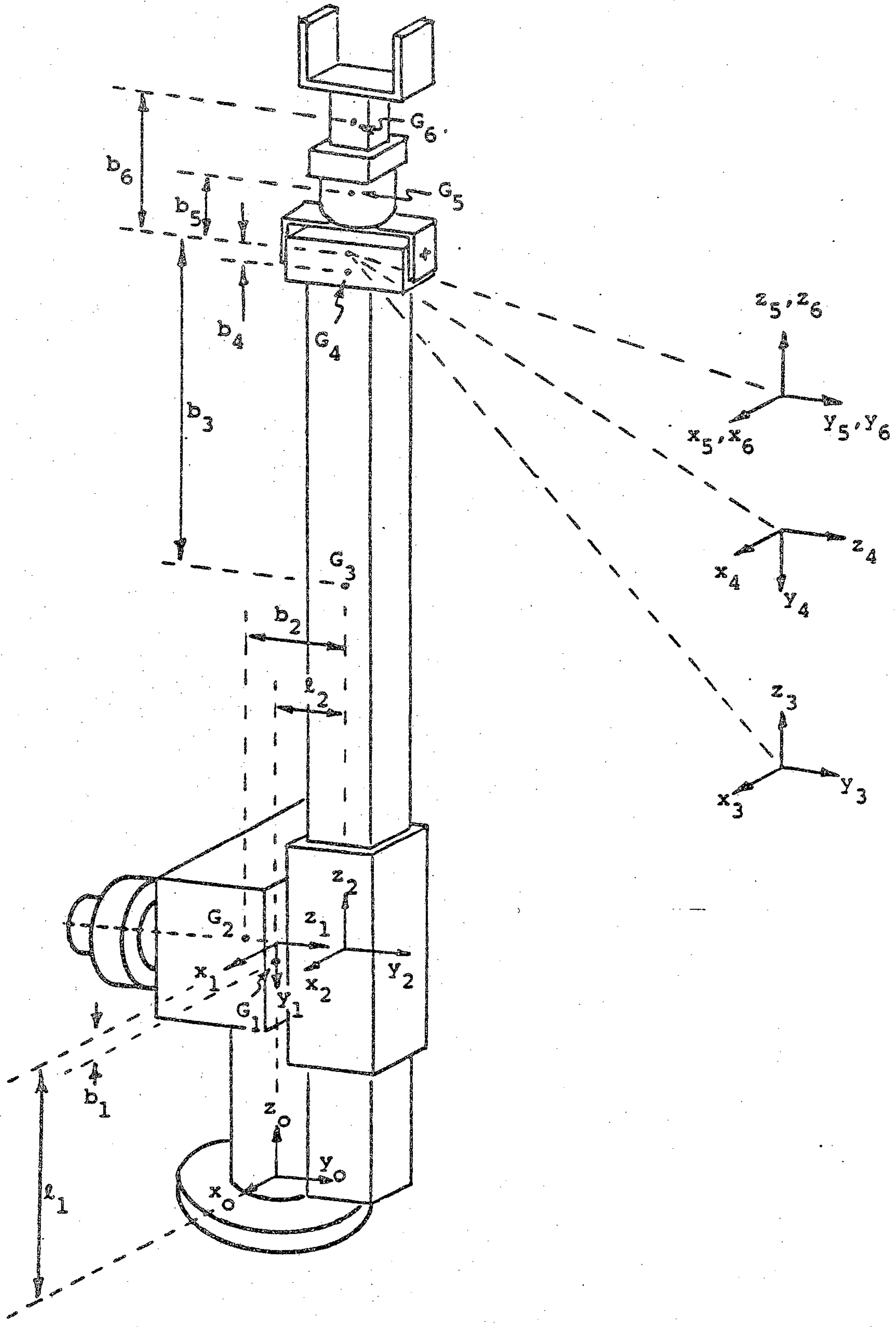


Figure 3. Stanford Manipulator with Joint Coordinates and Link Parameter

RTM (Robot Time and Motion) User Manual
Version 1.2

A.P. Robinson, H. Lechtman and S.Y. Nof

School of Industrial Engineering
Purdue University
West Lafayette, Indiana 47907

Research Memorandum No. 84-12
July 1984

This research was partially supported by the National Science Foundation under grant MEA-8119884. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

RTM (Robot Time and Motion) User Manual

Version 1.2

A.P. Robinson, H. Lechtman, and S.Y. Nof

School of Industrial Engineering
Purdue University

1. INTRODUCTION

The RTM method is a means of estimating robot cycle times based on information describing the robot's task, but without a need for extensive programming. This method has been incorporated into a software system which reads standardized input describing the task and applies the RTM-method based on that input. This manual has been prepared to explain the proper use of the RTM input language and how it operates with the RTM prototype software. Currently, the system can model both the Stanford Arm (SA) and the Cincinnati Milacron T-3. (Additional RTM capabilities that have been researched, and examples of applying RTM to other robot types are provided in the references [1-7], but are not included in the standard RTM software.) A commercial version of RTM has also been developed [8, 9]. Discussions of the motion time modeling mechanism, a detailed example of its application, and experimental evaluation of RTM are presented in the appendix.

The manual is organized in the following manner.

It should be emphasized that the main purpose of RTM is to provide the foundation for a simple-to-use technique for practical robot work method evaluation, and robot models comparison. While computational models of individual work elements may occasionally be inaccurate, the RTM method has proven quite accurate for complete task method analysis. Additional subroutines for more robot models can be developed into the prototype structure of RTM.

2. RTM ELEMENTS

A listing of all RTM elements, their definitions, and their required parameters is presented in Table 1. All elements are applicable to the Stanford Arm. The "Stop on Position Error" and "Stop on Force or Moment" elements cannot be applied to modeling the T3. Also unused by the T3 modeler are the complex grasping elements GR2 and GR3. The T3 does not posses these capabilities and the elements should not be included in a T3 task description. Sensor elements can be added to both robot types if sensors are available, as specified in Section 11.

3. SOFTWARE OVERVIEW

A diagram representing the RTM software structure is shown in Table 2. The software can be thought of as three separate groups of subroutines.

1. Input Routines
2. Run Routines
3. Modeling Routines

Input routines receive the input data and store it in standardized formats and locations to be accessed by the run routines. The run routines represent the RTM analyzer of Table 2. They index through the motions of the task according to the control structures of the input tasks. Modeling routines are called by the run routines when a motion time is to be estimated.

Motion times for the T3 can be estimated in two ways. One way accesses a table of reference times and interpolates linearly to estimate motion time based on motion length and velocity. The second method utilizes equations which predict motion time based again on motion length and velocity, but relying on the velocity control models of T3. The Stanford Arm software possesses the second form of motion time estimation. A summary of all current RTM input statements is provided in Table 3.

4. INPUT DECK:

4.1 Card 1: Robot name and input type

Two characters signifying the robot name are specified in columns 1 and 2 [current available names include: T3 - Cincinnati Milacron T3 and SA - Stanford Arm]. If the robot is the T3 the users also must specify input type. This type can be either 1 or 2 and is declared in the fourth column. Input type is explained in a later section.

Table 1: RTM Symbols and Elements

Element No.		Symbol		Definition of Element	Element Parameters
1		Rn		*n-segment reach:* Move unloaded manipulator along a path comprised of n segments	Displacement (linear or angular) and velocity or Path geometry and velocity
2		Mn		*n-segment move:* Move object along path comprised of n segments	
3		ORn		*n-segment orientation:* Move manipulator mainly to reorient	
4		SEi		*stop on position error*	Error bound
	4.1		SE1	Bring the manipulator to rest immediately without waiting to null out joints errors	
	4.2		SE2	Bring the manipulator to rest within a specified position error tolerance	
5		SFi		*Stop on force or moment*	Force, torque and touch values
	5.1		SF1	Stop the manipulator when force conditions are met	
	5.2		SF2	Stop the manipulator when torque conditions are met	
	5.3		SF3	Stop the manipulator when either torque or force conditions are met	
	5.4		SF4	Stop the manipulator when touch conditions are met	
6		Vi		Vision operation	Time function
7		GRi		*Grasp an object*	Distance to close/open fingers
	7.1		GR1	Simple grasp of object by closing fingers	
	7.2		GR2	Grasp object while centering hand over it	
	7.3		GR3	Grasp object by closing one finger at a time	
8		RE		Release object by opening fingers	
9		T		Process time delay when the robot is part of the process	Time function
10		D		Time delay when the robot is waiting for a process completion.	Time function

Table 2: General Structure of the RTM Analyzer

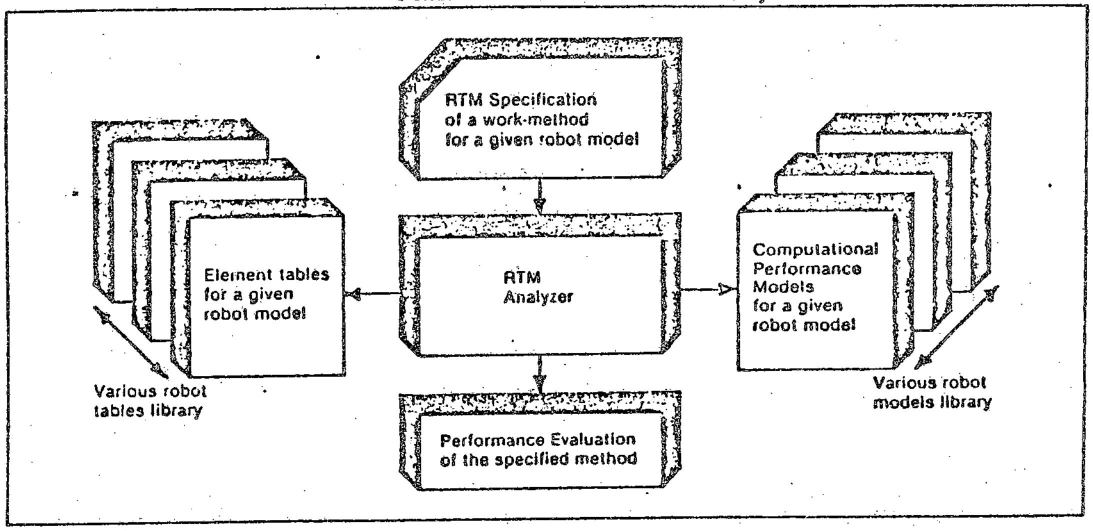

RTM Specification of a work-method for a given robot model
Element tables for a given robot model
RTM Analyzer
Computational Performance Models for a given robot model
Various robot tables library
Various robot models library
Performance Evaluation of the specified method

Table 3   Summary of RTM system's statements

Statement Type	Statement Structure
1. Sub-task title	SUBT,(no.),(title),(comment)
2. REPEAT control card	REP, no.of first operation ,TO, no.of last operation , no.times to repeat* ,(comment)
3. PARALLEL control card	PAR, no.of first operation ,TO, no.of last operation ,(comment)
4. Conditional branching	IF,(condition name.condition.value*),GOTO, operation no. subtask number
5. Control transfer	GOTO, operation no. subtask no. ,(comment)
6. Movement elements (Rn,Mn,ORn)	
a. Position initialization	(joints parameters)
b. By end-point of segments	(operation no.)(R.T.M. symbol),(comment) (velocity),(joints parameters)
or: c. By displacement	(operation no.),(R.T.M. symbol), A-Angular / D-Linear ,(velocity),(displacement)
7. All other R.T.M. elements	(operation no.),(R.T.M. symbol),(operation parameter),(comment)
8. END Card	END
9. CONDITION initialization	COND (condition name),(set of initial values*) END

* can be generated randomly

Examples: SA
T3 1
T3 2

4.2 Card 2: Task title

80 character limit - no format required.

4.3 Cards 3 - N: RTM Motion Descriptors

If the robot is the Stanford Arm or the T3 with input type 1, the first card of this section [card 3 of the standard input deck] is used to specify the robots initial position.

4.4 Final Card: END

End is written in the first three columns of the last card to specify the last card of the input deck.

5. RTM ELEMENT INPUT FORMATS:

For analysis of T3 tasks, RTM allows two means to convey motion length data to the program. Only M, R, and OR elements are affected by this distinction. SA input allows only one format.

5.1 T3 Input Format Type 1:

With this input type motion length is determined by changes in World Coordinates. World coordinates specify an end effecter position with X,Y,Z translation coordinates, and D,E,R rotation coordinates, all relative to the robot base.

Motion card format is as follows:

```
[serial #] [b] [motion type] [# of segments]
[velocity,X,Y,Z,D,E,R]    user comment
```

where:

Serial number	- an integer number assigned to this move. Serial numbers are assigned by the users to all task elements.
b	- one or more blanks
Motion type	- M = move arm with loaded gripper R = reach for object with unloaded gripper OR = orient the gripper [loaded or unloaded]
# of segments	- the number of linear segments through which the robot moves before coming to a complete stop. Velocity and coordinate cards must be specified for each segment.
User comment	- free format. Must begin in column 50 and end before or at column 79.
Velocity	- the velocity assigned to this motion. For the T3 this must be 1, 2, 5, 10, 20, 30, 40, or 50 ips.

When using this input type the first card of the motion descriptor section must hold six values representing the robot's initial position.

An example of type 1 input is shown in Figure 1.

5.2 T3 Input Format Type 2:

With this input type motion lengths are simply specified in inches (or degrees) with no reguard to relative location in the robot's work area. The format is as follows:

```
[serial #] [b] [motion type] [# of segments]
    [b] [velocity, motion length]
```

Items have the same definition as in type 1 input. Motion length is a real number specifying motion distance in inches (if motion type is M or R) or angle of rotation in degrees (for motion type OR). If a user comment is required an asterisk must be placed in column 80 of the above card. The next card must contain only a user comment, beginning in column 50 and ending at or before column 79.

An example of type 2 input is shown in Figure 2.

```
t3 1
simulation of loading and unloading a lathe
40, 13, -23. 15, 180, 0, 0
1 orl                                       go to bar stock
10, 60, 35, -29, 90, 35, 0
2 re                                        open fingers of tool #2
3 rl                                        lower fingers over bar stock
5, 65, 38, -29, 90, 35, 0
4 grl                                       grasp the bar stock
5 rl                                        raise in two stages
5, 65, 37, -20, 90, 35, 0
6 orl                                       move while rotating
10, 38, 22, -20, 90, 35, -90
7 ml                                        move to front of lathe
10, 69, 23, -26, 90, 35, -90
8 re                                        open fingers of tool #1
9 d 42.78                                   wait for lathe cover to open
10 ml                                       enter: bring fingers over part
2, 74, 26, -26, 90, 35, -90
11 grl                                      grasp the processed part
12 t 42.78                                  wait for chuck to open
13 ml                                       raise part
5, 74, 26, -20, 90, 35, -90
14 orl                                      rotate gripper 180 degrees
10, 74, 36, -20, 90, 35, 90
15 ml                                       lower arm
2, 34, 26, -26, 90, 35, 90
16 t 13.7                                   wait for chuck to close
17 re                                       release bar stock
18 ml                                       move out of lathe
5, 69, 23, -26, 90, 35, 90
19 grl                                      close fingers of tool #2
20 ml                                       move to intermediate point
10, 72, 10, -26, 9, 035, 90
21 orl                                      rotate gripper while moving
10, 83, 12, -26, 90, 35, 0
22 ml                                       move to part disposal
2, 83, 12, -29, 90, 35, 0
23 d 21.94                                  wait for release signal
24 re                                       release finished part
25 rl                                       raise up
5, 79, 9, -29, 90, 35, 0
26 ml                                       return to start position
10, 40, 13, -23. 15, 180, 0, 0
end
```

Figure 1 - Input type 1 for the T^3

```
t3 2
turning centers - load/unload and guaging with double gripper
1 ri 5,4.1                                                    *
                                         reach for bar stock part
2 gri                                    grasp it, identify by diameter
3 mi 5,10.2                                                   *
                                         raise the part
4 mi 30,97.2                                                  *
                                         move part to required machine
5 d 116.7                                wait for machine cover to open
6 mi 20,28.7                                                  *
                                         move to part in machine
7 gri                                    grasp finished part (2nd grip)
8 d 166.6                                wait for chuck to retract
9 mi 20,1.0                                                   *
                                         remove finished part
10 mi 20,7.0                                                  *
                                         move out of machine
11 ori 50,90.0                                                *
                                         flip gripper
12 mi 20,7.0                                                  *
                                         move back into machine
13 mi 1,1.0                                                   *
                                         place new part in machine
14 d 100.0                               wait for chuck to hold part
15 re                                    release part
16 mi 20,22.8                                                 *
                                         move out of machine
17 d 100.0                               wait for cover to close
18 mi 30,70.0                                                 *
                                         move to guaging station
19 mi a,10,5.0,a,2,2.0                                        *
                                         place part in guage
20 re                                    release part
21 d 10.0                                guaging cycle
22 gri                                   grasp guaged part
23 mi 5,4.0                                                   *
                                         move to depart from guage
24 mi 20,20.0                                                 *
                                         move to finished part exit
25 mi 5,4.0                                                   *
                                         place part on exit rack
26 re                                    release finished part
27 mi 20,4.0                                                  *
                                         raise arm
end
```

Figure 2 - Input type 2 for the T^3

5.3 General T3 Input:

Inputs not affected by the choice of input type are Grasp, Release, Delay, and Vision. Formats are as follows:

Grasp:

```
[serial #] [b] [GR1]                             user comment
```

gripper closing is simulated

Release:

```
[serial #] [b] [RE]                              user comment
```

gripper opening is simulated

Time Delay:

```
[serial #] [b] [T] [b] [delay time in tmu]
```

add given time to the total task time, where the robot is actively involved in processing.

Delay:

```
[serial #] [b] [D] [b] [delay time in tmu]       user comment
```

add a given time to the total task time, where the robot is waiting for process completion.

Vision:

```
[serial #] [b] [VI] [b] [vision cycle time]      user comment
```

a value representing a vision cycle time is added to the total task time

All user comments must begin in column 50 and end at or before column 79.

5.4 SA Motion Input:

Motions of the stanford arm can be described to the program in three ways.

1. The user can specify a three component vector describing the direction in which the motion is performed. This input type is useful for one segment moves and reaches.
2. The motion's six component endpoint location (in world coordinates) can be specified. This input type is used for orient motions and

moves which combine either a move or a reach with an orient.

3. The motions departure and approach vectors can be specified, in addition to the motion's end point, to fully specify a 3 segment motion.

All three types of input use a single format which is as follows:

```
[Serial #][b][mot.type][# of segments][b][spec.time][X,Y,Z,D,E,R][*]
[X,Y,Z],[X,Y,Z]                                        [user comment]
```

Where:

Serial #	-	as before
b	-	as before
Motion type	-	M move with loaded gripper R reach with unloaded gripper OR orient loaded or unloaded gripper
# of segments	-	number of segments of this move. Limited to either 1 or 3 for the standard arm.
Specified time	-	the time, specified by the user, that this motion should take. This is the SA's velocity control mechanism. If this value is zero, the SA will operate at its maximum velocity.
X,Y,Z,D,E,R	-	use only X,Y,Z if input type one is used. Input all six values if type two is used.
*	-	asterisk placed in column eighty if comment is to follow on next card. This can be omitted if there is no comment.
X,Y,Z,X,Y,Z	-	departure and approach vectors. These can be omitted if input type three is not used.
User comment	-	user comment beginning after column 50

5.5 General SA input:

All other RTM elements for the Stanford arm have at most one parameter. GR1 and RE elements have length as their parameter. T, V, D, and GR2 have their operation time as a parameter. SE1, SE2, and SF have no parameter. The input format for these elements is:

[Serial #] [b] [element type] [b] [operation parameter] [comment]

where:

Serial number and b are as before. Element type and operation parameter are defined above. And comment is a user comment which begins after column fifty. An example of SA input is shown in Figure 3.

6. CONTROL STRUCTURES:

In addition to motion by motion input, RTM allows various logical input structures to permit more versatile task descriptions. These are:

1. Repeat
2. Parallel
3. Goto
4. If (condition) goto

Formats are given below:

6.1 Repeat:

Repeat a set of instructions (serial numbers I through J) a given (N) number of times. Format is as follows:

[REP] [b] [I] [b] [TO] [b] [J] [b] [N] [b] [TIMES]

Where:

b	-	blank
I	-	the serial number of the first element of the repeat block
J	-	the serial number of the last element of the repeat block
N	-	the number of times the block is to be repeated

The REP card should directly proceed input statement I.

6.2 Parallel:

Perform instructions I through J in parallel. The time recorded for the parallel block will be that of the longest operation in the block. Format is as follows:

```
sa
bring pin back to place
30.75,31,2.5,90,90,0
1 r3 0,15.45,32.35,5.1,135,90,0
0,0,3,-1,-1,4                                       go to pin
2 seP                                               stop there
3 grJ 0.5                                           grasp it
4 mi 0,0,0,1                                                                    *
                                                    release pin from place
5 sf
6 m3 0,28.2,29.88,2.7,90,90,0
-1,-1,4,0,0,3                                       go above pin store location
7 seP                                               stop there
8 mi 0,0,0,1                                                                    *
                                                    enter pin into hole
9 sf                                                stop when entered into place
10 re 0.375                                         release pin
end
```

Figure 3 - Input for the Stanford Arm

[PAR] [b] [I] [b] [TO] [b] [J]

Where:

I	- the serial number of the first element of the parallel block
J	- the serial number of the last element of the parallel block

6.3 <u>Goto</u>:

Change the logical flow of the program. Format is as follows:

[GOTO] [b] [goto descriptor]

Where:

goto descriptor	- can either be an element serial number, a subtask, or the end of the program (END)

6.4 <u>If</u> <u>(condition)</u> <u>goto</u>:

Take various paths throughout the program based on prespecified condition values. Format is as follows:

[IF] [b] [(] [b] [Cond ID] [b] [.comparator.]
    [b] [Cond value] [)] [b] [GOTO]
                                                    [goto descr]

Where:

Cond ID	- a prespecified conditional identifier (see later section)
comparator	- a standard Fortran comparator [.EQ. .NE. .LE. .GE. .LT. .GT.]
Cond Value	- a value previously assigned to the Cond ID
Goto descr	- goto descriptor -- as above (included on the same line as the IF statement)

6.5 <u>Conditional</u> <u>identifiers</u> <u>and</u> <u>values</u>:

Conditional identifiers and values must be declared near the beginning of the input deck, directly after the Task Title. Format is as follows:

```
[*COND]
[COND ID#1] [b] [CONDVAL#11,CONDVAL#12,......CONDVAL#1N]
[COND ID#2] [b] [CONDVAL#21,CONDVAL#22,......CONDVAL#2M]
      .
      .
[COND ID#J] [b] [CONDVAL#J1,CONDVAL#J2,......CONDVAL#JP]
[*END]
```

The number of condition values declared for a single condition ID must be equal to the number of times that condition is going to be examined (tested) during the program (or N,M,and P do not have to be equal). Condition values are used sequentially, in other words the first time COND ID#1 is examined it will have the value of CONDVAL#11, the second time it is examined it will have the value of CONDVAL#12, and so forth. If conditions are not being used in an application the entire condition declaration section can be omitted.

6.6 Subtasks:

A subtask is a set of motions which can be thought of as being complete and distinct from the entire task being examined. They are generally more applicable to robots with sensory inputs and decision/branch capabilities. Subtasks in RTM are analogous to Fortran statement labels. These labels can be accessed by GOTO and IF(cond) GOTO statements. Examples of all control structures are given in Figure 4.

7. OUTPUT STRUCTURE

RTM generally supplies a line of output for each input element. Included in the standard output line for the T-3 is:

1. Serial Number
2. Element Type
3. Number of Segments
4. Motion Length in Inches (or degrees)
5. Motion Velocity in Inches per Second
6. Motion Time in TMU (see below)
7. User Comment
8. RTM Comment

Control structure output frequently incites RTM comments which explain the execution. This output is best explained through the example outputs of Figures 5, 6, and 8. These figures correspond to the input of Figures 1, 2, and 4.

```
t3 2
demonstration of control structures - sort and palletize
*cond
nopart 0,1,0,0,1,1
norep 0,0,0,1
*end
subt1
1 r1 10,4.0
                                                  move to part on feeder
2 gr1                                             grasp part
3 m1 50,15.0
                                                  raise part
if (nopart .eq. 1) goto subt2
4 m1 30,35.0
                                                  move to part type 1 washer
5 m1 1,10.0
                                                  insert part into washer
rep 6 to 7 3 times
6 m1 5,5.0
                                                  scrub part with multiple
7 m1 5,5.0
                                                  up and down motions
8 m1 30,10.0
                                                  raise out of washer
9 m1 50,15.0
                                                  move to type 1 pallet
10 m2 a,30,4.0,a,10,2.0
                                                  lower part to pallet in 2 seg
11 re                                             release part
12 r1 20,5.0
                                                  raise arm
13 r1 50,30.0                                                      *
                                                  return to feeder
if (norep .eq. 1) goto end
goto subt1
subt2
par 1 to 2
1 m1 50,60.0                                                       *
                                                  move to type 2 pallet
2 or1 50,180.0                                                     *
                                                  and invert casting
3 m2 a,30,5.0,a,5,2.0                                              *
                                                  lower part to pallet in 2 segs
4 re                                              release part
5 r1 30,5.0                                                        *
                                                  raise arm
6 r1 50,55.0                                                       *
                                                  return to feeder
if (norep .eq. 1) goto end
goto subt1
end
```

Figure 4 - Examples of RTM Control Structures

the tomorrow tool

simulation of loading and unloading a lathe

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	1	ori	90.00	10.00	97.4	go to bar stock	
	2	re			.3	open fingers of tool #2	
	3	ri	3.83	5.00	42.5	lower fingers over bar stock	
	4	gri			.3	grasp the bar stock	
	5	ri	9.06	5.00	60.4	raise	
	6	ori	90.00	10.00	97.4	move while rotating	
	7	mi	31.59	10.00	97.9	move to front of lathe	
	8	re			.3	open fingers of tool #1	
	9	d			42.8	wait for lathe cover to open	
	10	mi	5.83	2.00	91.1	enter: bring tool over part	
	11	gri			.3	grasp the processed part	
	12	t			42.8	wait for chuck to open	
	13	mi	6.00	5.00	43.5	raise part	
	14	ori	180.00	10.00	184.7	rotate gripper 180 degrees	
	15	mi	41.67	2.00	588.8	lower arm	
	16	t			13.7	wait for chuck to close	
	17	re			.3	release bar stock	
	18	mi	35.13	5.00	205.3	move out of lathe	

Figure 5

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	19	gri			.3	close fingers of tool #2	
	20	ml	81.00	10.00	88.7	move to intermediate point	
	21	ori	90.00	10.00	97.4	rotate gripper while moving	
	22	ml	3.00	2.00	51.8	move to part disposal	
	23	d			21.9	wait for release signal	
	24	re			.3	release finished part	
	25	rl	5.00	5.00	37.9	raise up	
	26	ml	39.64	10.00	120.2	return to start position	

```
the total time for the task is            2028.3 [tmu] =    73.0 [sec] =      1.2 [min]
```

Figure 5 (continued)

the tomorrow tool

turning centers - load/unload and guaging with double gripper

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	1	r1	4.10	5.00	32.9	reach for bar stock part	
	2	gr1			.3	grasp it: identify by diameter	
	3	m1	10.20	5.00	66.8	raise the part	
	4	m1	97.20	30.00	100.1	move part to required machine	
	5	d			116.7	wait for machine cover to open	
	6	m1	28.70	20 00	50.0	move to part in machine	
	7	gr1			.3	grasp finished part (2nd grip)	
	8	d			166.6	wait for chuck to retract	
	9	m1	1.00	20.00	11.5	remove finished part	
	10	m1	7.00	20.00	22.3	move out of machine	
	11	or1	90.00	50.00	27.6	flip gripper	
	12	m1	7.00	20.00	22.3	move back into machine	
	13	m1	1.00	1.00	37.9	place new part in machine	
	14	d			100.0	wait for chuck to hold part	
	15	re			.3	release part	
	16	m1	22.80	20.00	41.8	move out of machine	
	17	d			100.0	wait for cover to close	
	18	m1	70.00	30.00	75.0	move to guaging station	

Figure 6.

```
control  serial   rtm    motion   motion   operation  user comment                          comment
card      no.    symbol  length  velocity    time
                          [in]    [ips]     [tmu]
-----------------  ---------------------------------------------------------------------------------

          19      m2      7.00    10.00      37.9     place part in guage

          20      re                           .3     release part

          21      d                          80.0     guaging cycle

          22      gr1                          .3     grasp guaged part

          23      m1      4.00     5.00      32.4     move to depart from guage

          24      m1     20.00    20.00      37.9     move to finished part exit

          25      m1      4.00     5.00      32.4     place part on exit rack

          26      re                           .3     release finished part

          27      m1      4.00    20.00      17.0     raise arm


 the total time for the task is           1210.7 [tmu] =    43.6 [sec] =      .7 [min]
-------------------------------------------------------------------------------------------
```

Figure 6 (continued)

stanford - arm

bring pin back to place

control card	serial no.	rtm symbol	motion length [in]	operation time [tmu]	user comment	comment
	1	r3	16.01	40.3	go to pin	move instruction
	2	se2		22.0	stop there	
	3	gr1	.50	6.3	grasp it	
	4	m1	1.00	8.0	release pin from place	change instruction
	5	sf		7.0		
	6	m3	18.07	36.3	go above pin store location	move instruction
	7	se2		22.0	stop there	
	8	m1	1.00	8.0	enter pin into hole	change instruction
	9	sf		7.0	stop when entered into place	
	10	re	.37	5.5	release pin	

the total time for the task is 162.4 [tmu] = 5.8 [sec] = .1 [min]

Figure 7

the tomorrow tool

demonstration of control structures - sort anu palletize

subtask number 1

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	1	r1	4.00	10.00	21.2	move to part on feeder	
	2	gr1			.3	grasp part	
	3	m1	15.00	50.00	22.3	raise part	
if							if (nopart .eq. 1.00) go to the start of subtask # 2 the cond value = 0 so do not go
	4	m1	35.00	30.00	42.5	move to part type 1 washer	
	5	m1	10.00	1.00	287.9	insert part into washer	
rep							repeat from 6 to 7 3 times
	6	m1	5.00	5.00	37.9	scrub part with multiple	
	7	m1	5.00	5.00	37.9	up and down motions	

Figure 8

```
control  serial   rtm    motion   motion   operation  user comment                              comment
card      no.    symbol  length  velocity    time
                          [in]     [ips]     [tmu]
--------------------------------------------------------------------------------------------------------------

                                               75.8                                      one repeat cycle time

                                   227.5                                       total repeat time

           8      m1     10.00    30.00       22.3    raise out of washer

           9      m1     15.00    50.00       22.3    move to type 1 pallet

          10      m2      6.00    30.00       12.5    lower part to pallet in 2 segs

          11      re                            .3    release part

          12      r1      5.00    20.00       17.0    raise arm

          13      r1     30.00    50.00       26.8    return to feeder
if                                                                             if (norep   .eq.    1.00)
                                                                                go to end of task
                                                                                the cond value =      0 so do not go

goto                                                                            go to the start of subtask @ 1
 the total time of the subtask is    702.9 [tmu] =    25.3 [sec]
```

Figure 8 (cont.)

```
  subtask  number   1
  --------------------


control  serial   rtm     motion   motion    operation   user comment                          comment
card      no.    symbol   length  velocity     time
                           [in]    [ips]      [tmu]
-----------------------------------------------------------------------------------------------------------

          1       r1       4.00    10.00       21.2    move to part on feeder

          2       gr1                            .3    grasp part

          3       m1      15.00    50.00       22.3    raise part

if                                                                       if (nopart  .eq.     1.00)
                                                                          go to the start of subtask # 2
                                                                          the cond value =    1.00 so go
 the total time of the subtask is      43.8 [tmu] =      1.6 [sec]



  subtask  number   2
  --------------------


control  serial   rtm     motion   motion    operation   user comment                          comment
card      no.    symbol   length  velocity     time
                           [in]    [ips]      [tmu]
-------- ------------- ----------------------------------------------------------------------------------

par                                                                      activities  1 to  2 performed in paralel

          1       m1      60.00    50.00       43.5    move to type 2 pallet

          2       ori    180.00    50.00       45.0    and invert casting

                                               45.0                            the operation time is that of
                                                                               activity #  2 ,the largest activity
```

Figure 8 (cont.)

```
control  serial   rtm    motion   motion   operation   user comment                                   comment
card      no.    symbol  length  velocity    time
                          [in]    [ips]     [tmu]
--------------------------------------------------------------------------------------------------------------------

            3      m2      7.00    30.00        8.5   lower part to pallet in 2 segs

            4      re                            .3   release part

            5      r1      5.00    30.00       17.0   raise arm

            6      r1     55.00    50.00       40.7   return to feeder
if
                                                                                   if (norep   .eq.    1.00)
                                                                                    go to end of task
                                                                                    the cond value =      0 so do not go

goto
 the total time of the subtask is     111.5 [tmu] =    4.0 [sec]                    go to the start of subtask # 1



   subtask  number    1
   ---------------------


control  serial   rtm    motion   motion   operation   user comment                                   comment
card      no.    symbol  length  velocity    time
                          [in]    [ips]     [tmu]
------ -------------------------------------------------------------------------------------------------------------

            1      r1      4.00    10.00       21.2   move to part on feeder

            2      gr1                           .3   grasp part

            3      m1     15.00    50.00       22.3   raise part
if
                                                                                   if (nopart  .eq.    1.00)
                                                                                    go to the start of subtask # 2
                                                                                    the cond value =      0 so do not go

            4      m1     35.00    30.00       42.5   move to part type 1 washer
```

Figure 8 (cont.)

```
control   serial   rtm     motion    motion    operation   user comment                              comment
card       no.    symbol   length   velocity     time
                            [in]     [ips]       [tmu]
----------- ----- ------------------------------------------------------------------------------------------------

             5      m1      10.00     1.00       287.9    insert part into washer

rep                                                                              repeat from  6 to  7       3 times

             6      m1       5.00     5.00        37.9    scrub part with multiple

             7      m1       5.00     5.00        37.9    up and down motions

                                                  75.8                                  one repeat cycle time

                                      227.5                                      total repeat time

             8      m1      10.00    30.00        22.3    raise out of washer

             9      m1      15.00    50.00        22.3    move to type 1 pallet

            10      m2       6.00    30.00        12.5    lower part to pallet in 2 segs

            11      re                              .3    release part

            12      r1       5.00    20.00        17.0    raise arm

            13      r1      30.00    50.00        26.8    return to feeder

if                                                                               if (norep   .eq.    1.00)
                                                                                  go to end of task
                                                                                  the cond value =      0 so do not go

goto                                                                              go to the start of subtask # 1
 the total time of the subtask is    702.9 [tmu] =    25.3 [sec]
```

Figure 8 (cont.)

```
   subtask  number    1
   --------------------


control  serial   rtm    motion   motion   operation   user comment                               comment
card      no.    symbol  length  velocity    time
                          [in]    [ips]      [tmu]
-------------------------------------------------------------------------------------------------------------------------

           1.     r1      4.00    10.00       21.2     move to part on feeder

           2      gr1                           .3     grasp part

           3      m1     15.00    50.00       22.3     raise part

if                                                                                if (nopart  .eq.    1.00)
                                                                                   go to the start of subtask # 2
                                                                                   the cond value =      0 so do not go

           4      m1     35.00    30.00       42.5     move to part type 1 washer

           5      m1     10.00     1.00      287.9     insert part into washer

rep                                                                                repeat from  6 to  7       3 times

           6      m1      5.00     5.00       37.9     scrub part with multiple

           7      m1      5.00     5.00       37.9     up and down motions

                                              75.8                                        one repeat cycle time

                                  227.5                                            total repeat time

           8      m1     10.00    30.00       22.3     raise out of washer

           9      m1     15.00    50.00       22.3     move to type 1 pallet

          10      m2      6.00    30.00       12.5     lower part to pallet in 2 segs

          11      re                            .3     release part

          12      r1      5.00    20.00       17.0     raise arm

          13      r1     30.00    50.00       26.8     return to feeder

if                                                                                if (norep   .eq.    1.00)
                                                                                   go to end of task
                                                                                   the cond value =   1.00 so go
```

Figure 8 (cont.)

```
the total time of the subtask is     702.9 [tmu] =    25.3 [sec] =       .4 [min]

the total time for the task is      2263.9 [tmu] =    81.5 [sec] =      1.4 [min]
-------------------------------------------------------------------------------------
```

Figure 8 (cont.)

Stanford Arm output is similar to T3 output except that motion velocity is not included. An example of SA output is shown in Figure 7. This output corresponds to the input of Figure 3.

The time for the total task is summed at the end of the output. The sum is presented in TMU (Time Motion Units; 1 TMU = 0.036 Sec), seconds, and minutes.

8. RTM'S DISTRIBUTION SAMPLING CAPABILITIES

8.1 Introduction

The capability of sampling from distributions to define parameters previously input as constants by the user has been added to a new version of RTM. This capability makes RTM a more effective modeler of robotic tasks. A task where this capability would be realistic would be, for instance, one in which the robot waits for parts arriving along a conveyor at random intervals. After a part arrives the robot would escort it through its processing. The distributions available, the syntax and some examples of their use are explained in this section.

8.2 Available Distributions

The distributions currently available are listed in Table 4. They include the uniform, normal, triangular, and exponential distributions. These were judged to be the most important in the modeling of robotic tasks, and were included in this probabilistic extension of RTM for that reason. These distributions may be sampled to define the variables listed in Table 5. These variables include:

1. length of a delay;
2. distance the arm is to move;
3. number of times a block of RTM primitives is to be repeated, and
4. sequential conditional values for a given conditional identifier.

The above constitute all the situations for which distribution sampling would be practical for RTM.

8.3 Distribution Declaration Syntax

Distributions and their parameters are specified before any motion primitives are input, and proceed specification of conditional identifiers. The specification format is as follows:

```
$DIST #, type, parameters
```

Distribution	Required Parameters
1. Uniform (U)	minimum and maximum boudaries
2. Normal (N)	mean and standard deviation
3. Triangular (T)	minimum, mode, and maximum
4. Exponential (E)	mean

Table 4 Probability Distributions Applied by RTM

RTM variables for which distribution
samplings can be substituted:

1. Time of delay for the RTM operatives T, D, and Vi.

2. Distance of a reach (R), move (M), or an Orientation (OR). Note this is only applicable to type 2 RTM input, where velocity and distance are specified.

3. The number of times a repeat block is to be performed

4. The sequential values representing a conditional identifier.

Table 5

where:

#	- The distribution identification number. It is assigned and referenced by the user later in the input to specify which distribution should be sampled.
type	- A single capital letter specifying distribution type. N denotes Normal, E denotes Exponential, U denotes Uniform, and T denotes Triangular.
Parameters	- as required by the distribution type in the order shown in Table 4. Parameters should be separated by commas.

8.4 Substituting Samples for Constants

For cases one through three of Table 5, distributions are accessed by the user specifying "D#" (# is the identification number of the requested distribution) in place of the motion, time, or repeat parameter.

For example:

```
T D4
```

would use a sample from distribution four as the time to be delayed.

```
M1 10,D3
```

would use a sample from distribution three as the motion length of the one segment move. 10 specifies the motion velocity.

```
REP 3 TO 7 D2 TIMES
```

would sample distribution two, round the sample to the nearest integer, and repeat instructions 3 through 7 that number of times.

Case four of Table 5 is handled differently. When not using the distribution sampling capabilities, the user specifies a conditional identifier, followed by the list of sequential values that identifier is to take on when examined. Using the distribution sampling capability, the user can now generate that list of sequential values by using the following format:

```
CONDID "D"#,numval
```

where:

CONDID	- is the conditional identifier (as before)
#	- is the identification number of the distribution to be sampled.
numval	- is the number of conditional values to generate. The distribution will be sampled this many time and each sampling will be rounded to the nearest integer.

Note that when RTM expects an integer, the sample from the distribution is rounded to the nearest integer. This occurs in the case of block repetition definition, and conditional value generation.

8.5 Example

An example of the use of distribution sampling with RTM is shown in Figure 9. Three distributions are declared. The first is normal with a mean of ten and a standard deviation of three. The second is uniform with a minimum of thirty, and a maximum of fifty. The third is uniform with a minimum of sixty and a maximum of ninety.

The first distribution is used to determine the number of times the block of primitives from serial number 1 to serial number 29 is repeated. The second distribution determines the amount of time the robot waits before a part arrives (input instruction #1). The third distribution is used to determine the cycle time of the gauging device (input instruction #22).

The output from this example is shown in Figure 10. The sample of distribution 1 yielded a value of 8 when rounded to the nearest integer. Thus, the block of primitives is repeated 8 times. The sample of distribution 2 returned a value of 42.7, thus the robot waits 42.7 TMU before the part arrives. The sample of distribution 3 returned a value of 68.3, which is the time it takes for the gauging cycle to complete.

9. STUDY OF AUTOMATIC INTERFACING BETWEEN RTM AND RCCL

9.1 Introduction

RTM (Robot Time and Motion) is a software system which estimates the time a robot takes to perform work motions, as described above. Its input describes the points in space through which the robot moves and the actions performed at some of these locations (eg. close gripper). One objective of RTM is to allow the user to optimize the motions of a given task (ie. minimize the cycle time) without requiring actual operation of the robot. A current limitation of RTM is that after describing the task to the simulator and optimizing it, the user must

```
t3 2
$dist 1,n,10.,3.
$dist 2,u,30.,50.
$dist 3,u,60.,90.
turning centers - load/unload and guaging with double gripper
rep 1 to 29 d1 times
1 d d2                                             wait for part to arrive
                                                                           *
2 rl 5,4.1
                                                   reach for bar stock part
3 grl                                              grasp it; identify by diameter
4 ml 5,10.2                                                                *
                                                   raise the part
5 ml 30,97.2                                                               *
                                                   move part to required machine
6 d 116.7                                          wait for machine cover to open
7 ml 20,28.7                                                               *
                                                   move to part in machine
8 grl                                              grasp finished part (2nd grip)
9 d 166.6                                          wait for chuck to retract
10 ml 20,1.0                                                               *
                                                   remove finished part
11 ml 20,7.0                                                               *
                                                   move out of machine
12 wrl 50,90.0                                                             *
                                                   flip gripper
13 ml 20,7.0                                                               *
                                                   move back into machine
14 ml 1,1.0                                                                *
                                                   place new part in machine
15 d 100.0                                         wait for chuck to hold part
16 re                                              release part
17 ml 20,22.8                                                              *
                                                   move out of machine
18 d 100.0                                         wait for cover to close
19 ml 30,70.0                                                              *
                                                   move to guaging station
20 m2 a,10,5.0,a,2,2.0                                                     *
                                                   place part in guage
21 re                                              release part
22 d d3                                            guaging cycle
23 grl                                             grasp guaged part
24 ml 5,4.0                                                                *
                                                   move to depart from guage
25 ml 20,20.0                                                              *
                                                   move to finished part exit
26 ml 5,4.0                                                                *
                                                   place part on exit rack
27 re                                              release finished part
28 ml 20,4.0                                                               *
                                                   raise arm
29 rl 30,27.
end
```

Figure 9

the tomorrow tool

turning centers - load/unload and guaging with double gripper

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
rep							repeat from 1 to 29 8 times
	1	d			42.7	wait for part to arrive	
	2	r1	4.10	5.00	32.9	reach for bar stock part	
	3	gr1			.3	grasp it; identify by diameter	
	4	m1	10.20	5.00	66.8	raise the part	
	5	m1	97.20	30.00	100.1	move part to required machine	
	6	d			116.7	wait for machine cover to open	
	7	m1	28.70	20.00	50.0	move to part in machine	
	8	gr1			.3	grasp finished part (2nd grip)	
	9	d			166.6	wait for chuck to retract	
	10	m1	1.00	20.00	11.5	remove finished part	
	11	m1	7.00	20.00	22.3	move out of machine	
	12	or1	90.00	50.00	27.6	flip gripper	
	13	m1	7.00	20.00	22.3	move back into machine	
	14	m1	1.00	1.00	37.9	place new part in machine	
	15	d			100.0	wait for chuck to hold part	
	16	re			.3	release part	
	17	m1	22.80	20.00	41.8	move out of machine	

Figure 10

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	18	d			100.0	wait for cover to close	
	19	m1	70.00	30.00	75.0	move to guaging station	
	20	m2	7.00	10.00	37.9	place part in guage	
	21	re			.3	release part	
	22	d			68.3	guaging cycle	
	23	gr1			.3	grasp guaged part	
	24	m1	4.00	5.00	32.4	move to depart from guaye	
	25	m1	20.00	20.00	37.9	move to finished part exit	
	26	m1	4.00	5.00	32.4	place part on exit rack	
	27	re			.3	release finished part	
	28	m1	4.00	20.00	17.0	raise arm	
	29	r1	27.00	30.00	35.1	return to input conveyor	
					1276.8		one repeat cycle time
					10214.5		total repeat time

the total time for the task is 10214.5 [tmu] = 367.7 [sec] = 6.1 [min]

Figure 10 (cont.)

again describe the motions to the robot control language, thereby unnecessarily repeating some input information. It is the purpose of this section to outline a data translator which will modify the RTM task description into actual robot control instructions. The feasibility and requirements for such translation are studied, but no program has been developed based on the results of this study. The robot control language used is described below.

9.2 RCCL

RCCL, a Robot Control "C" Library, is a library of data manipulation and robotic execution functions. These functions can be called by a user's program to manipulate a robotic arm. Basic use of the library entails manipulation of homogeneous transformations to represent points in space the robot is to reach. The capacity is provide by the library for the storing and manipulation of these transforms, as well as generation of robot arm position by solving transform equations. Background in homogeneous transforms and their manipulation is offered in Paul [10]. A complete discussion of RCCL and a user's manual are presented by Hayward in [11, 12, and 13].

A simple example of the use of RCCL, taken from Hayward's manual [13] is shown in Figure 11. The variables to be used are first declared, two as transforms (t and e) and one as a position (p0). The two transforms are then initialized with the gentr_ statements. The position variable is then set to correspond to the solution for the t6 component of the transform equation t*t6=b. The first move call instructs the robot to move to position p0. The second move call instructs the robot to move to a system-kept position, called park.

This example is intended to show the general structure of an RCCL program. The reader is directed to Hayward's "RCCL User's Manual" [13] for any actual working knowledge of the library.

9.3 RTM Primitive's Link to RCCL

The RTM input language consists of ten primitive operators. These are shown in Table 1. As can be seen, some operators describe motion, (M, R, and OR), some describe actions (SF, SE, GR, and RE), and some describe extraneous processing for which the robot must be delayed (D, T, and VI). To modify this input into RCCL compatible input, a system must output the RCCL instructions which would perform the same operations. Table 6 shows each RTM operator translated into equivalent RCCL commands. It should be noted here that RCCL, as an actual control program, is a much more detailed descriptor of tasks than is the input language of RTM. Thus, while the RCCL translation of the RTM input will perform the required tasks, it is by no means the most graceful way to accomplish the task using RCCL.

```
RTM descriptor                RCCL equivalent
--------------                ---------------


Translation and orientation motions:

  R1                          e=gentr_trsl(0.,0.,tool length);
                              b=gentr_trsl("b",x,y,z);
  M1                          p=makeposition("p",t6,e,EQ,b,TL,e);
                              move(p);
                              stop(0);


  OR1                         as above but
                              b=gentr_rpy("b",x,y,z,d,e,r);



  Rn                          for(i=1; i<=n; ++i) {
                                 b=gentr_trls("b",x.i,y.i,z.i);
  Mn                             p=makeposition("p",t6,e,EQ,b,TL,e);
                                 move(p);
                                 freepos(p);
                                 freetrans(b);
                              }
                              stop(0);


  ORn                         as above but:
                              b=gentr_rpy("b",x.i,y.i,z.i,d.i,e.i,r.i)

  parameters x, y, z, d, e, and r come from position
  description in world coordinates.  tool length must
  be assumed or input in later phase.  If input is in
  displacement-velocity format, direction information
  must be gained externally.  RCCL does not distinguish
  between a reach with an unloaded end effector (Rn) and
  a move with a loaded end effector (Mn).
----------------------------------------------------------------

  SE                          Not able to be modeled by RCCL

----------------------------------------------------------------
```

Table 6 RCCL Equivalents of RTM Primitive Operators

Stop on force, torque, both

SF1 stop on force	limit("fdir",force); move(p);
SF2 stop on torque	limit("tdir",torque"); move(p);
SF3 stop on either force or torque	limit("fdir","tdir",force,torque); move(p);
SF4 stop on touch (same as stop on force with low threshold)	limit("fx","fy","fz",1,1,1); move(p);

fdir = fx, fy, or fz meaning force in x, y, or z direction
tdir = tx, ty, or tz meaning torque in x, y, or z direction
force is specified in newtons. torque is specified in newton-meters.

Delay, time delay, and vision:

D	stop(delay time);
T	" "
Vi	" "

Stop command follows a move command. Delay time is specified in milliseconds.

Gripper actions:

GR	move(p); waitfor(p -> end); CLOSE;
RE	move(p); waitfor(p -> end); OPEN;

High level gripper actions cannot be modeled.

Table 6 (cont.)

```
t3 ?
2 cinturn turning centers - load/unload and guaging with double gripper
$dist 1,u,0.,1.
$dist 2,n,80.,22.
$dist 3,n,50.,10.
#cond
nopart d1,5
norep 0,0,0,0,1
#end
subt1
1 rl 5,4.1                                                                    *
                                               reach for bar stock part
2 grl                                          grasp it, identify by diameter
if (nopart .eq. 1) goto subt2
3 ml 5,10.2                                                                   *
                                               raise the type 1 part
4 ml 30,97.2                                                                  *
                                               move part to first cinturn
5 d 116.7                                      wait for cinturn cover to open
6 ml 20,28.7                                                                  *
                                               move to part in cinturn
7 grl                                          grasp finished part (2nd grip)
8 d 166.6                                      wait for collet to retract
9 ml 20,1.0                                                                   *
                                               remove finished part
10 ml 20,7.0                                                                  *
                                               move out of cinturn
11 orl 50,90.0                                                                *
                                               flip gripper
12 ml 20,7.0                                                                  *
                                               move back into cinturn
13 ml 1,1.0                                                                   *
                                               place new part in cinturn
14 d 100.0                                     wait for collet to hold part
15 re                                          release part
16 ml 20,22.8                                                                 *
                                               move out of cinturn
17 d 100.0                                     wait for cover to close
18 ml 30,70.0                                                                 *
                                               move to guaging station
19 m? a,10.5.0,a,2,2.0                                                        *
                                               place part in guage
20 re
21 d d2                                        guaging cycle
22 grl                                         grasp guaged part
23 ml 5,4.0                                                                   *
                                               move to depart from guage
24 ml 20,20.0                                                                 *
                                               move to finished part disposal
25 ml 5,4.0                                                                   *
                                               place part on disposal rack
26 re                                          release finished part
27 rl 10,4.0                                                                  *
                                               raise arm
28 rl 30,62.                                                                  *
if (norep .eq. 0) goto subt1
goto end
```

Figure 11

```
subt2
1 ml 5,10.2                                                   *
                              raise the type 2 part
2 ml 30,36.2                                                  *
                              move part to second cinturn
3 d 116.7                     wait for cinturn cover to open
4 ml 20,28.7                                                  *
                              move to part in cinturn
5 gr1                         grasp finished part (2nd grip)
6 d 166.6                     wait for collet to retract
7 ml 20,1.0
                                                              *
8 ml 20,7.0                   remove finished part
                                                              *
9 wr1 50,90.0                 move out of cinturn
                                                              *
10 ml 20,7.0                  flip gripper
                                                              *
11 ml 1,1.0                   move back into cinturn
                                                              *
12 d 100.0                    place new part in cinturn
13 re                         wait for collet to hold part
14 ml 20,22.8                 release part
                                                              *
15 d 100.0                    move out of cinturn
16 ml 30,85.0                 wait for cover to close
                                                              *
17 m2 a,10,5.0,a,2,2.0        move to guaging station
                                                              *
18 re                         place part in guage
19 d d3
20 gr1                        guaging cycle
21 ml 5,4.0                   grasp guaged part
                                                              *
22 ml 20,20.0                 move to depart from guage
                                                              *
23 ml 5,4.0                   move to finished part disposal
                                                              *
24 re                         place part on disposal rack
25 r1 10,4.0                  release finished part
                                                              *
33 r1 30,70.                  raise arm
                                                              *
if (norep .eq. 0) goto subt1  return to feeder
end
```

Figure 11, continued

9.4 RTM Control Link to RCCL

In addition to primitive operators, RTM allows some task control commands, as shown in Table 7. These control commands allow the user to define Repeat blocks, which are collections of primitives repeated a number of times; Parallel blocks, which are a collection of primitives performed simultaneously, and Conditional Branching. The translation of the control commands to RCCL is shown in Table 8.

9.5 Example

An example translation of RTM input to RCCL input is shown in Figures 12 and 13. (The RTM input is shown in Figure 12, the RCCL translation is shown in Figure 13.) For easy comparison of the two, distances of the RTM input have been kept in inches in the RCCL translation. An actual RCCL program would require these distances to be specified in millimeters. Delay times, however, have been translated from TMU (time measurement units of RTM) into milliseconds. A number of observations can be made from this example. First, it can be seen that the translating program must examine the entire RTM input deck before translating any of it. This first pass is to count the number of positions for which RCCL requires transform and position pointers to be declared. A second pass is required in order to actually generate the RCCL input. Second, the robot's tool length and the zero point of its world coordinates (e and z transforms of Figure 13) must be assumed according to robot type. This can be easily accomplished since robot type is specified in the first line of the RTM input. Third, this example assumes that the dual gripper commands OPEN1, OPEN2, CLOSE1, and CLOSE2, can be incorporated into RCCL. Finally, it can be seen that the stop(0) commands following all move statements (shown in the translation table, Table 6, have been mostly omitted from the example translation. This is because the robot is not required to come to a complete stop unless an action needs to be performed at the point. Omitting these stops will allow the robot to move through the specified points without stopping, rather than stopping at each intermediate point.

9.6 Discussion

The example shows that given the detailed type of RTM input, it is relatively easy to translate RTM input into RCCL commands. The input shown represents points by their world coordinates, and no elements were included in the RTM input which could not be modeled with RCCL.

A second type of RTM input, for which the user only supplies distance moved and does not specify world coordinates, cannot be effectively translated. The translating program, upon receiving this type of input, could conceivably prompt the user for the direction he/she wished the robot to move (+X, -X, +Y,...). The resultant RCCL program would maneuver the robot direction-by-direction in order to reach the specified point. For example a move of 10 inches in the x direction, -15 inches in the y, and 3 in the z would require three separate move segments rather than a single move to the destination

Control Operator	Description
REP(eat)	Repeat a set of primatives a given number of times.
PAR(allel)	Perform a set of primatives simultaneously
IF (condition) GOTO	Branch to various points in the task based on status of condition.

Table 7 RTM Control Operators

```
t3 1
simulation of loading and unloading a lathe
40,13,-23.15,180,0,0
1 orl                                           go to bar stock
10,60,35,-29,90,35,0
2 re                                            open fingers of tool #2
3 rl                                            lower fingers over bar stock
5,65,38,-29,90,35,0
4 grl                                           grasp the bar stock
5 rl                                            raise in two stages
5,65,37,-20,90,35,0
6 orl                                           move while rotating
10,38,22,-20,90,35,-90
7 ml                                            move to front of lathe
10,69,23,-26,90,35,-90
8 re                                            open fingers of tool #1
9 d 42.78                                       wait for lathe cover to open
10 ml                                           enter: bring fingers over part
2,74,26,-26,90,35,-90
11 grl                                          grasp the processed part
12 t 42.78                                      wait for chuck to open
13 ml                                           raise part
5,74,26,-20,90,35,-90
14 orl                                          rotate gripper 180 degrees
10,74,36,-20,90,35,90
15 ml                                           lower arm
2,34,26,-26,90,35,90
16 t 13.7                                       wait for chuck to close
17 re                                           release bar stock
18 ml                                           move out of lathe
5,69,23,-26,90,35,90
19 grl                                          close fingers of tool #2
20 ml                                           move to intermediate point
10,72,10,-26,9,035,90
21 orl                                          rotate gripper while moving
10,83,12,-26,90,35,0
22 ml                                           move to part disposal
2,83,12,-29,90,35,0
23 d 21.94                                      wait for release signal
24 re                                           release finished part
25 rl                                           raise up
5,79,9,-29,90,35,0
26 ml                                           return to start position
10,40,13,-23.15,180,0,0
end
```

Figure 12: Sample RTM Task

```
t3task()
{
        trsf_ptr z, e, b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14;
        pos_ptr p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14;

        e=gentr_trsl("e", 0. , 0. , 10. );
        z=gentr_trsl("z", 0. , 0. , 800. );

        b0=gentr_rpy("b0", 40. , 13. , -23. 15, 180. , 0. , 0. );
        b1=gentr_rpy("b1", 60. , 35. , -29. , 90. , 35. , 0. );
        b2=gentr_trsl("b2", 65. , 38. , -29. );
        b3=gentr_trsl("b3", 65. , 37. , -20. );
        b4=gentr_rpy("b4", 38. , 22. , -20. , 90. , 35. , -90. );
        b5=gentr_trls("b5", 69. , 23. , -26. );
        b6=gentr_trsl("b6", 74. , 26. , -26. );
        b7=gentr_trsl("b7", 74. , 26. , -20. );
        b8=gentr_rpy("b8", 74. , 36. , -20. , 90. , 35. , 90. );
        b9=gentr_trsl("b9", 34. , 26. , -26. );
        b10=gentr_trsl("b10", 69. , 23. , -26. );
        b11=gentr_trsl("b11", 72. , 10. , -26. );
        b12=gentr_rpy("b12", 83. , 12. , -26. , 90. , 35. , 0. );
        b13=gentr_trsl("b13", 83. , 12. , -29. );
        b14=gentr_trsl("b14", 79. , 9. , -29. );

        p0=makeposition("p0", z, t6, e, EQ, b0, TL, e);
        p1=makeposition("p1", z, t6, e, EQ, b1, TL, e);
        p2=makeposition("p2", z, t6, e, EQ, b2, TL, e);
        p3=makeposition("p3", z, t6, e, EQ, b3, TL, e);
        p4=makeposition("p4", z, t6, e, EQ, b4, TL, e);
        p5=makeposition("p5", z, t6, e, EQ, b5, TL, e);
        p6=makeposition("p6", z, t6, e, EQ, b6, TL, e);
        p7=makeposition("p7", z, t6, e, EQ, b7, TL, e);
        p8=makeposition("p8", z, t6, e, EQ, b8, TL, e);
        p9=makeposition("p9", z, t6, e, EQ, b9, TL, e);
        p10=makeposition("p10", z, t6, e, EQ, b10, TL, e);
        p11=makeposition("p11", z, t6, e, EQ, b11, TL, e);
        p12=makeposition("p12", z, t6, e, EQ, b12, TL, e);
        p13=makeposition("p13", z, t6, e, EQ, b13, TL, e);
        p14=makeposition("p14", z, t6, e, EQ, b14, TL, e);
```

Figure 13: RCCL translation of sample RTM task

```
    setmod(c);                      /* set cartesian mode */
    setvel(10,10);                  /* set velocity */
    move(p0);                       /* move to initial position */
    move(p1);                       /* go to bar stuck */
    waitfor(p1->end);
    OPEN2;                          /* open fingers of tool #2 */
    setvel(5,5);
    move(p2);                       /* lower fingers over bar stock */
    waitfor(p2->end);
    CLOSE2;                         /* grasp the bar stock */
    move(p3);                       /* raise bar stock */
    setvel(10,10);
    move(p4);                       /* move while rotating */
    move(p5);                       /* move to front of lathe */
    waitfor(p5->end);
    OPEN1;                          /* open fingrs of tool #1 */
    stop(1530.);                    /* wait for lathe cover to open */
    setvel(2,2);
    move(p6);                       /* enter: bring fingers over part */
    waitfor(p6->end);
    CLOSE1;                         /* grasp the processed part */
    stop(1530.);                    /* wait for chuck to open */
    setvel(5,5);
    move(p7);                       /* raise part */
    setvel(10,10);
    move(p8);                       /* rotate gripper 180 degrees */
    setvel(2,2);
    move(p9);                       /* lower arm */
    stop(500.);                     /* wait for chuck to close */
    OPEN2;                          /* release bar stock */
    setvel(5,5);
    move(p10);                      /* move out of lathe */
    CLOSE2,                         /* close fingers of tool #2 */
    move(p11);                      /* move to intermediate point */
    move(p12);                      /* rotate gripper while moving */
    setvel(2,2);
    move(p13);                      /* move to part disposal */
    stop(780.);                     /* wait for release signal */
    OPEN1;                          /* release finished part */
    setvel(5,5);
    move(p14);                      /* raise up */
    setvel(10,10);
    move(p0);                       /* return to start position */
}
```

Figure 13 (cont.)

RTM statments	RCCL equivalent

REP (serno1) TO (serno2) N Times	for (i=0 ; i < N-1 ; ++i) {
	        RCCL equivalents of
	        RTM primatives numbered
	        serno1 through serno2
	}

PAR (serno1) TO (serno2)	move(p) (serial number 1)
	secondary action to be performed in parellel
	waitfor(p -> end);
	
	comment: only applicable to an action perormed in parallel with a move.

IF (condition) GOTO (goto descriptor)	if (condition) {
	   statements to be executed
	}
	if (not condition) {
	   alternative statements
	}
	
	comment: problems here sorting out gotos

Table 8: RCCL Equivalents of RTM Control Commands

point. This restriction is a major drawback of a RTM to RCCL translation system, since this second type of RTM input is the easier of the two to use.

A second point to comment on is the free use of position and transform pointers in the example of Figure 13. In the pointer declaration section, all pointers were reserved as permanent parts of the memory. RCCL permits the user to temporarily assign memory to a point which is not needed throughout the entire program. This memory can be reallocated to another point later in the program. This feature can keep the memory requirements at a reasonable level and prevent the storing of unnecessary positions and transforms. For any large RTM program, the use of this feature is recommended.

9.7 Shortcomings

As mentioned before, not all RCCL capabilities can be implemented from RTM information. These include:

1. Distinguishing between joint and cartesian motions

2. Interacting with a moving conveyor

3. Complying or exerting a force in one or more directions

4. Integrating with sensors

An example of an RCCL program which could not have been generated from an RTM input deck is shown in Figure 14 along with a step by step explanation of the RCCL input. The example and the explanation are both taken from [13]. Such input could not be requested by an RTM input translation because the RCCL input 1) requires both cartesian and jointed motions, and 2) requires the robot to interface with a moving conveyor.

9.8 Conclusion

Through the examples presented, one can see that while RTM input can be translated into a runable RCCL program, the use of such a translation program will severely limit the user's RCCL capabilities.

10. COMBINING RTM AND SINDECS-R TO MODEL ROBOT WORK CELLS

10.1 Introduction

We have found that RTM and a robotic work cell simulator called SINDECS-R combine well to accurately model robotic work cells. This section briefly describes SINDECS-R, and points out how RTM can be used to generate some of SINDECS-R's input parameters. An example of combining RTM and SINDECS-R is then shown. Finally, some research

1) The first example defines two locations that differ by position and orientation. The two positions are described with respect to a moving frame in *world* coordinates. A loop causes a motion back and forth from one position to the other. The final motion translates along the Y axis.

```
 1  #include "rccl.h"
 2
 3  pumatask()
 4  {
 5          TRSF_PTR z, e , b, pa1, pa2, conv;
 6          POS_PTR  p0, pt1, pt2;
 7          int convfn();
 8          int i;
 9
10          conv = newtrans("CONV",convfn);
11          z = gentr_trsl("Z",  0.,  0., 864.);
12          e = gentr_trsl("E" , 0. , 0. , 170.);
13          b = gentr_rot("B", 600. , -500., 600., yunit, 180.);
14          pa1 = gentr_eul("PA1"  , 30., 0., 50., 0., 20., 0.);
15          pa2 = gentr_eul("PA2"  , -30., 0., 50., 0., -20., 0.);
16
17          p0 = makeposition("P0" , z, t6, e, EQ, b, TL, e);
18          pt1 = makeposition("PT1", z, t6, e, EQ, conv, b, pa1, TL, e);
19          pt2 = makeposition("PT2", z, t6, e, EQ, conv, b, pa2, TL, e);
20
21          setvel(300, 50);
22          setmod('c');
23          setime(300, 0);
24          move(p0);
25          for (i = 0; i < 4; ++i) {
26                  movecart(pt1, 100, 1000);
27                  movecart(pt2, 100, 1000);
28          }
29          setmod('j');
30          move(park);
31  }
32
33  convfn(t)
34  TRSF_PTR t;
35  {
36          t->p.y += 3.;
37  }
```

Line 1 includes the necessary RCCL declarations. Line 3 deserves a comment : when using the puma manipulator, the RCCL library calls the function 'pumatask' as the task to be executed. Before calling the 'pumatask' function, the system perform some initializations. When the function returns, as you might expect, the system performs a 'waitfor(completed)' before concluding and exiting. Line 5 and 6, allocates transform and position pointers as needed by the task. Line 7 declares the name 'convfn' as a pointer to a function that describes the moving coordinate frame, and line 8 allocates a counter variable. Line 10, allocates a functionally defined transform attached to 'convfn'. Lines 11 through 15, allocate and initialize transforms as described earlier. The *Z* transform sets a frame at the base of the manipulator. The *E* and *B* transforms are the tool transform and a location with respect to the simulated conveyor. Note that the *B* transform contains a 180 degree rotation around the Y axis such as the Z direction of frame described by *B* points downward (relatively to *CONV* and *Z*). The transforms *PA*1 and *PA*2 define two locations with respect to the frame described by *B*.

Figure 14

Lines 17, 18, and 19 set up the position equations as described earlier.

Line 21 sets the velocity to 300 millimeters per seconds and 50 degrees per second and the motion mode is set to *Cartesian* mode on line 22. The call to setime on line 23, containing a null segment time, and specifies a 3/10 of a second acceleration time when reaching P0 to allow for a sufficiently long transition time because the next motion occurs with respect to a moving frame (the system has no means to now how fast it is going to move). The 'for' loop, lines 25 to 28, causes eight move requests to be entered in the queue. The eight motions are performed in 1 second each with a 1/10 of a second transition time as specified by the macro movecart. Line 29 sets the mode to *joint* because the arm is to perform a large motion and the path the *tool* frame is going to follow is of no concern. Line 30 is the last motion request to the 'park' position.

The function 'convfn', lines 33 to 37, starts being evaluated when the first motion to "PT1" begins and during the seven subsequent motions. The background function attached to the transform is called by the system with one argument pointer, a pointer to the transform it is attached to. This permits us to write functions independently from the actual transform they are attached to. Since *newtrans* created the transform *CONV* as a unit transform, the value of the p_z element of the position vector increases from 0 to approximately 280 millimeters (it is increased by 1 millimeters each 28 milliseconds [illegible] *CONV* [illegible] equal to the unity transform. The first time the manipulator moves to PT1, the motion is the result of a combination of the *Cartesian* motion from P0 toward PT1 and the motion due to the moving coordinate frame.

This example introduce the first method for generating functionally defined motion by a periodic increment of a static variable (here a transform element).

Figure 14 (cont.)

projects in which these two systems have been applied are discussed.

10.2 SINDECS-R

SINDECS-R simulates a number of robots working in a cell. Stations of the work cell can model NC machines, assembly, or other operations. When tending NC machines the robot(s) act only as materials handling devices, delivering parts and loading machines when necessary. When tending assembly or other operations, the robot(s) may be required to be present throughout some or all of a part's processing. The program simulates the operation of the defined cell under user selected flow control strategies. It generates performance data on the cell which includes production rates, machine utilizations, and robot utilizations.

Input to the simulator consists of the number of machines or stations, definition of part types and their processes, definition of robot motion times between stations, definition of the times the robot takes to unload and reload machines, and the user's choice of rules to be applied to solve the flow control decisions that have to be made during the cell's operation. Further details about SINDECS-R can be found in [14, 15].

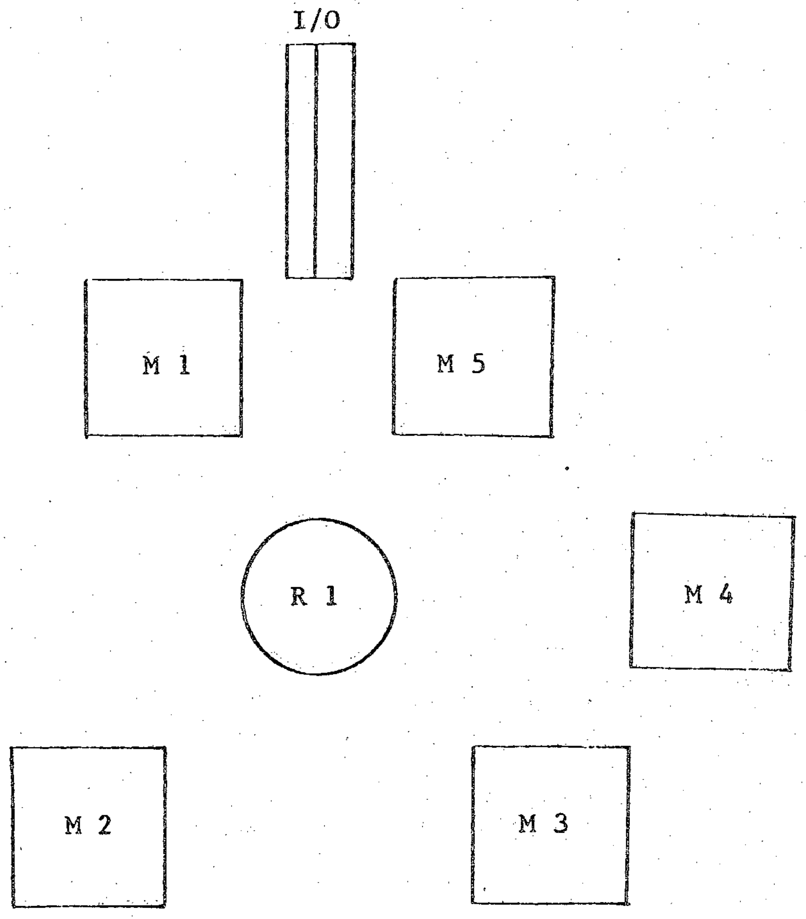
10.3 RTM and SINDECS-R

RTM can be used to generate the robot motion times that are required by SINDECS-R's input. After spatially defining the locations of stations in the cell, RTM can generate the motion times the robot would require to move parts the distances described in the the spatial arrangement. RTM can also be used to determine the times the robot should take to unload and reload the stations. This requires the user to determine the motions necessary to perform the unload and/or reload at a station, and generate the times to perform these motions using RTM.

10.4 Example

Suppose a work station has the spatial layout shown in Figure 15. A single robot tends the stations, and acts as a materials handling device transporting parts throughout the cell. The RTM output of the analysis of motion times within the cell is shown in Figure 16. The comment of each primitive's output denotes which stations this motion time is to be applied to. It should be pointed out here that station number zero denotes the input and output station of the cell. It is assumed that parts enter and leave the system at this station.

Figure 17 shows the output of the RTM analysis to determine the times for individual unload and reload of the machines. In SINDECS-R, unload time is the time the robot requires to remove a completed part from a machine. This time is used when no parts are waiting in the machine's queue. Reload time is the time required to remove a completed part from a machine and load the machine with an unprocessed part from its queue.



R = Robot
M = Machine
I/O = Input/Output

Figure 15

the tomorrow tool

move times between stations

control card	serial no.	rtm symbol	motion length [in]	motion velocity [ips]	operation time [tmu]	user comment	comment
	1	m1	24.00	30.00	32.4	0 - 1	
	2	m1	84.00	30.00	87.9	0 - 2	
	3	m1	96.00	30.00	99.0	0 - 3	
	4	m1	72.00	30.00	76.8	0 - 4	
	5	m1	24.00	30.00	32.4	0 - 5	
	6	m1	24.00	30.00	32.4	1 - 0	
	7	m1	72.00	30.00	76.8	1 - 2	
	8	m1	84.00	30.00	87.9	1 - 3	
	9	m1	72.00	30.00	76.8	1 - 4	
	10	m1	36.00	30.00	43.5	1 - 5	
	11	m1	84.00	30.00	87.9	2 - 0	
	12	m1	72.00	30.00	76.8	2 - 1	
	13	m1	48.00	30.00	54.6	2 - 3	
	14	m1	72.00	30.00	76.8	2 - 4	
	15	m1	84.00	30.00	87.9	2 - 5	
	16	m1	96.00	30.00	99.0	3 - 0	
	17	m1	84.00	30.00	87.9	3 - 1	
	18	m1	48.00	30.00	54.6	3 - 2	

Figure 16

```
control   serial     rtm     motion    motion    operation   user comment                                  comment
card       no      symbol   length   velocity     time
                             [in]     [ips]       [tmu]
--------------------------------------------------------------------------------------------------------------------

           19        m1      48.00    30.00       54.6      3 - 4

           20        m1      84.00    30.00       87.9      3 - 5

           21        m1      72.00    30.00       76.8      4 - 0

           22        m1      72.00    30.00       76.8      4 - 1

           23        m1      72.00    30.00       76.8      4 - 2

           24        m1      48.00    30.00       54.6      4 - 3

           25        m1      60.00    30.00       65.7      4 - 5

           26        m1      25.00    30.00       33.3      5 - 0

           27        m1      36.00    30.00       43.5      5 - 1

           28        m1      84.00    30.00       87.9      5 - 2

           29        m1      84.00    30.00       87.9      5 - 3

           30        m1      60.00    30.00       65.7      5 - 4


  the total time for the task is          2082.9 [tmu] =     75.0 [sec] =      1.2 [min]
 ------------------------------------------------------------------------------------------
```

From-to move times - rtm output

Figure 16 (cont.)

```
t3 2
unload time - single or double gripper
1 r2 a,40,24.0,30.2,4.0

2 gr1                                           reach into machine
3 d 4.0                                         grasp part
4 m1 2,2.0                                      wait for chuck to open

5 m1 40,28.0                                    pull part off chuck

end                                             move out of machine
```

Unload time - rtm input

```
                                                the  tomorrow  tool
                                                -------------------

                                            unload time - double gripper


control  serial   rtm    motion   motion   operation   user comment                              comment
card      no.    symbol  length   velocity   time
                          [in]     [ips]     [tmu]
------------------------------------------------------------------------------------------------------------

            1      r2     28.00    40.00      16.4    reach into machine

            2      gr1                          .3    grasp part

            3      d                           4.0    wait for chuck to open

            4      m1      2.00     2.00      37.9    remove part from chuck

            5      m1     28.00    40.00      29.6    move out of machine


 the total time for the task is              88.2 [tmu] =     3.2 [sec] =       .1 [min]
---------------------------------------------------------------------------------------
```

Figure 17 Unload time - rtm output

```
no. of machine types=  5


     part type  1

          process #  1

            p( scrap )=         0
            p( rework)=     .10000
            p( good  )=     .90000

          operation      machine      process time      p(returning to this
                                                         station for rework)

              1             4.            30.00              1.00


          process #  2

            p( scrap )=         0
            p( rework)=         0
            p( good  )=    1.00000

          operation      machine      process time      p(returning to this
                                                         station for rework)

              1             3.           120.00               0
              2             1.            75.00               0


          process #  3

            p( scrap )=     .20000
            p( rework)=     .10000
            p( good  )=     .70000

          operation      machine      process time      p(returning to this
                                                         station for rework)

              1             1.            40.00              .80
              2             5.            80.00              .10
              3             2.           110.00              .10
```

Figure 18

```
part type  2

     process #  1

       p( scrap )=         0
       p( rework)=         0
       p( good  )=   1.00000

     operation      machine      process time      p(returning to this
                                                   station for rework)

         1            4.              60.00              0
         2            3.              90.00              0


part type  3

     process #  1

       p( scrap )=         0
       p( rework)=         0
       p( good  )=   1.00000

     operation      machine      process time      p(returning to this
                                                   station for rework)

         1            2.             100.00              0


robot    1

     0   1.17   3.16   3.56   2.76   1.17      0
  1.17      0   2.76   3.16   2.76   1.57      0
  3.16   2.76      0   1.96   2.76   3.16      0
  3.56   3.16   1.96      0   1.96   3.16      0
  2.76   2.76   2.76   1.96      0   2.36      0
  1.17   1.56   3.16   3.16   2.36      0      0
     0      0      0      0      0      0      0

 station        unload       reload

  0              3.20        14.20
  1              3.20        14.20
  2              3.20        14.20
  3              3.20        14.20
  4              3.20        14.20
  5              3.20        14.20
```

Figure 18 continued

```
results of sindecs-r analysis

          production rate for this system =     1.200 pieces per hour

          standard deviation =     2.871


          production rates by part type

                            prod rate           std dev
                part 1          .480              1.909
                part 2          .400              1.752
                part 3          .320              1.576


          average time in system =  451.079 minutes

          standard deviation =  204.488


          average time in system by part type

                             ave time           std dev
                part 1        445.862           269.164
                part 2        519.488           204.192
                part 3        373.392            53.817


       quality rates

                 rate            std dev

       scrap      .080              .800
       rework    0                 0
       good     1.200             2.871
```

Figure 18 continued

server utilization measures

tape run stats

machine type	server utilization	standard deviation
1	.300	.458
2	.856	.351
3	.853	.354
4	.684	.465
5	.066	.248

machine occupation stats

machine type	server utilization	standard deviation
1	.352	.478
2	1.000	0
3	1.000	0
4	.925	.264
5	.081	.273

robot utilzation statistics

serial number	robot utilization	standard deviation
1	.725	.446

Figure 18 continued

These robot motion times are supplied by a user to SINDECS-R. The SINDECS-R input is not shown here. The output of the simulator is shown in Figure 18. The three part types represent the three distinct types of parts which were produced in this robotic cell.

Note here that any fictitious robot motion times could have been supplied to SINDECS-R. The use of RTM to estimate these times merely makes the simulation more realistic in its depiction of cell performance.

10.5 Research Combining RTM and SINDECS-R

The combination of RTM and SINDECS-R has been used for a number of research topics since the development of SINDECS-R in 1983. One application was in a comparative analysis of a robot in a cell using a double gripper to one using a single gripper. The motion times of parts between stations was the same in either case, but the RTM-generated reload times distinguished the two cases. The robot with the double gripper could simultaneously manipulate the part being unloaded and the fresh part being loaded into the machine. The robot with the single gripper could only manipulate one part at a time [15].

A similar study was performed comparing a robotic cell applying vision to one with touch sensing. RTM was used in this case to determine the inter-station motion times [15].

RTM and SINDECS-R were also used to examine the performance of an operating system whose purpose is to coordinate multiple robots in a cell. RTM was used to generate the motion times between stations. A modified version of SINDECS-R was used to generate performance information of a cell with dynamic avoidance capability collision to one with no possibility of collision.

Another investigation has combined RTM and SINDECS-R to model a single machine station with tool changing capabilities. The cell was modeled with the robot changing tools, using times generated by RTM, to automatic tool changing by a dedicated tool changing mechanism.

References

1. Paul, R.P. and Nof, S.Y., "Human and Robot Task Performance", presentation at the International Symp. on Computer Vision and Sensor Based Robots, Warren, Michigan, November 1978. Appeared as a book chapter in Computer Vision and Sensor Based Robots, G.G. Dodd and R. Lothar (Eds.), Plenum Press, New York, 1979. Also published in a revised version under the title "Work Methods Measurement - A Comparison Between Robot and Human Task Performance", Int. J. of Production Research, Vol. 17, No. 3, 1979, pp. 277-303.

2. Nof, S.Y. and Paul, R.P., "A Method for Advanced Planning of Assembly by Robots", Proc. of Autofact West, Anaheim, California, November 1980.

3. Lechtman, H., "Robot Performance Models Based on the R.T.M. Method", unpublished M.S. Thesis, School of Industrial Engineering, Purdue University, West Lafayette, Indiana, May 1981.

4. Nof, S.Y. and Lechtman, H., "Robot Time and Motion", Industrial Engineering, April 1982, pp. 38-48.

5. Nof, S.Y. and Lechtman, H., "Now It's Time for Rate Fixing for Robots", The Industrial Robot, June 1982, pp. 106-116.

6. Lechtman, H. and Nof, S.Y., "Performance Time Models for Robot Point Operations", Int. J. of Production Research, Vol. 21, No. 3, 1983.

7. Nof, S.Y., "Robot Ergonomics: Optimizing Robot Work", chapter in the Handbook of Industrial Robotics, S.Y. Nof, Editor, John Wiley and Sons, New York, 1985.

8. Hershey, R.L., Leztz, A.M. and Nof, S.Y., "Computer Methods for Predicting Robot Performance", Proc. of Autofact 5, Detroit, Michigan, November 1983, pp. 3.9-16.

9. Hershey, R.L., Leztz, A.M. and Nof, S.Y., "Predicting Robot Performance with ROFAC, A Decision Making Aid", Proc. of IIE Conf., Toronto, Canada, November 1983.

10. Paul, R.P., "Robot Manipulators: Mathematics, Programming, and Control", MIT Press, 1981.

11. Hayward, V., "Introduction to RCCL: A Robot Control 'c' Library", TR-EE83-43, School of Electrical Engineering, Purdue University, West Lafayette, Indiana, October 1983.

12. Hayward, V., "Robot Real Time Control User's Manual", TR-EE83-42, October 1983.

13. Hayward, V., "RCCL User's Manual", TR-EE83- , October 1983.

14. Robinson, A.P. and Nof, S.Y., "SINDECS-R: A Simulator for Robotic Cell Activities", Proc. Winter Simulation Conference, Arlington, Virginia, December 1983, pp. 350-355.

15. Robinson, A.P., "Principles for Robot Work Design", Unpublished M.S. Thesis, School of Industrial Engineering, Purdue University, West Lafayette, Indiana, August 1984.