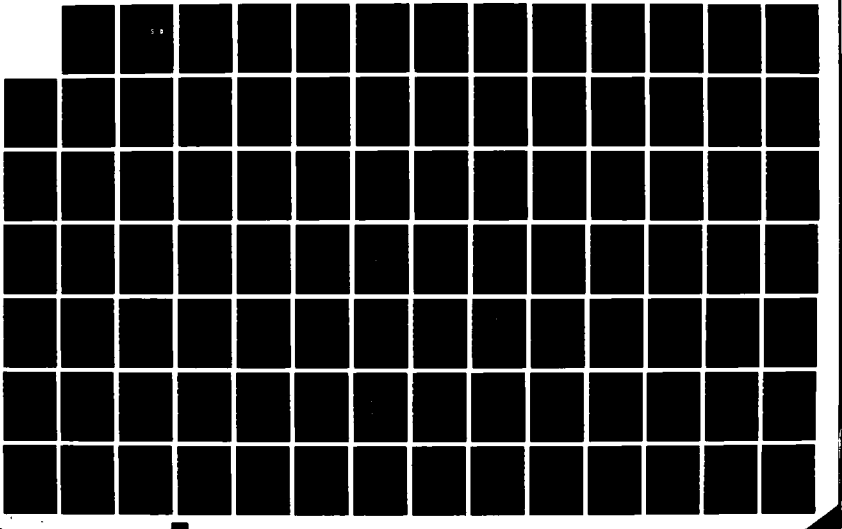
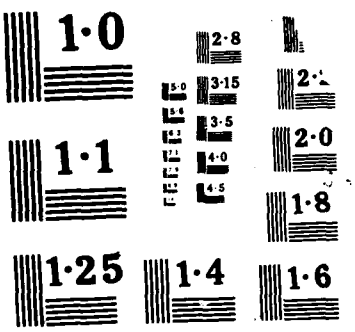
AD-A190 869 THE COORDINATION OF MULTIPLE ROBOTIC MANIPULATORS(U) 1/2
ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA R F YOUNG
11 DEC 87

UNCLASSIFIED F/G 13/8 NL




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·8
3·15
3·5
4·0
4·5

THE COORDINATION OF MULTIPLE ROBOTIC MANIPULATORS

Reed F. Young, 1LT
HQDA, MILPERCEN (DAPC-OPA-E)
200 Stovall Street
Alexandria, VA 22332

11 Dec 87

Approved for public release

A thesis submitted to Duke University in partial fulfillment of the requirements for the degree of Master of Science

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp Date Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: DUKE UNIVERSITY / 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Durham, NC 27706	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: US Army (TEP Program) / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

THE COORDINATION OF MULTIPLE ROBOTIC MANIPULATORS

12. PERSONAL AUTHOR(S)

YOUNG, Reed Fisher

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM Aug. 86 TO May 88	1987 Dec. 11	174

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In this thesis, the problem of the coordination of multiple robots is investigated at length. Included in the discussion are the position control, path planning, and collision avoidance strategies involved in the control of two robots mutually holding an object. The theories and algorithms presented in this thesis are tested and implemented in the computer simulation "CoordSim."

Major results from this thesis include a verification of the resolved position control theory as it drives the robots to any position and orientation. Also, a concatenation and adaptation of various single robot theories is made and introduced as the "Striving Technique." This solution is a complete movement algorithm which drives the two robots mutually holding an object through a field with obstacles. The algorithm also generates a path function given only the start and end positions and orientations. Finally, the concepts of the "coordinated work envelope" and "twisting collision" are derived and discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS		
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

THE COORDINATION OF MULTIPLE ROBOTIC MANIPULATORS

Reed F. Young

Department of Mechanical Engineering
and Materials Science

Duke University

Date: December 11, 1987

Approved:

Devendra P. Garg

Devendra P. Garg, Supervisor

C M Gaman

Jack L. Lehman

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of
Science in the Department of Mechanical Engineering
and Materials Science in the
Graduate School of Duke University

1988

ABSTRACT

In this thesis, the problem of the coordination of multiple robots is investigated at length. Included in the discussion are the position control, path planning, and collision avoidance strategies involved in the control of two robots mutually holding an object. The theories and algorithms presented in this thesis are tested and implemented in the computer simulation "CoordSim".

Major results from this thesis include a verification of the resolved position control theory as it drives the robots to any position and orientation. Also, a concatenation and adaptation of various single robot theories is made and introduced as the "Striving Technique". This solution is a complete movement algorithm which drives the two robots mutually holding an object through a field with obstacles. The algorithm also generates a path function given only the start and end positions and orientations. Finally, the concepts of the "coordinated work envelope" and "twisting collision" are derived and discussed.

ACKNOWLEDGEMENTS

I would foremost like to thank my advisor, Dr. Devendra P. Garg, Professor of Mechanical Engineering, for all of his effort, help, and guidance throughout this work. I would also like to thank my family for the support and encouragement they have given me.

My gratitude to the United States Army for the funding of my master's education. I can only hope to serve them as they have served me.

Finally, my thanks to Dr. C. M. Harman and Prof. Jack Rebman for their input and participation on my committee.

You have all made it possible for me.

DEDICATION

This thesis is dedicated to all of my friends who made my stay here at Duke University such an enjoyable one and especially to Carmen--you made it all that much better.

TABLE OF CONTENTS

LIST OF FIGURES

NOMENCLATURE

A	projection of the satellite's z-axis onto the reference frame
AC	Activity Controller
AP	alternate point
F()	a generalized function
J	Energy Function
N	projection of the satellite's x-axis onto the reference frame
O	projection of the satellite's y-axis onto the reference frame
P	position of the satellite's origin in the reference frame.
T	standard transformation matrix
t	time
u	system input
x	state variables
α	roll angle
β	pitch angle
Γ	yaw angle
τ	finish time

CHAPTER I

INTRODUCTION

1.1 Motivation for Study

One of the emerging areas of research in a rapid advance of robotic technologies has been concerned with the coordination of multiple robotic manipulators. This technology proves to be very useful for many reasons as the coordinated robot scheme has several capabilities which are not found in systems utilizing only a single robot. These capabilities include factors such as: increased payload rating; manipulation of long, heavy, irregularly shaped, or unbalanced objects; and allowance for an extremely dynamic work-cell.

In this thesis, a complete position control algorithm is presented in the "Striving Technique" which provides for the resolved position control, path planning, and collision avoidance of two coordinated robots simultaneously holding an object. Along with this technique, the concepts of coordinated work envelope and twisting collision are introduced and discussed.

1.2 Outline of Thesis

Chapter 2 presents a literature search into the various aspects of coordinated robot systems. Discussion

is included which describes applications, justification and benefits of multiple robot systems, some potential problems which may be encountered in the implementation of these systems, and various algorithms which are available in literature. Also, the concept of the coordinated work envelope is introduced and discussed at length.

In chapter 3, a summary of collision avoidance techniques in single robots is presented as well as means by which these algorithms can be adapted to control coordinated robots. This chapter also introduces the concept of twisting collision and methods for solving this problem.

Chapter 4 discusses the path planning of coordinated robots, and presents available algorithms. This chapter also contains a comparison between the concepts of path planning and collision avoidance and how they can be combined in a control structure for the coordinated scheme.

Contained in chapter 5 is the system modeling implemented in this thesis' work. Included are the position control, path planning, and collision avoidance algorithms which are actually implemented in the computer simulation. Also, a description of the simulated robot hardware is included.

Chapter 6 describes the computer simulation CoordSim, and how the program utilizes the algorithms presented in chapter 5. The definition of the various program procedures and routines used are also included.

The last chapter, chapter 7, discusses the results and conclusions obtained from the testing performed in this thesis. Along with this discussion, several suggestions are made for work which could extend the knowledge gained by this thesis' work.

Appendix 1 contains a detailed listing of the computer simulation. Appendix 2 is a guide for potential users of the simulation.

CHAPTER II

BACKGROUND IN COORDINATION

2.1 Introduction

This chapter will provide background information in the area of robot coordination and control. In addition, a discussion of the motivation and practicality of coordination will be given. Some benefits and setbacks that are characteristic of coordination as well as some real world applications which have utilized more than one robot will also be given. Finally, the discussion will focus on the various algorithms which have been set forth to control the desired motions.

2.2 Background

Many of the same principles apply in coordination of multiple robots, as do in the control of a single robotic manipulator. The obvious difference is that instead of having to calculate the joint angles and various paths of the single robot, one has to calculate these factors for several coordinated robots, as well as calculate other added complexities. These complexities include such things as the avoidance of collision with each other or insuring motion in unison if the robots are to share a task cooperatively.

The degree to which robots can be considered in cooperation has been divided into three general categories. These are; uncoordinated, loosely coordinated, and closely coordinated [1, 2, 3].

The uncoordinated robots scheme implies the types of systems which are most recognizable as the production or assembly line robots; for example, systems which are designed to paint or weld. These robots simply follow a pre-defined motion with no knowledge or concern about the objects which may be present in their working envelopes. In this sense the robot is completely blind to the surroundings. However, for the simple, repetitive functions which they are required to perform, this setup is more than adequate.

Another classic characteristic of uncoordinated robots is the fact that they usually have non-overlapping work envelopes. This is used as a safety precaution, since no matter how well planned a motion is, there is always the possibility of collision due to a malfunction or operator error. Since the uncoordinated robots cannot communicate with each other, separating their work envelopes is an effective way of insuring that no collision takes place.

Loose coordination is the name given to the scheme whereby the robots maintain a dynamic communication link

with a higher level intelligence device such as a computer. Here two or more robots determine their own motions via a set of commands that a coordinating computer gives to them. At this level comes the first indication of some sort of intelligent control. The coordinating computer monitors the actions of the two robots and provides decisions as to their progress. One also begins to observe overlapping work envelopes with two or more robots sharing work space in the process, and even working simultaneously on the same workpiece since in this case there is an ability to prevent collisions and determine mutual paths intelligently.

Since loose coordination is usually accomplished via another controlling computer, there is an inherent time delay caused by the extra data transmission. Thus, this type of coordination is usually limited to projects such as two robots operating in an alternating fashion performing similar functions. A typical example may include a series of fed parts where the communication is kept at a minimum. It is also beneficial if the cycle periods of the overall processes are large so as to allow enough time for those communications to occur.

The third type of coordination is called close coordination. It encompasses the field of two or more robots truly communicating with one another and simultaneously striving towards a common goal. Here one

finds the robots sharing a direct communication link which transmits information on factors such as location, force, and speed constantly among the manipulators. The controlling algorithms dynamically adapt to changes in the environment such as obstacles, or even new processes altogether. In this manner, the system acts in a completely adaptive fashion towards any conflict situation which may arise. Finding a solution to the close coordination problem is a current topic of much research interest. Most work is geared towards handling such problems as gripping an object simultaneously using multiple grippers or performing different processes on a single workpiece at the same time.

2.3 Multiple Robot Control

Along with the various levels of coordination are associated various ways in which to approach the problem of control of the robots. New methods must be developed to handle the control of several robots beyond the capacity of their individual controllers [1].

Uncoordinated robots will not need much effort in the way of advanced control beyond the capability of their own controllers. Simple task planning and some provision for appropriate timing and triggering usually suffices to provide the system with enough information to carry out

conventional tasks such as welding or painting. However, these types of systems are limited to relatively simply tasks and usually do not have the capability for complex movements and calculations.

In one method of control, Chimes [18] suggests that a single controller may be used to drive the various joints of all of the robots involved. For example, for two six degree-of-freedom manipulators, a coordinated controller would provide signals for all twelve joint motors. Several problems arise in this connection. First, the same computing device must now perform over twice as many calculations since there are now twice as many joint angles and motor signals to solve for. Thus, there is a need for twice as much time to perform these calculations and this added time may degrade the overall performance of the system. Secondly, due to the limitations of the joint angles, there is no concern with a collision occurring since it is a physical impossibility for a robot to collide with itself unless it exceeds its own constraints. With another robot added, a new problem arises since assuming that the two robots share a common workspace, now they can interfere with each other's motions and collide. Thus, the controller must be fitted with an automatic collision avoidance mechanism which prevents the two manipulators from coming in contact with each other. This extra

computation also takes additional time and again slows down the overall speeds of the two manipulators.

A possible solution to the problem mentioned above is to employ parallel processing techniques where an evaluation of the individual robot's joint angles is accomplished at the same time by different electrical devices. Thus, there are two computation machines working on the same amount of data that was being manipulated by only one computer. This appears to be an easy solution but the same problems of communication remain.

A second method of control is to have the robots control themselves through their individual controllers while simultaneously communicating with a coordinating device such as a computer, in a hierarchical scheme [19]. From the definitions set forth, this method appears to be analogous to the definition of the loosely coordinated system presented earlier. The one major problem here is involved again with the speed of the process. The process can soon become a lengthy one if for every motion a controller must first communicate with a central coordinator and obtain permission to move as well as inform the controller of its actions at each step. Thus, this scheme only lends itself to situations where most of the processes can be achieved without much knowledge of the activities of the surroundings. The robot's own controller

is able to make decisions with only limited communication with the coordinator for factors such as parts location, availability of shared tools, or breakdowns. As a consequence of limiting communication, very little time is spent in the actual communication between the devices and the loose coordination can be achieved with little increase in cycle time.

A process which is well-suited to hierarchical control is event sequencing [3]. Event sequencing is commonly found on routine assembly line operations where several robots are arranged to sequentially work an a belt-fed part. In this scheme, each of the robot's controllers is connected to a central controller which monitors the location of the parts and their progress. The only time that a controller needs to communicate with the coordinator is when it needs permission to start a process and to indicate when it is finished with its present piece. In an assembly operation, this structure is very attractive for inventory control as well as for diagnostic intelligence since each piece is closely monitored throughout the process and system characteristics such as bottle-necks or throughput can easily be accessed by the coordinating computer without disturbing the operation of the individual manipulators. The individual robots also have quite a bit of freedom to perform their tasks as they can work with no

concern for their surroundings until a new part is to be presented or the completed part is to be moved on to the next station.

The third method of coordination consists of the robot's individual controllers being able to talk directly with one another. This mode requires the application of true coordination where the motion of one robot is constantly a function of the progress of its mate. Many of the excessive time related problems are now alleviated since there is no intermediary computer to slow down the processes. However, this method requires algorithms which are well-designed and very adaptable since each machine not only has to be concerned with its own path generation and progress, but it has to be conscious about its partner's progress at all times. The algorithms must be very adaptable to changes to the point of simulated intelligence and must be able to handle the extremely dynamic situations which are presented.

2.4 Justification for Coordinated Robots

In order for tomorrow's factory to be able to keep up with competition from abroad and the consumers' high demands, it will have to be able to find ways to significantly reduce costs and operation times while still improving the product's overall quality. One of the ways

to solve these problems is to utilize the technology of coordination of robots.

There are many financial benefits of the multiple robotic systems. For almost any robotic process, only 20% to 30% of the setup cost is in the actual robots themselves [4]. The remaining 70% to 80% is tied up in peripheral devices such as conveyor belts, tool holders, and vices. This means that whenever a new operation is implemented, quite a bit of the start up cost must be incurred in the procurement of peripheral devices. While the robots themselves can be adapted to most any operation, the other devices bought or built to support the robots are usually very specialized for the job and cannot be easily adapted to new functions. With two robots however, much of this cost can be eliminated since the functions of the peripheral devices are now accomplished by the utilization of the multiple robots. In general, the robots are easily adapted to new functions and can be set up to perform a vast variety of holding or feeding functions. Therefore, when new processes are implemented, only a program change needs to be made with much less capital investment in extraneous hardware [23].

There are many other benefits of the multiple robot system as suggested by Grossman et al [21]. These systems can perform tasks which were perhaps impossible for only

one robot such as lifting a very heavy, irregular, or oblong object. Many of the resources which were used by only one machine can now be shared by several. These shared devices include items such as floor-space or working envelopes, material handling equipment, tools, and parts feeders. Several of the process characteristics are also changed. The cycle time of an operation can be reduced since several robots work on a single work-piece instead of just one. This increase in cycle time could also mean that bottleneck processes in an assembly line could be overcome [25].

The sharing of tasks also increases the reliability of the overall system also. For example, if one of the robots were to break down, the other could continue the process perhaps at a slower speed, but the process would not stop. In addition, choosing robots which have different specialties and having them utilize each other's benefits increases the potential for work of the overall system. For example, one robot could be a heavy-duty machine to be used as a vice while its mate might be a precision machine to be used in a drilling process. Thus, a very versatile programmable fixture is developed which greatly increases the variety of tasks that the work-cell can accomplish.

2.5 Difficulties Encountered in Coordinated Robots

Along with the many benefits of the multiple robotic systems are also some disadvantages. However, these are few and are usually outweighed by the benefits.

Since there are now several robots to be controlled, there will be a problem in the speed of execution. Many calculations must be performed in order to control the movements of a single robot. This number will more than double with the addition of another robot since now such items as mutual collision avoidance as well as communication must be dealt with. This problem has been partially overcome through the use of parallel processing and a hierarchical structure. However, with an increase in the amount of hardware and software needed to drive a coordinated system also comes a possible increase in cost and system downtime.

One of the more complex problems presents itself through the mechanical characteristics of the system. Since the work-cell in a multiple robot environment consists of several devices, the accuracy to which a robot can define it's own location becomes a very important consideration. This accuracy is usually defined by factors such as motor resolution and repeatability as well as a load, possibly causing a mechanical deflection in the robot

arm itself. These deviations from a desired position affect the job performance since under these conditions new problems arise such as a stress being applied to a mutually held part. Since there will be a deviation from true desired coordinates, a deflection or bending may occur in a part leading to the extreme case of damaging the piece. These problems can be solved through the use of such complex systems as a wrist force sensor but there is a definite trade-off between accuracy and cost.

There will also be a need for more complex path-planning and collision avoidance algorithms. Instead of a single robot avoiding a stationary object or even an object in a pre-defined motion, the several robots must avoid each other dynamically as well as any obstacles in their workspaces. All of these new calculations as well as communication steps add to the size and complexity of the controlling algorithm, and thus, affect the motion speeds and reliability.

2.6 Work Envelopes of Coordinated Robots

The calculation of the work envelope of the coordinated robot scheme is a non-trivial problem since it plays an important role in deciding the extent of all motions that can occur in the system. Since the robots are usually mounted at different locations, the work envelope

of the coordinated robot scheme will be considerably less than the sum of the two robot's work envelopes since one robot will not be able to reach the far side of the other's work envelope and vice-versa.

The object's work envelope is defined as the locus of points which can be obtained by the coordinate system located on the object (refer to figure 2.1). It is first assumed that the robots are fully extended in order to achieve the most distant set of points in the work envelope. This configuration is called the "extended arm" configuration. The distance between the base and the end point of the extended arm is a constant called the "extended radius". Note that the individual work envelopes can be separate or overlapping with no consequence to this discussion.

The two dimensional work envelope of the x-z plane is bounded by four curves (refer to figure 2.1). Curve a is a circular arc whose radius is the sum of the extended radius of robot 2 and one-half the length of the object. Curve b is also a circular arc whose radius is the sum of robot 1's extended radius and one-half the length of the object. The centers of these arcs are located at the particular robots base coordinate system. Curves c and d are irregular arcs and are traced under the midpoint of the object as the object end points follow around the work boundary of the

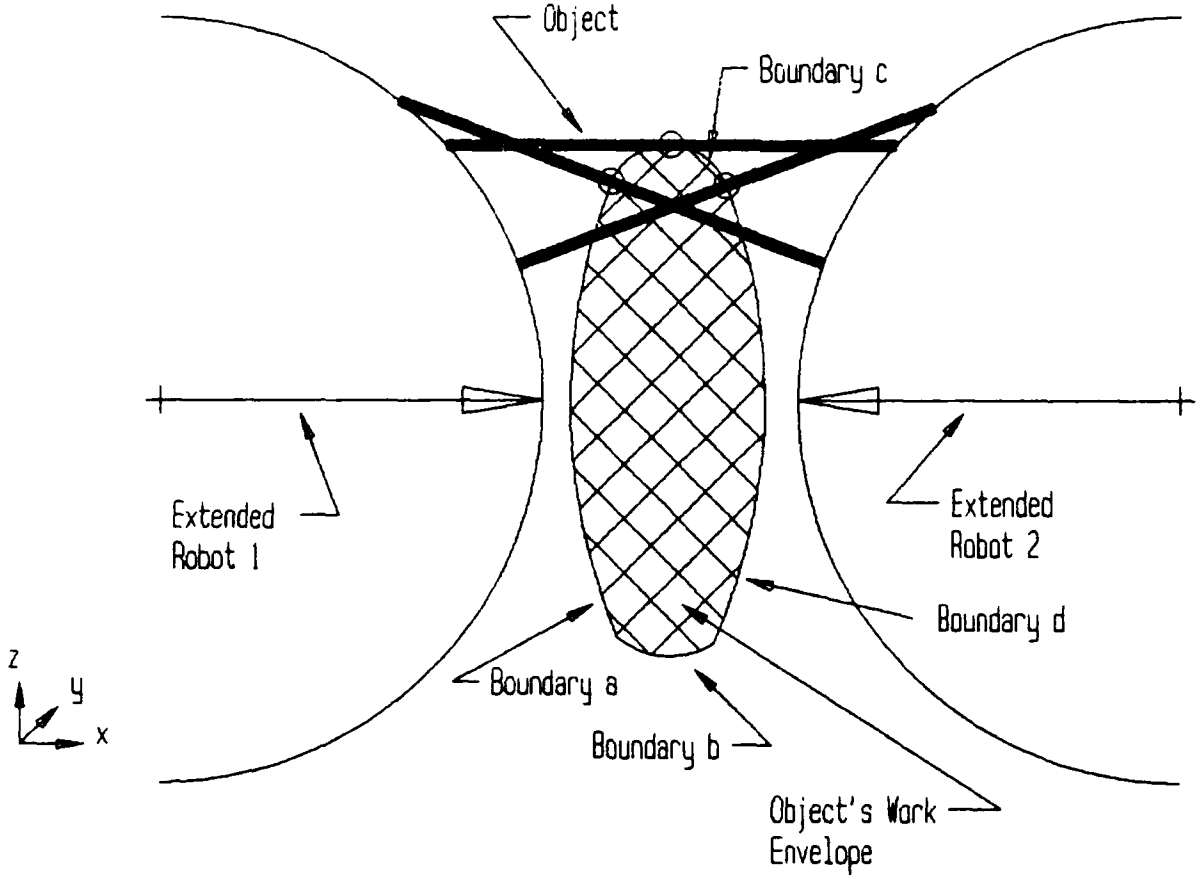


Figure 2.1
Coordinated Work Envelope

individual robots. Since this exact same shape is produced as the work boundary in the x-y plane, the actual work envelope can be generated by spinning the x-z work boundary around a line oriented parallel to the x-axis and traveling through the geometric center of the shape.

A sample calculation is now presented to illustrate the relative size of the work envelopes. Assume that the extended radius of both robots is one meter, the robot bases are mounted two and one-half meters apart, and that the held object is one meter long. If the robots' individual work-envelopes are assumed to be half-spheres the total volume of the sum of the work envelopes is:

$$\text{Volume} = {}^{4}/_{3}\ \pi\ r^{3} \qquad (2.1)$$

where "r" is the extended radius of the robots. If the object's work envelope is estimated as the upper half of an ellipse-like shape rotated about its long axis, the volume would approximately be:

$$\text{Volume} = {}^{2}/_{3}\ \pi\ x_1\ x_2{}^{2} \qquad (2.2)$$

where "x_1" is the long radius, and "x_2" is the short radius of the ellipse-like shape. For this example, "r" equals one meter; therefore, the volume of the robot's combined work envelopes is ${}^{4}/_{3}\ \pi\ m^{2}$. Also, through measurement of a scaled diagram, "x_1" equals ${}^{3}/_{4}$ meters and "x_2" equals ${}^{2}/_{3}$

meters producing a volume of $^{2}/_{9}\ \Pi\ m^{2}$. This is an estimation, but does emphasize the vast difference in the volume of the work envelopes.

As the robots are mounted farther apart, the object's work envelope decreases proportionately and it would seem that mounting the robots close together would be optimal situation. However, as the robots are mounted closer and closer together, a cluttering of obstacles will occur due to the robot joints and the bases. Thus, an optimal solution can only be found when specific tasks, and collision avoidance and path planning restrictions are considered and the trade-offs between the object's work area and the cluttering are weighed.

2.7 Applications

Several specialized uses of coordinated robots have been mentioned in literature to include lifting very heavy or long objects. One of the main advantages and applications for the multiple robotic system comes in its ability to be dynamically adaptable [3].

A dynamically adaptable system is a type of production system which is able to change its function with very little change in hardware. This change in function could be necessary due to a variety of situations. One situation would be an operation where the same type of function is

performed on several different types of a similar product. Examples of such systems would include a welding robot on an automotive assembly line which can perform body welding on a variety of randomly presented cars. Another situation would be where the operation changes altogether. For example, a setup in which a new product may have to be manufactured utilizing the same machinery as was used for the manufacture of the old product.

In normal circumstances both of these situations would require extensive hardware and peripheral device changes. With the implementation of a coordinated robotic system, the solution to these and many other problems can be realized easily. Since the system emulates the functions of the peripheral devices, a change in the make-up of those devices will simply mean a change in the software of the robots with very limited hardware changes, perhaps mainly in the grippers or parts feeders.

The true utility and functionality of the coordinated system is realized and can easily be identified as dynamically adaptable, since in a much shorter period of time and with much less difficulty, a new procedure with varying requirements can be conveniently carried out.

2.8 Available Algorithms and Scheme Prototypes

There are several mathematical algorithms which are discussed below in order to provide some cognition as to the controlling scheme of the coordinated robots. While these methods do not indicate a specific programming routine, they do provide a systematic procedure for a programmer to follow while setting up the various operations.

One of the most common of these algorithms is called the master-slave relationship [5]. The basic premise of this routine is to define one of the several robots as the master and then base the movements and operations of the other devices from the master's position and progression.

This method has proven to be very effective and efficient in several different situations. First, the slave can be directed to simply copy the exact motion that the master is programmed to perform. This is a useful control scheme in pick and place operations since both the master and the slave robots can be programmed to perform the same task but a time delay between the two cycles. Thus, twice as much work can be accomplished through the same program since two robots perform a task in an alternating fashion. In the same sense, this application can also solve the problem of bottle-necking as several

robots can perform similar operations simultaneously and speed up the cycle time of a particular station.

A second situation would be to have the second robot generate a symmetric motion function relative to the first robot, for example two robots turning a wheel. The master would base its position function on the location of the wheel and the desired angular speed. The slave would simply calculate its motion as the master's reflection through the middle of the wheel. This requires much less calculation than if both robots were to calculate positions relative to the wheel and thus lends itself to being an operation to occur in real time.

Another situation would be synchronous motion where the slave would perform a motion similar to the master's but at some offset distance. An example of this would be two robots carrying a long bar. The desired path of the bar would provide the master with a path to follow. The slave robot would simply follow the same direction and orientation as the master knowing that it must first place itself at a distance offset equal to the length of the bar. The relative locations of the bases will not become a hindering factor in the path generation since only the end effector position and orientation need be known, not the individual joint angles which would be a function of the bases' position.

The master-slave routines can be very effective for simple or repetitive tasks but they may fail to perform satisfactorily when complex motions are presented such as movement to avoid obstacles. The slave tends to lack information since it bases its motion on the motion of the master and thus does not inherently recognize problems arising in the surroundings.

A more complex and intelligent algorithm is one where there are effectively two slaves and no masters. The algorithm here stipulates that both robots have equal intelligence and thus need to determine their own paths and collision avoidance schemes independently from each other. The input for determining these paths is based upon some sort of external reference, perhaps the center of mass of the carried bar, and thus acts as the master to lead the slaves through the desired motion. This is the algorithm which has been implemented in this thesis and will be discussed in detail in a later chapter.

Another scheme is that of the Activity Controller [6, 3]. This scheme is analogous to the hierarchy control algorithms since work is divided up into a series of levels with the underlying assumption that sequentially intelligent functions are to be performed at increasingly higher levels. The low intelligence functions such as stepper motor signal generation occur at the lowest level

in the robot controller, path generation and collision avoidance schemes occur at the next higher level in an on-line computer, and more complex functions such as coordination of multiple robots and supervisory control occur at even higher levels perhaps in a mainframe computer.

This theory utilizes a hierarchical scheme but also adds some underlying assumptions to the intelligence of the motion. It makes allowances for much more complex functions and defines the various levels at which certain decision making processes are to occur. With this scheme one has the power to simultaneously make decisions for problems such as process scheduling, robot scheduling, optimization of sequences and even jobs themselves for particular robots as well as guidance through collision situations and recovery algorithms. A more detailed discussion is provided in chapter 6.

2.9 Concluding Statements

Coordination of multiple robotic arms is a relatively recent topic of concern with most of the advancements having been made in the past few years. The industry is starting to actually implement the truly coordinated machines in real world situations. With the development of inexpensive and very fast computers comes the ability to

implement the lengthy and cumbersome algorithms needed to intelligently control the several robots. It will not be unrealistic to dream of systems which can carry out extensive self maintenance or perhaps even become intelligent enough to determine their own motions through manufacturing processes with no "a priori" knowledge or help from the humans.

This chapter has provided a general overview of the coordination of multiple robotic arms.

The discussion started with the general theory concerning coordination followed by the benefits and setbacks involved with implementation of multiple robots and concluded with a presentation of typical applications and examples of some algorithms which have been developed.

CHAPTER III

COLLISION AVOIDANCE IN COORDINATED ROBOTS

3.1 Introduction

This chapter provides a synthesis of information derived from several published sources in the area of collision avoidance. It describes a chronological progression of the technology starting with a robot avoiding a stationary object or objects, through avoiding moving objects, and ultimately leading to two coordinated robots avoiding each other.

3.2 Stationary Obstacle Avoidance

The task of having a robot avoid a stationary object located within its work envelope can be usually accomplished through off-line scheduling. Alternatively, an operator could manually guide the manipulator through the task with a teach pendant, and record the manipulator sequence, keeping in mind that the obstacles must be avoided. At that point the motions can be played back and parameters such as efficiency and cycle time can be evaluated and improved, if desired.

For a variety of simple tasks, this technique is generally effective; however, there are two distinct disadvantages. First, the cost of the "teaching" process

is high since it must occur either on the assembly floor where production lines must be stopped, or in a simulation laboratory where the equipment itself may be expensive. Secondly, there is always the risk of damaging the equipment, the environment, or even injuring the operators. The solution lies in the progression of computer graphics and numerical analysis where different algorithms can be developed and tested through simulation long before they are implemented at the assembly line. Using this approach, downtime is held at a minimum and any chance of accident or damage can be realized in advance using the computer simulation [7].

An efficient algorithm tested in a computer simulation was set forth by Lozano-Perez in "Configuration Mapping" [8]. In this scheme, the manipulator is shrunk to a point and the objects are grown appropriately to account for the shrinkage. Thus, as long as the manipulator point is out of the grown obstacle region, no collision will occur. Figure 3.1, which is taken directly from this paper, illustrates their example of the mapping. Note that since the object in this case is only translating, the objects need only be enlarged on certain surfaces.

This process is justified due to an improvement in speed of data processing. Instead of comparing each point on the manipulator to each point located on the various

objects (a large mass of data), one can simply compare the single point of the shrunken manipulator to the enlarged objects. However, there are also certain disadvantages associated with this scheme. The main disadvantage is that

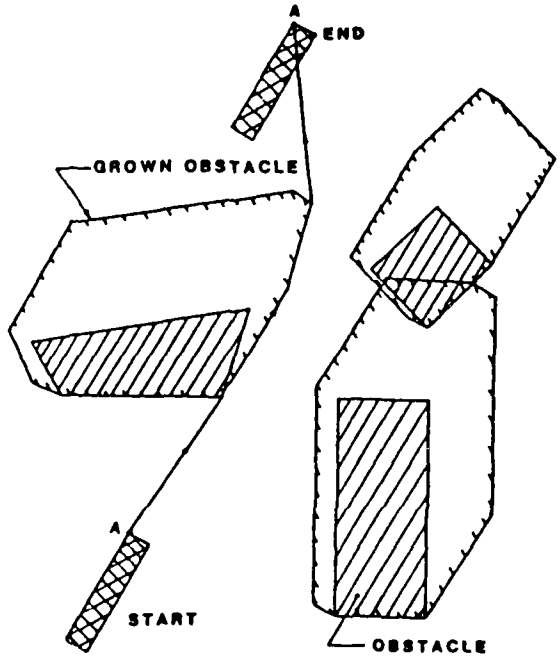


Figure 3.1
Configuration Mapping in Translation

if there are many objects located in a congested area, the enlarging process may choke off all of the available pathways for the manipulator to follow.

In light of the above potential difficulty, another solution is referenced where instead of refining an algorithm, a brute force "calculator" is developed where

the mass of data can be processed in real time. This was realized through the utilization of a controller consisting of sixty-four microprocessors whereby the on-line control was accomplished through parallel processing [9]. In this process, an object's location and dimensions are provided as input information. Through various path planning algorithms, a suitable movement is determined and implemented. Note that in this case a starting point, ending point, and obstacle characteristics can be externally entered into the machine which calculates possible paths and implement the "best choice".

A method for significantly decreasing the amount of needed calculations is to represent objects by simple polygons such as circles, squares, and trapezoids (refer to figure 3.2). Thus, the object itself is defined by a simple equation which is easily and quickly compared in a program as opposed to, say, a large locus of surface points which must be analyzed one by one in a lengthy process. This too has its drawbacks in that again certain shapes are poorly represented and take up large amounts of extra space. For example, a jagged, spindly, star-shaped object has a small total area yet if one encloses it in a square, a much larger polygon would develop.

3.3 Avoiding Obstacles in Motion

The next order of complexity in solving the collision avoidance problem occurs when of a manipulator moves through a field where the obstacles are in motion. This creates the additional problem of having yet another set of data to process. It was remarked by Kokaji [9] that the sixty-four parallel on-line microprocessors are able to

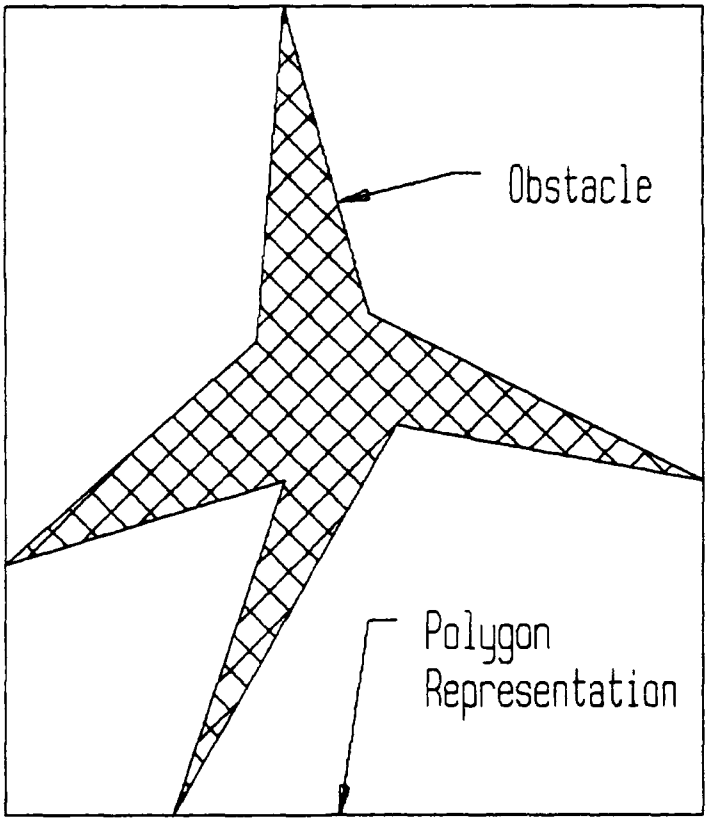


Figure 3.2
Polygon Representation of Obstacles

handle this situation. An effective algorithm was also suggested by Khatib which is based on the "Artificial Potential Field Concept" [10]. Here, real-time collision avoidance by a robot is achieved by utilizing visual sensing in an environment with moving obstacles. The theory set forth is to assign an attractive charge to the desired destination while assigning a repulsive charge to obstacles along the way. Thus, not only the end effector is involved in the process, but each point of the manipulator is also considered.

The advantages of the process are many. Most collision avoidance algorithms are implemented at a high-level control whereby the speed of process is paced by the time cycle at this level. This is several orders of magnitude slower than the real time process of the robot, thus limiting the speed at which the robot can perform its work. This algorithm is implemented at low-level control thus allowing feedback from a complex environment while still maintaining high process rates. Note that this is not meant to replace high-level processes, but to better utilize those at lower levels.

A second advantage is that different potential field functions can be tested so as to provide different levels of avoidance. For instance, objects may provide a linear function of distance versus force to have large or small,

positive or negative function slope. They may also have some sort of exponential relation where the force is a function of distance raised to a prespecified power. A disadvantage is that the path chosen is a function of these forces alone which may not be the optimal choice. Here, a higher level control may be influenced by this algorithm in that the potential field functions provide a partial decision as to the path. Then, it can be used in conjunction with other optimal path algorithms.

3.4 Navigation of Robots Through Unknown Terrain

The next step of complexity is to move a manipulator through a field with no prior knowledge of the obstacles located in that field which may also include other robot arms. The actual research was motivated not to control multiple robots, but to navigate a mobile robot through unknown terrain [11]. However, a very good analogy can be made and the theory will transfer to multiple robot coordination well. The two underlying concepts here are the fact that first, the manipulator needs an external sensory system to detect the obstacles; and second, once the robot realizes this object, it must "remember" its location and size in order to make better judgment decisions for the next sequence of movements.

For the sensing aspect, several methods are introduced to include vision locally and globally as well as tactile proximity sensors. The feedback from coordinated robots may be utilized to have one robot communicate with the other about its location, and vise-versa. However, this problem is more relevant for the computer vision and sensor specialist. For the purposes of this thesis, it will be assumed that an accurate set of data has been produced. The real problem now arises in that the controller must be able to manipulate vast amounts of data which is constantly changing. One approach is "Quadtree-Based Path Planning" [12, 26]. Very simply stated, this approach assigns various gray-scale values to locations in the area of movement. These values are thresholded and immediately values over the threshold are ignored, thus decreasing the number of computations. This process is adaptable so as new information is received, the "picture" will be changed to reflect new possible or more effective pathways. Through this method, a learning algorithm can identify obstacles quickly and ignore their location's computations to speed up the movement process.

3.5 Mutual Collision Avoidance of Coordinated Robots

This discussion has been geared to lead into an analysis of two coordinated robots working towards a common goal and simultaneously realizing each other's actions and

locations so as to prevent any possible collision. Note that this algorithm must be implemented in real time and must be automatic enough so that a programmer will be able to identify a new process (say an assembly) to the robots and not have to worry about their collision. The name given to the process of two coordinated robots avoiding each other is "Dynamic Avoidance" as opposed to static avoidance where a robot is avoiding stationary or well-defined objects. One method of dynamic avoidance is to simply have the controller always program the robots to two completely separate paths. This becomes unrealistic as it would make the problem of path planning much more complex, if not impossible, and probably at least greatly increase the required amount of computation time.

Two very feasible alternatives arise when the concept of dividing much of the work among several controllers is evoked. The first is an adaptation of the "Artificial Potential Field Concept" [10] using two controllers. Here one can have controller 1 calculating the attractive and repulsive forces for robot 1, where it assigns a repulsive charge to the dynamic robot 2 and at the same time have controller 2 doing vise-versa for robot 2. This way, there are no more added computations per controller and real time calculations are still maintained. This algorithm is also a very useful since the two controllers can provide

feedback to the coordination controller thus decreasing the work load at that level. Note that the corollary of "if one robot avoids the other, only one collision avoidance mechanism is required" is false since each robot must also independently avoid other obstacles at the same time. However, there would be a time savings if instead of having both robots check each other's position, only one of them performed the mutual collision avoidance schemes. Both would still search for obstacles in their paths but only one would be able to detect the other's presence. The setback here is that this tends to be a master/slave situation where the slave would be checking the master's position and the master would move with ignorance to its mate's position. Thus, it may unknowingly cause the slave to drive towards an obstacle. For the savings in computation time, the small loss in intelligence may be acceptable.

A second alternative is the permission technique. This algorithm also lends itself well to real time since many of the processes can be divided among various controllers. The idea here is that each robot seeks permission to enter a particular "space" before the actual motion takes place. At a first glance it seems that this is a very "last minute" process, but in reality the permission process occurs over a much larger scale in that

permission is requested to perform a sequence of movements and the entire sequence is checked against obstacles and the other robot. This lends itself to speed since one of the ends has permission being requested from two separate robots, and the opposite end has a constant influx of information as to presence/absence of devices, sensors, input, etc.. In addition, in the controller there is an intelligent decision making algorithm which is very straightforward and capable of real time calculations.

Another physical problem encountered in coordination is that of twisting (refer to figure 3.3). In this situation the two robots which are holding an object, may be directed to rotate in a horizontal plane thus causing the links of the robots to twist upon themselves and perhaps collide.

There are several ways in which this problem can be solved. The most straightforward, yet most limiting, format is to restrict the roll, pitch, and yaw of the object to angles between Π and $-\Pi$ radians as compared to the world frame. This effectively prevents the robots from twisting to the point of contact but does limit the movement of the object from obtaining certain possible orientations.

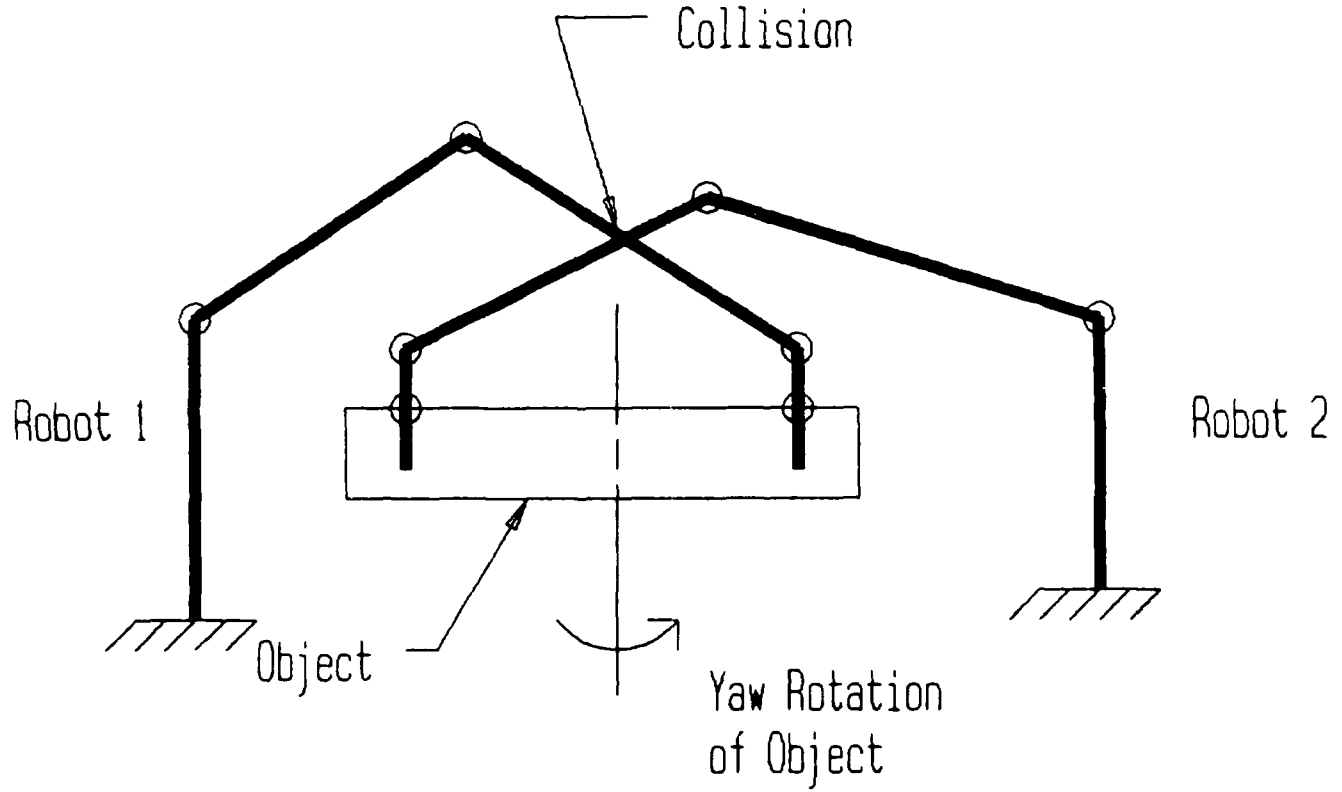


Figure 3.3
Twisting Collision

The second solution is to implement the Graph Node Search algorithm not to the robots and their surroundings but between certain nodes along the length of the robot's arms. Thus, at every iteration the distance between each of the nodes is compared to some minimum value and when this value is crossed, a recovery routine takes over.

3.6 Concluding Statements

Collision avoidance can be considered to serve a dual purpose. First, it provides vital information regarding the whereabouts of obstacles present in an area so as to allow a computer to make intelligent decisions about the path planning problem. Secondly, it provides safeguards from the catastrophic occurrence when robots come in unintentional physical contact with an object to cause some sort of damage. In order to maintain efficient, cost saving robotic operations, the concept of collision avoidance must be well understood and implemented in the process.

CHAPTER IV

PATH PLANNING IN COORDINATED ROBOTS

4.1 Introduction

Path planning involves the determination of an object's trajectory between a prespecified start point and some end point while attempting to minimize the cost of the movement in terms of time, or energy spent. Path planning techniques must also take into consideration such problems as finding tne minimum horizon or in other words, the most efficient path to follow while still satisfying a variety of through-points.

There exists a fine line between the ideas of collision avoidance and path planning. This is mostly due to the fact that the two are dependent upon each other for calculations. Also, there is rarely any utilization of one without a similar implementation of the other. As shown in one of the solutions to the path planning routines below, a combination of both path planning and obstacle avoidance is suggested since they are so closely integrated.

This chapter deals with the problems encountered in the planning of routes for the coordinated robots to follow. These problems include timing, deadlocks, and recovery actions as typical examples. In addition to the

discussion of these problems, a few examples of path planning techniques which are utilized in industry are presented.

4.2 Path Planning in Coordinated Robots

Path planning for a mobile robot or a single robot arm takes several variables into consideration. The choice of these variables depends on considerations such as what types of tools are being used, what are the tools' availabilities, how close can the manipulator approach the workpiece or the boundaries of the work environment, and is there a timing sequence involved where a closely controlled process is occurring. These factors provide a firm base and a well defined set of restrictions on which the rules for path planning can be formulated. Thus, path planning at this level is a relatively easy process which simply requires some prior thought as to effective and efficient paths for the robots to follow given the guidance of the constraints.

The coordinated robots also have these types of problems and solutions. However, since there is more than one manipulator in the work envelope, some other problems present themselves to the path planning scheme.

The most apparent of these problems is the fact that the path of one manipulator will constantly be a function

of the path of the second manipulator and vise-versa. Thus, in order to determine the route that a robot must take, the robots not only have to monitor obstacles in the work envelope, but they also have to take into account the movements of the other robot. In the discussion of coordinated collision avoidance, many of the complexities of data processing and computation speed were identified as being inherent problems with limited solutions. Since in path planning there is the same situation of multiple robots communicating to each other, these same problems apply for the same reasons. A vast amount of data must be manipulated for intelligent, on-line path planning, and efficient and powerful algorithms must be developed in order to effectively process this data.

The second major problem is that of proper timing [22]. With one manipulator, the exact moment in time that the manipulator occupies a position usually is not as important as the fact that it must travel from a start point to an end point smoothly along a predefined path. With multiple robots, this timing becomes a very important factor as each of the robots must be aware of the other's movements at all times if it is to generate collision free paths. In the case of a mutually held object, the mutual timing is important for the obvious reason that the part is being rigidly held. If the timing is not very close to

perfect, the robots may damage the part. While it is true that the collision avoidance schemes would provide checks against collision, a prior knowledge of the motions' timing could provide a much quicker means for the individual robots to perform their path calculations.

There are at least two solutions to these problems. First, if the process is simple enough, it can be stripped of its intelligence and made into a hard automation process. Thus, all of the path planning can be performed off-line and tested for accuracy. Since this discussion is geared toward intelligent manipulators, this solution is trivial.

A better solution is to implement the simple path planning techniques developed for single manipulators and rely upon the collision avoidance routines in order to prevent catastrophic failure of the manipulators. This is advantageous in the sense that since those collision avoidance routines are being run at all times, they might as well be utilized to help generate paths. Thus, these collision avoidance schemes will allow simpler path planning algorithms to be as useful and complete as the more complex algorithms. Also, a decrease in computation time should be realized due to the use of the simpler algorithms. The combination of path planning and collision avoidance techniques provide very straightforward solutions

as the manipulators ultimately reach their destinations in a collision-free movement. However, there is an inherent cost in attempting to attain path efficiency.

Since the obstacle avoidance schemes do not consider the "a priori" knowledge of the obstacle's positions, there is no guarantee that the path that is forced by the collision avoidance schemes will minimize energy or time. This becomes increasingly true as the workspace becomes more clustered with obstacles as the manipulators spend more time trying to avoid obstacles than they do trying to follow efficient paths towards their goals. But again each problem is very independent and trade-offs such as these must be weighed to find an appropriate solution to individual problems.

The third problem is that of deadlocks. Imagine a process where two manipulators share workspace and tools but perform different operations on the same part. The rate at which they do their work may be different; therefore, they start and finish a part cycle as they are able to. The deadlock in the process may occur when both robots reach for the same tool. If both need to use the tool, at the same moment in time, they effectively prevent each other from continuing the process. A conflict resolution strategy is required in this instance.

The solution to this problem is the implementation of recovery algorithms which provide an intelligence to decide upon appropriate actions and direct control to the more qualified manipulator. A more complete discussion of this issue is included in the following section.

4.3 Available Algorithms

Two path planning routines are discussed which provide real-time, proven solutions to the problem of path planning in a work-cell with only one robot. These routines do lend themselves well to implementation with multiple robots and thus will be extended to be implemented in work-cells which contain multiple robots. These routines are the Activity Controller Scheme [6, 24] and Resolved Position Control [13].

The Activity Controller Scheme is a structure which has been set up in order to provide guidance in developing a large scale system (refer to figure 4.1). In general, an activity controller (AC) (usually a dedicated computer) is assigned to each work-cell in a plant. Each of these microcomputers is in turn linked to a coordinating computer at the plant level which in most instances, is some sort of mini-computer. The plant controller can effectively control the processes of the entire operation as it has an

indirect communication link with every device in that plant.

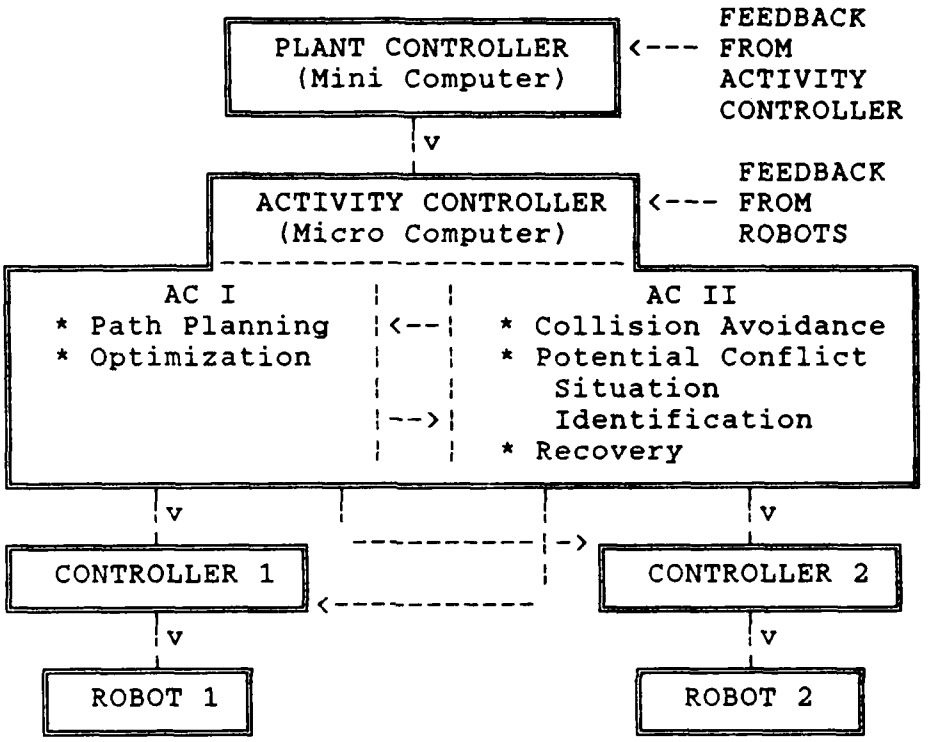


Figure 4.1
Schematic of an Activity Controller

The activity controller receives instructions from the plant computer as to the tasks it is required to perform. Thus, its goal is to create specific instructions for the robots to follow in order to accomplish that specific task. This instruction generation process takes into consideration available tools and stock, time requirements, the particular characteristics of the manipulators, the coordination of the manipulators, and their path planning.

The activity controller deals with these complex and complete tasks through its division into two separate units. The two units have unique functions, but rely upon information which is received from the other half. This information is transmitted through a feedback communications network which links the activity controllers, the manipulators, and the plant controller.

The first unit, or AC I, deals mainly with the planning and optimization of the paths in ways similar to those which have been discussed. Routes, paths, through-points and the such are all generated and sent to the robot's controller.

The creativity comes in the implementation of the second half of the activity controller or AC II. This section has the ability to not only handle impending collisions but it looks ahead in the process to identify potential collision situations. This effectively operates as a path planning routine as the path is generated with the knowledge of the these potential collisions about to occur.

AC II implements itself in a three step process. The first step is to identify those items which both robots must share in the process and with this knowledge develop a set of potential conflict situations which may arise due to

the acquisition of these shared resources. This is done simultaneously as the process progresses and allows for real time implementation as the robot can perform its manipulations independently of AC II and will not be hindered by its progress or lack thereof.

The second step is the use of "prevent" and "detect" algorithms. The first of two alternate approaches is to have the computer monitor potential conflict situations. If one is detected, the computer will direct the robot to choose an alternate route or to simply wait. The second alternative is to have the robots ask for permission before they can proceed on with their tasks, thus providing a last line of defense against any collision from occurring as alternate routes can be chosen.

The third step simulates intelligence as it provides a recovery algorithm for the system to follow in case all of the other safeguards have been unsuccessful. The situation occurs where the manipulators progress to a point where there is an impending collision to occur. For example, the two grippers may want to retrieve the same part or tool. Up until this point however, AC II does not detect the impending collision due to the fact that it has been occupied with other computations. Here, AC II will be interrupted and forced to implement the recovery algorithms where it provides the manipulators with an intelligent

means by which they can control themselves in order to regress away from this situation.

The most optimistic of recovery algorithms provides answers for two problems. First, it will be the decision maker as to which of the robots is to continue on with its routine as planned. This is a relatively simple process as it would be able to easily evaluate which robot has the more important task at hand or which is in more need of the device. Second, and a much harder problem, is to instruct the waiting robot as to some other work which it can perform while it is waiting for the tool to be freed. This presents itself as a fairly difficult task since there are many variables which would influence this decision. The waiting robot may be able to use another tool, since another function may still need to be performed. This action may help the first robot speed its routine thereby making the tool available sooner, or in the worst case scenario it may simply be directed to wait until the tool is once again available.

While this provides groundwork for the implementation of multiple robots, the same problem of computation speed arises. This is a complex algorithm and the extent to which it can be used is definitely a function of the computing power available.

The second algorithm which is available is called Resolved Position Control. Here, the path function is generated via some external reference. This reference could be the center of mass of a mutually held bar or perhaps the point of contact between a part and where it is to be drilled. This algorithm is discussed in detail in Chapter 6 as it is the algorithm which was implemented for this research.

4.4 Two-dimensional Approaches

Along with these two algorithms, two two-dimensional methods are discussed which provide insight into the basis of path planning. While they do not readily provide complete solutions to the problem of multiple arm path planning, they do give guidelines to follow in specific aspects of the path planning problem. These techniques are based upon the Distance Function [14] and the Local Dynamic Path Generation theories [15].

The first of the two theories is the Distance Function Theory. A brief discussion of this approach is given below. This topic is included in this thesis since it emphasizes several points which provide useful information when applying various path planning algorithms.

The use of distance functions basically entails an optimal-control problem which requires that the following energy function be minimized:

$$J = F_0(\underline{x}(0), \underline{x}(\tau)) + \int_0^\tau F(\underline{x}(t), \underline{u}(t))\, dt \qquad (4.1)$$

where "F" is a predefined, arbitrary cost function, "x" represents the state variables, and "u" represents the system's input. Time t runs from 0 to finish time τ.

The state variables in equation (4.1) would be both the position and the velocity of the manipulator. Therefore, the minimization of "J", which is indicative of the movement's energy, would be directly related to the route which the manipulators follow.

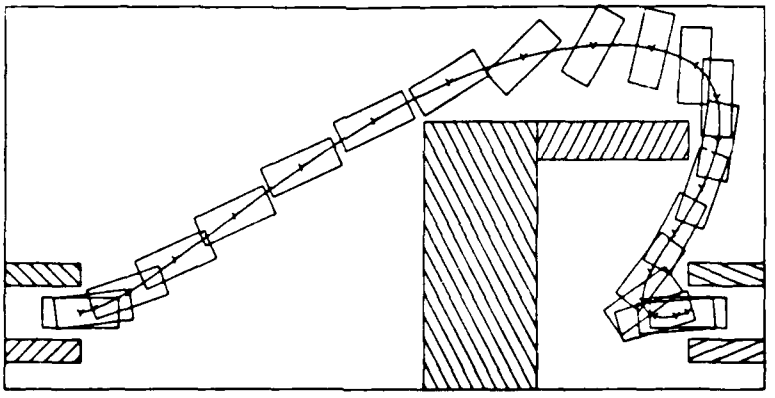


Figure 4.2
Implementation of a Distance Function

Figure 4.2, which is taken directly from Gilbert's paper on Distance Functions [14], is a graphic illustration of a path generated with the minimum energy stipulation. This figure shows actual output from a numerical simulation that he performed and vividly demonstrates the intelligence with which the path is chosen. Note how the object rotates counter-clockwise at the high point of the path in order to allow a vertical threading through the obstacles. Also note the smooth path taken. This path does not necessarily skirt directly around the obstacle, but rounds the corners in a continuous motion.

These calculations point out several properties which allow for intelligent movement. First, since the actuator moments and torques are proportional to the energy, a minimization of that energy will decrease the proportional torque and moment at the individual joints. This has the effect of causing the motion to be smooth and fluid as opposed to being sporadic.

Secondly, since energy is a function of distance, minimizing energy will be directly related to a minimization in distance. There are many paths that a manipulator can follow between the start point and the end point. The minimization of the distance of that path will most obviously provide an optimal solution.

The second of the theories is the Local Dynamic Path Generation Theory. Note the use of "generation" versus "planning". This indicates that the path is being chosen only at the local level in the vicinity of the manipulator and that there is no "a priori" knowledge of the obstacles.

Two underlying assumptions are made at the onset of this type of movement. First, the entire body of the robot is able to detect an obstacle as if the robot is sheathed in some sort of tactile blanket. This function could be realized through many sensors or perhaps more realistically through numerical calculations but regardless, the information is assumed to be available. Second, once the robot detects the said obstacle, there is an inherent ability to skirt the obstacle. This not only includes the end point of the effector moving around the obstacle but also the individual arms of the robot. Thus, the solution is not to simply have the end effector feel its way around the perimeter of the obstacle, but to have the entire body sense around the obstacle.

Once these assumptions are made, the theory is very simply stated. There is a known start point and end point in space. The most desirable path to follow is a straight line between these two points. The robot follows that path until an obstacle in detected. Upon this detection, the robot uses its blanket of sensors to skirt around the

perimeter of the robot until it reaches the original path on the other side of the obstacle whereupon it continues on with its motion.

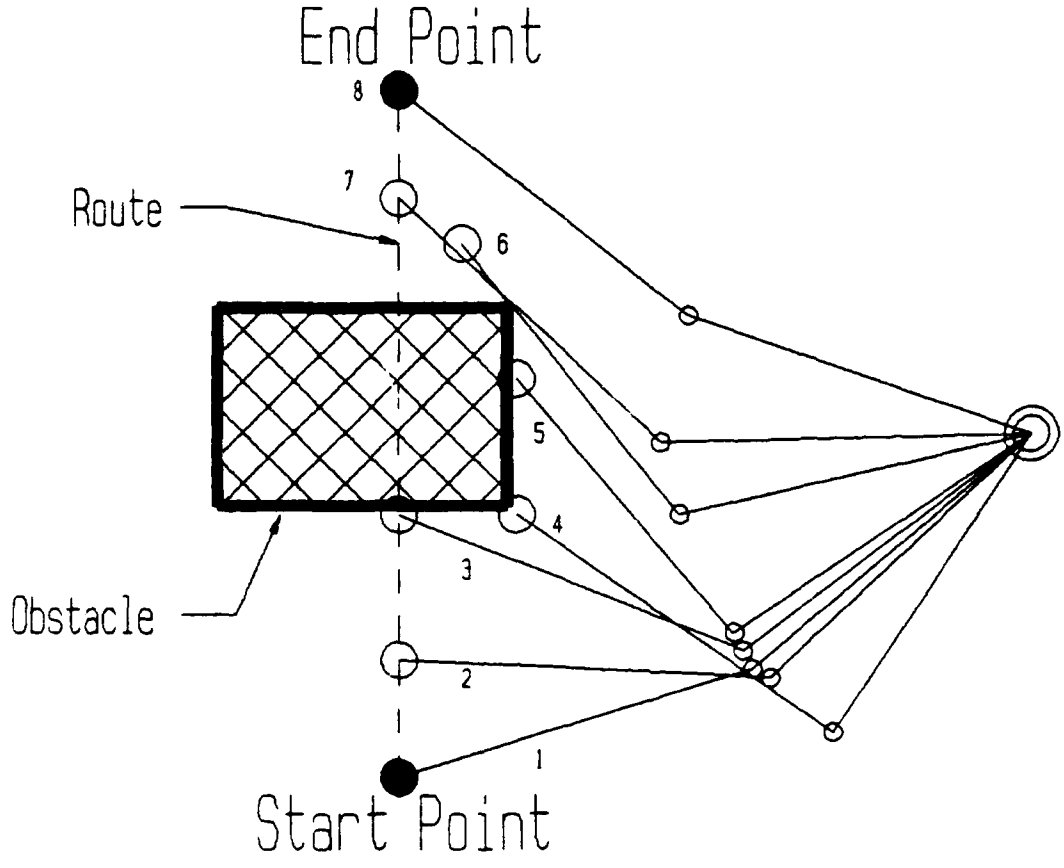


Figure 4.3
Local Dynamic Path Generation

Figure 4.3 illustrates the motion described above. The manipulator starts and travels along its most desirable route (positions 1 and 2). Once the obstacle is detected

(position 3), the sensors on the end effector allow the manipulator to skirt around the obstacle (positions 4, and 5). Note in position 6 that the body sensors now provide the input for avoidance. Once again, the robot finds its original desired path (position 7) and continues towards the end point.

This theory brings up several interesting points. First, there are two possible alternatives of travel in order to avoid the obstacle. The manipulator can skirt either to the left or to the right. Since there is no "a priori" knowledge of the obstacle, an intelligent decision cannot be made as to the most feasible choice and a default direction is chosen. This thesis suggests that even minimal knowledge of the obstacle in the field would provide enough information to make an intelligent decision and that this information should be included in the formulation of the theory.

Secondly, the concept of the "virtual obstacle" is introduced. The virtual obstacle not only includes the space occupied by the physical obstacle itself, but it also includes that space which is unobtainable by the manipulator due to the presence of the obstacle. This extra space is referred to as the "shadow" of the obstacle (refer to figure 4.4). The virtual obstacle for a simple two degree-of-freedom robot is minimal when compared to

that of two robots mutually holding an object. Thus, a point well made is that when dealing with more complex systems or those cluttered with obstacles, movement may become very limited if not impossible with fields deemed simple for certain representations.

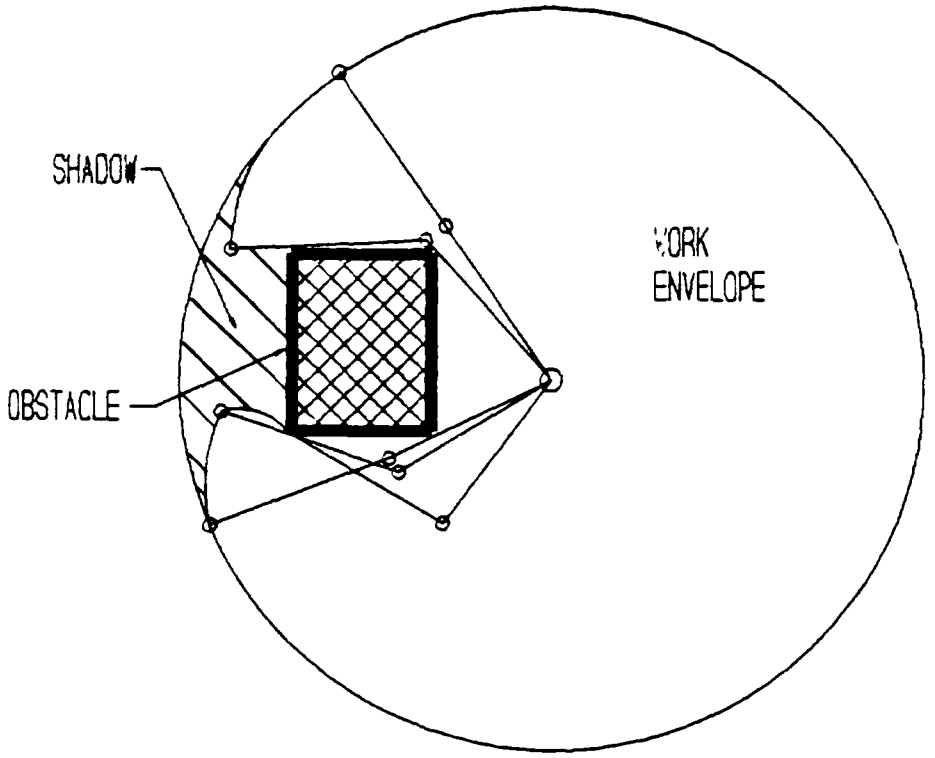


Figure 4.4
Shadow and the Virtual Obstacle

Third, is this routine's solution to obstacles located on the boundary of the work space. If the manipulator happens to skirt to the intersection of the obstacle and the boundary, there must be recovery from stopping at the

boundary. In this situation, the boundary is not considered to be an obstacle. The joint angle limitations and the singularities will prevent the boundary from imposing a discontinuation of movement. The manipulator will continue to skirt around the perimeter of the obstacle using the body sensors. Note that the end point of the robot does not have to be the point to contact with the obstacle during the skirting process. The body points can provide this function since they are equally well-equipped with sensors.

The main point to be emphasized by this theory is that when dealing with an obstacle field in motion or one where the exact locations of the obstacles are unknown, a local, dynamic path generation scheme can provide good solutions to the problem of movement. These solutions may not be the time or energy optimal but they do indicate possible and reliable paths for the manipulators to follow.

4.5 Concluding Statements

While the problem of path planning is closely related to collision avoidance, there are distinct differences. The presence of obstacles do affect the generated path. However, instead of being concerned solely with the avoidance of obstacles in the path, path planning

techniques can be designed to determine the path of least energy or time spent given the location of the obstacles.

This chapter gives a summary of some of the theory behind the path planning techniques employed in the manipulation of multiple robotic arms. Several example algorithms are discussed in addition to some of the problems encountered while attempting to perform these calculations.

CHAPTER V

SYSTEM MODELING

5.1 Introduction

The general theory behind the coordination, path planning, and collision avoidance of coordinated robots has been presented in the previous chapters. Several examples of viable algorithms as well as benefits and drawbacks of those systems have been discussed. This chapter will concentrate on a specific task and set of algorithms which have been implemented in computer simulation. In addition, it will describe the various technologies and theories on which the present work has been based.

The specific task presented in this thesis is to guide two coordinated robots holding an object, through a work-space having obstacles. The motion of the system is only to be defined by the start and end positions as well as the orientation of a coordinated system which is located arbitrarily on the object's surface.

Several other assumptions are made which further define the problem to be solved:

1) The two robots are to have a shared work envelope. This means that there exists the possibility of collision between the two

robots as well as the problem of twisting collision of the two robots holding a bar. Twisting collision is the situation whereby the object being held by the robots rotates in such a manner as to cause the robot arms to twist upon each other and collide.

2) The two robots are to rigidly hold an object and coordinate that motion as opposed to two robots working simultaneously on a work piece as in the case of the vice/tool scheme.

3) The initial grasp that the robots have on the object is predefined. This grasp can be in any combination of positions and orientations possible on the held object.

4) Every conceivable position and orientation within the shared work envelope is to be obtainable by the configuration.

5) Path planning and obstacle avoidance in a space with obstacles is to provide intelligent motion. Intelligent motion in this case describes a successful, and efficient motion which is derived solely

from the arbitrary data field information and with no external help from human input.

Along with these assumptions, several realistic situations have been introduced which make the problems and solutions compatible with real-world problems encountered in industry. These examples will be illustrated in a later section.

5.2 Hardware Modeling

The PUMA (Programmable Universal Machine for Assembly) Robot arm made by Unimation was chosen to be the model for the work conducted in this research (Figure 5.1). This choice was made for many reasons including its readily available parametric values, kinematics, inverse kinematics, and Denavit-Hartenberg representations.

Another reason for this choice was that the PUMA robot possesses six degrees-of-freedom. The work which was performed by Lim and Chyung [13], utilized two Rhino robots each having only five degrees-of-freedom. As a result, the range of motion possible by the coordinated robot scheme is limited to cartesian movements in the work envelope and only one of the roll-pitch-yaw angle changes (depending on how these angles are initially defined). Note that these position and orientation changes refer to changes in the grasped object's coordinate frame, and not the robots'.

For example, if the z-axis is oriented perpendicular to the floor, a motion with the two Rhino robots would be limited

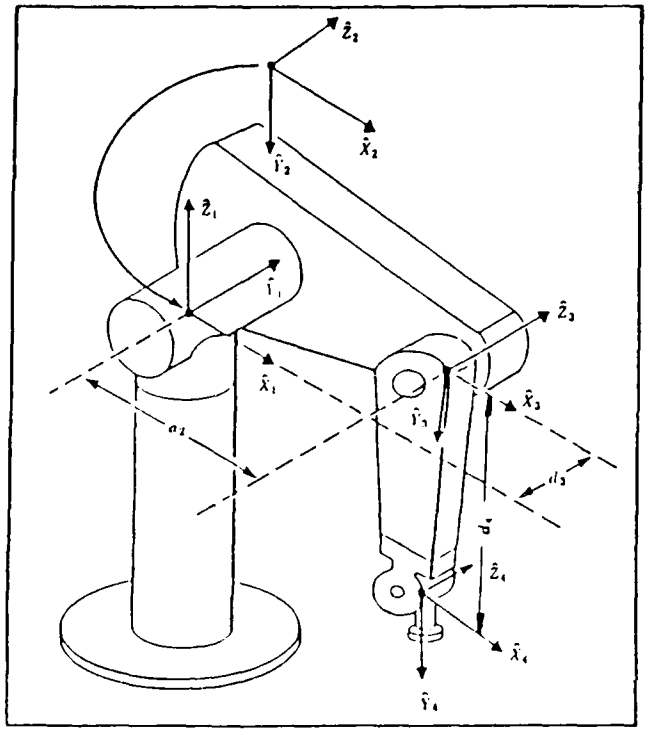


Figure 5.1
The Unimation PUMA Robot

to a change in the x-y-z cartesian coordinates and a rotation only about the z-axis. No rotation around the x-axis or y-axis would be possible except under very limiting situations.

In contrast, the PUMA has full six degree-of-freedom capability and is able to reach any position and

orientation in the work envelope. This introduces many more possibilities of motion and functions in operations such as, for example, picking up a piece and then drilling on the underside of that piece.

5.3 Transformations for Coordinated Robots

The primary goal in this position control scheme is to be able to generate the robots' transformation matrices given the transformation matrix of the mutually held object. This way, the motion can be described by a series of the object's transformations. This section draws upon the Lim and Chyung paper [13] with the extension being that it will be simulated and tested on six degree-of-freedom PUMA robots instead of five degree-of-freedom Rhino robots.

Let T define the standard homogeneous transformation matrix which describes the position and orientation of some satellite coordinate system with respect to a reference coordinate system (refer to figure 5.2). In Denavit-Hartenberg representation [20], the 4 by 4 matrix T is defined as:

$$T = \begin{bmatrix} N_x & O_x & A_x & P_x \\ N_Y & O_Y & A_Y & P_Y \\ N_Z & O_Z & A_Z & P_Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (5.1)$$

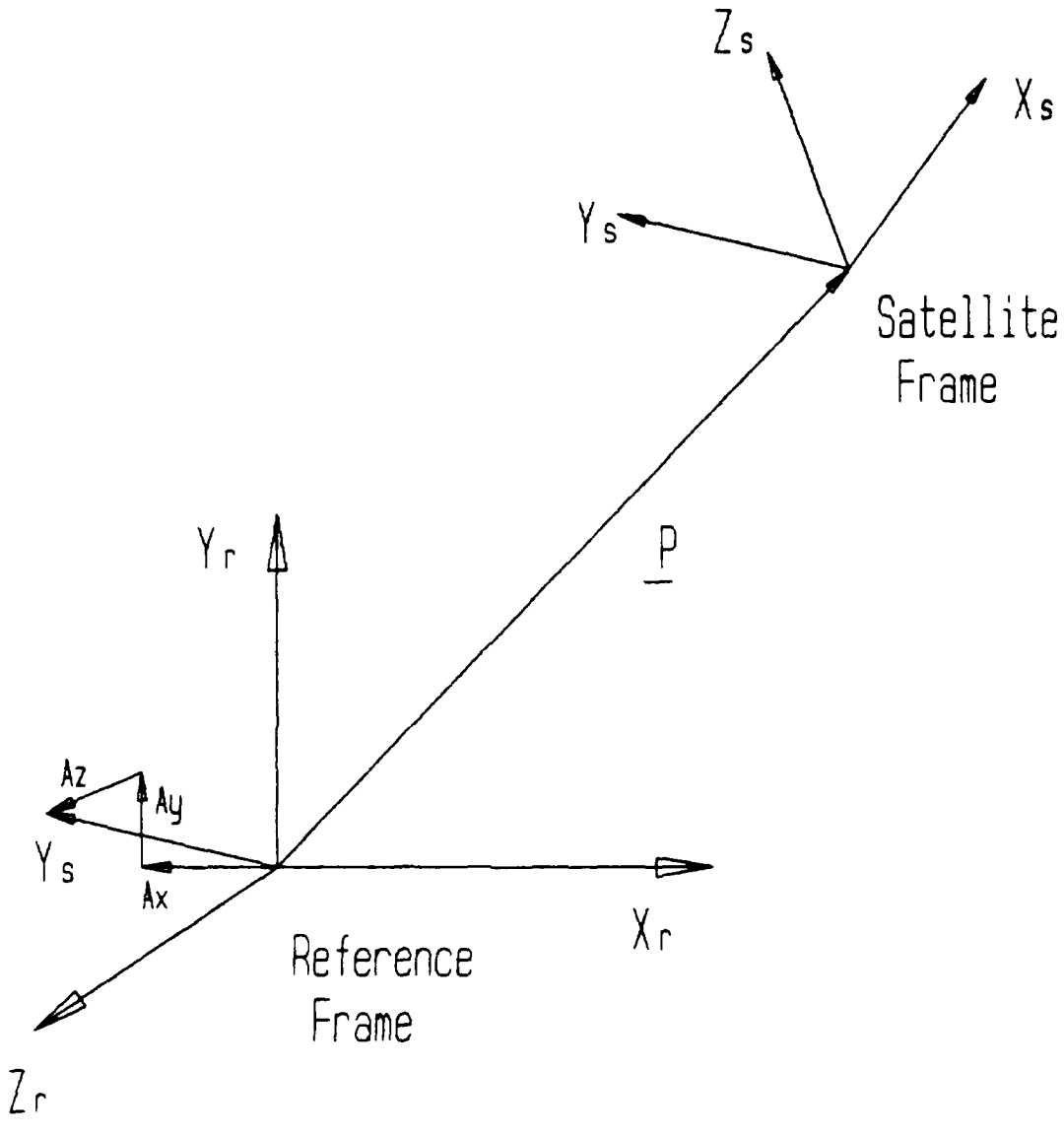


Figure 5.2
Representation of Satellite and Reference Frames

where N, O, and A are the projections of the satellite's axis onto the x-y-z base axis. For example, N_x describes the projection of the N-axis of the satellite coordinate

system upon the x-axis of the base coordinate system. P is the position vector of the satellite's origin in the base coordinate frame.

A standard notation must also be developed to describe the transformations between the world reference coordinate system, the mutually held object, and the robots' base and end effector coordinates (Figure 5.3). To accomplish this, the notation T_{ab} (rob, t) is used. Here, "T" indicates that this is a transformation matrix. The subscripts "$_a$" and "$_b$" indicate that this transformation is from the "$_a$" coordinate system to the "$_b$" coordinate system. Subscripts "$_a$" and "$_b$" can be replaced by "$_r$" for the reference system, "$_o$" for the object's system, "$_b$" for the robots' base system, and "$_h$" for the robots' hand system. Argument "rob" indicates which of the several robots is being referenced (e.g., 1 for robot number 1 and 2 for robot number 2). Symbol "t" is the present time progression. As an example, T_{bh} (1, 35) indicates a transformation matrix from the base coordinate system to the hand coordinate system of robot number 1 at the thirty-fifth time increment.

Another transformation matrix takes the form of T_{ro} (t). Here, there is a transformation between the reference and object's coordinate systems at time increment

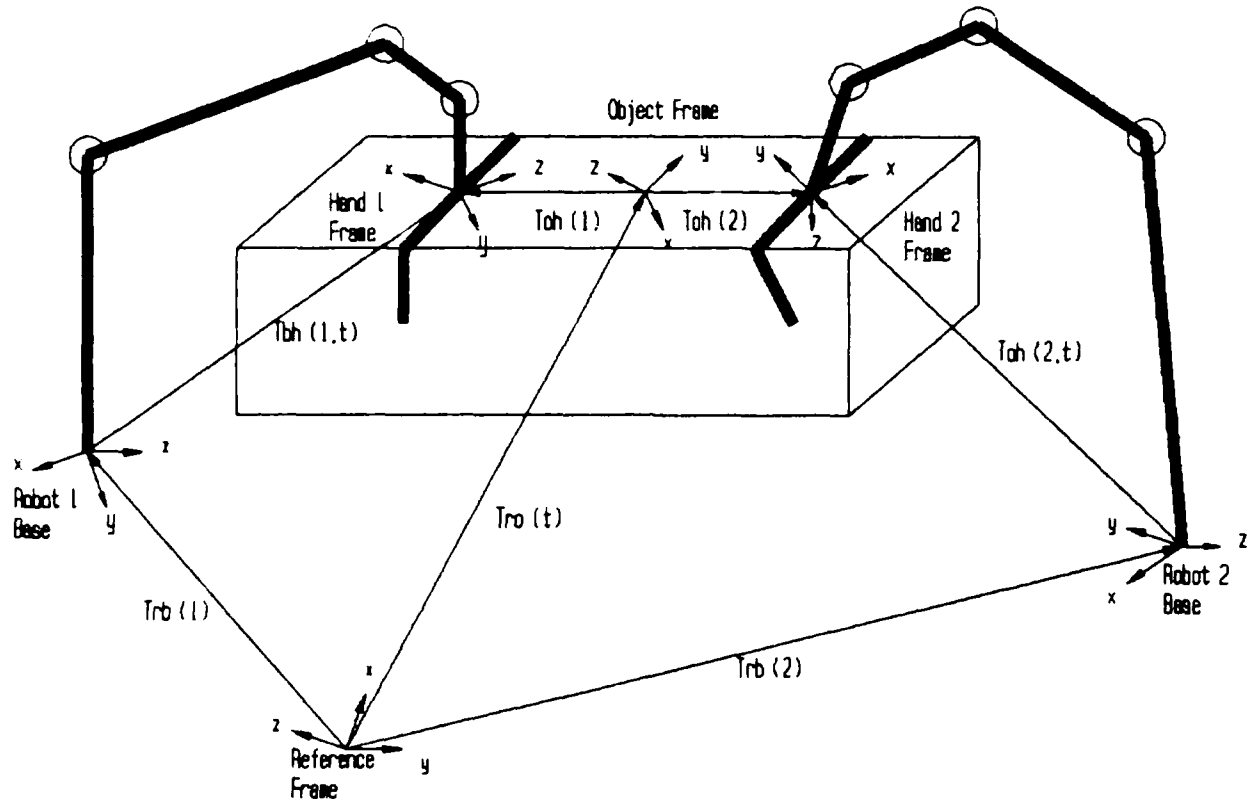


Figure 5.3
Transformation Vectors and Coordinate Frames

t. No robot is identified since none are involved with this matrix. Both of these notations may also be enclosed by brackets and followed by a superscripted "-1" to indicate an inversed transformation matrix.

With this notation, several factors can be reinforced:

1) The motion of the system will be described by the object's transformation matrix or T_{ro} (t).

2) Since the grasp that the robots have on the object is rigid:

$$Toh\ (rob,\ t) = T_{oh}\ (rob,\ 0) \qquad \text{for rob} = 1,\ 2 \qquad (5.2)$$

3) Specified initially as the data are the initial and final object transformations or T_{ro} (0) and T_{ro} (final).

Referring to Figure 5.3 and using simple vector algebra, the following equation can be derived:

$$T_{rb}\ (rob) * T_{bh}\ (rob,\ t) = T_{ro}\ (t) * T_{oh}\ (rob,\ t) \qquad \text{for rob} = 1,\ 2;\ \text{and all } t > 0 \qquad (5.3)$$

Setting t = 0 and rearranging equation (5.3):

$$T_{oh}(rob, 0) = [T_{ro}(0)]^{-1} * T_{rb}(rob) * T_{bh}(rob,0)$$

for rob = 1, 2;
and all $t > 0$ (5.4)

Substituting into equation (5.4) from equation (5.2):

$$T_{oh}(rob, t) = [T_{ro}(0)]^{-1} * T_{rb}(rob) * T_{bh}(rob,0)$$

for rob = 1, 2;
and all $t > 0$ (5.5)

This result is substituted into equation (5.3). Rearranging and solving yields:

$$T_{bh}(rob, t) = T_{ro}(t) * [T_{ro}(0)]^{-1} * T_{rb}(rob) * T_{bh}(rob,0);$$

for rob = 1, 2;
and all $t > 0$ (5.6)

Thus, a closed form solution is obtained for the robots' transformations. Note that this solution can be used for any robot for which the Denavit-Hartenberg transformations can be obtained. Now, the joint angles can be solved for using the inverse kinematics approach and the transformation calculations above.

As defined above, the problem states that $T_{ro}(0)$ and $T_{ro}(final)$ are the initial specifications provided. A method is needed to generate the $T_{ro}(t)$. At this point, the influence of path planning and collision avoidance is ignored but the method of generation will remain the same.

Any transformation matrix is basically composed of a rotation and a translation matrix. The translation matrix is simply generated by dividing the linear motion into an appropriate number of steps. In a computer simulation, each frame on the screen represents a particular instance in time. Thus, the number of steps between a start and end point is a function of the simulated speed of the robot. A robot in quick motion would have large changes in distances between frames or a low number of steps for a motion. A robot moving slowly would have a large number of steps effectively making the distances that the arm travels between frames be small. Therefore, the number of steps is chosen as a function of the desired speed.

The rotation matrix, however does not lend itself directly to an incremental change since it is composed of orthogonal vectors. Therefore, the orientation must be generated by other methods and then converted into a rotation matrix. One method is to represent the rotation matrix in terms of the six joint angles of an imaginary PUMA robot whose base is located at the reference coordinate frame and whose end-effector is located at the object's coordinated frame. Given these angles, the rotation matrix can be calculated through forward kinematics and installed for the rotation matrix of the object. This procedure proves to be a useful one for

situations where orientation changes similar to those of changing the wrist angles of the robot are desired. Unfortunately, the formulas for the kinematics are lengthy and require extra computation time.

A second method is to simply define the orientation through a standard scheme such as the roll-pitch-yaw system. Here, a rotation matrix is inserted into $T_{r,o}$ at the appropriate location to define the orientation. Thus, the roll, pitch, and yaw of the start and finish points can be calculated from $T_{r,o}$ (0) and $T_{r,o}$ (final) and then these angles are incremented as the motion progresses. The rotation matrix for the roll-pitch-yaw scheme is defined as:

$$\text{ROT}(\alpha,\beta,\Gamma) = \begin{bmatrix} C\Gamma C\beta & C\Gamma S\beta S\alpha - S\Gamma C\alpha & C\Gamma S\beta C\alpha + S\Gamma S\alpha \\ S\Gamma C\beta & S\Gamma S\beta S\alpha + C\Gamma C\alpha & S\Gamma S\beta C\alpha - C\Gamma S\alpha \\ -S\beta & C\beta S\alpha & C\beta C\alpha \end{bmatrix} \tag{5.7}$$

where "C" and "S" indicate cosine and sine functions, "α" is the roll angle, "β" is the pitch angle, and "Γ" is the yaw angle [17]. Therefore, given the locus of roll-pitch-yaw angles of the entire motion, the rotation matrix needed for $T_{r,o}$ (rob, t) can be calculated at each step through the use of equation (5.7)

5.4 Collision Avoidance and Path Planning Algorithms

Several previous sections have discussed a variety of path planning and collision avoidance algorithms. At this point, the specific methods that were implemented in the simulation presented in this thesis will be discussed in detail.

Human beings have an incredible ability to decide their route of motion. Data is collected by sensors such as the eyes for vision and the fingers for touch. Using this information humans can almost instantly compute a route which will take them quickly and easily to their destinations -- not to mention the generation of signals to the hundreds of muscles to accomplish that motion.

It has proven to be quite a challenging task to develop a mechanical system to simulate the actions of a human being. Until this time no one has completely and satisfactorily accomplished this task. By identifying computational features and cognitive processes, a programmer can attempt to simulate the intelligent actions of the human via computer control.

There are several factors which influence the choice of algorithms in the control of robots. These factors include such aspects as the speed of computation, effectiveness, and reliability. The computer must be able

to identify the obstacles and route through-points quickly, then determine an acceptable path, and finally generate the joint angles needed to drive the robot through its motion.

A combination of several algorithms was developed which provides a quick, and reliable method from which the computer can generate a collision free path.

This thesis introduces the term "Striving Technique" in order to describe the method by which path planning and collision avoidance are implemented for the movement of coordinated robots. The basis of this theory is to combine several of the more practical and successful features from other well known algorithms such as the Artificial Potential Field Concept [10] and Configuration Mapping [8]. Whereas these algorithms were designed to drive a single robot, the Striving Technique has been designed to drive multiple robots.

The equations derived in section 5.3 require that the matrices T_{r_0} (0), and T_{r_0} (final) be specified at the onset of movement. Therefore, the task for the path planning and collision avoidance algorithm is to generate an intelligent motion for the object's frame while moving from the start point to the end point.

The Striving Technique first redescribes the obstacles in the field of movement. Since the coordinate frame

located on the obstacle is closely controlled during the movement, it is most feasible to represent the object by this single point as opposed to representing the entire volume of the object. Thus, the dimension of the object is shrunk to a point located at the origin of the object's coordinate frame. As in the Configuration Mapping

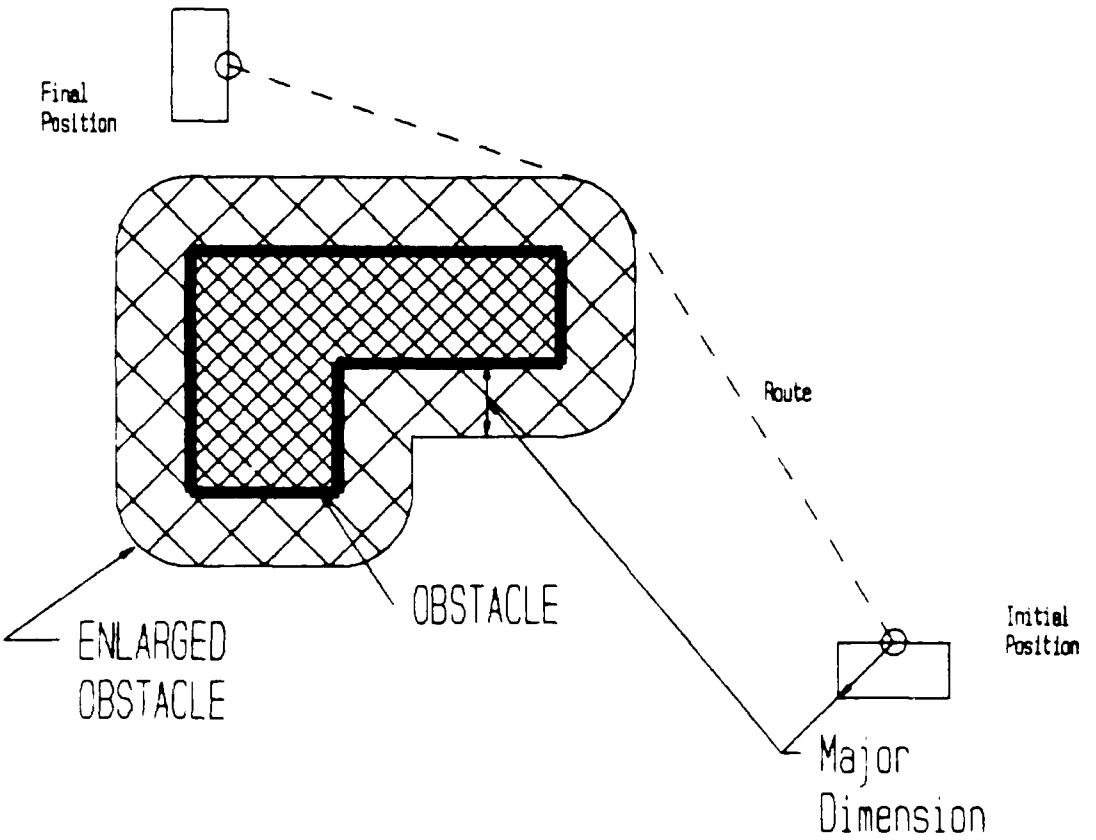


Figure 5.4
Configuration Mapping

technique [8], a shrinking of the object to a point must be complemented by an appropriate enlarging of the obstacles (refer to figure 5.4). Since the object has the ability to attain any orientation, the outer perimeter of the obstacles are increased by an amount equal to the major dimension of the object. The major dimension describes the distance from the object's frame to its farthest point on the perimeter.

Configuration Mapping has the potential problem of "choking". Choking is a situation whereby the obstacles have been enlarged to the point of eliminating all possible routes for the robots to take as the enlarged obstacles extend from boundary to boundary separating the start point from the end point. The problem of choking is relatively rare since it requires a large number of obstacles or physically large obstacles. Therefore, an obstacle field is to be chosen which will allow complete motion and the problem of choking will not become a prohibiting factor in the movement.

Thus, the Striving Technique defines that only the origin of the object's reference system need avoid the enlarged obstacles in order to prevent collision to occur. Due to the configuration mapping, this avoidance only involves comparing one point to the perimeter of the

obstacles as opposed to comparing all of the points of the object to all of the points of the obstacles.

In the discussion given above, collision avoidance has only been concerned with the object striking an obstacle. The potential collision of the robots are not included in this algorithm. The problem of robot collision avoidance is solved with the implementation of the Graph Node Search

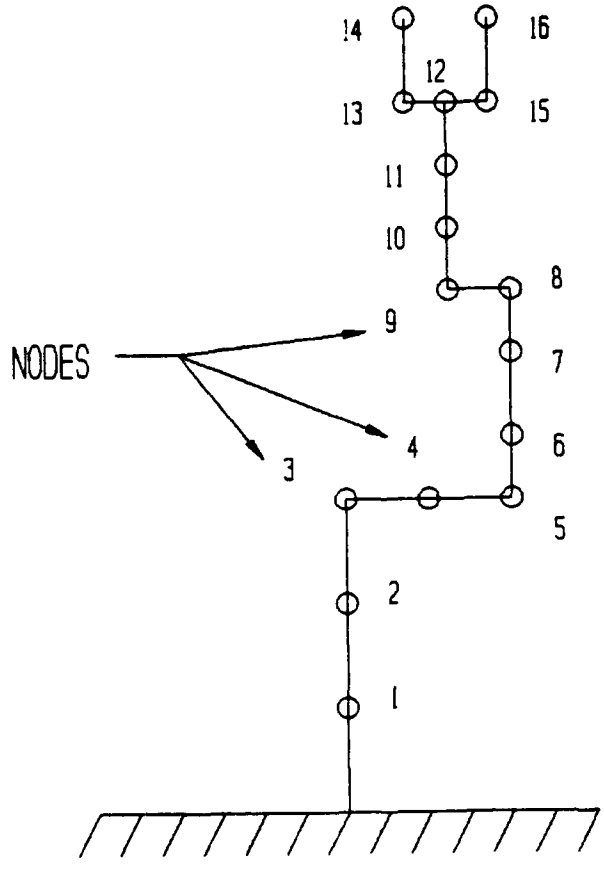


Figure 5.5
Robot Nodes

algorithm [16]. This algorithm first identifies nodes (Figure 5.5) which are located along the individual arms of the robots starting from the base and continuing up to the ends of the gripper parts. During the motion, the distance is calculated between the nodes of the first robot and the second robot, as well as the distance between the first robot and the obstacles. Then, the distances are calculated between the second robot and the obstacles. If any of these distances fall below a "safe" value, permission to perform that movement (as described in the following paragraphs) is not granted.

The obvious problem is that of computation speed due to the large number of nodes needed to describe the robots and the obstacles. If each robot has twenty nodes and two obstacles have ten nodes each, a total of twelve hundred distances will need to be calculated for each movement. This could computationally become very cumbersome. In trying to decrease the number of calculations, several of these distances which will never fall below the safe value can be eliminated. These distances, include for example, the distance between the bases which obviously will not present a problem unless the robots are located on a sliding base track. Even with these reductions, real-time data processing can still only be achieved on a very fast computer.

Now that the collision avoidance algorithm is implemented, a complementary path planning algorithm is introduced in a combination of the Potential Field Concept [10] and the Permission Technique. This combination is then adapted to use with multiple robotic systems.

The name given to this concept, Striving Technique, is derived from the underlying assumption that the priority of movement is geared towards the ultimate goal location as opposed to intermediate through points. This assumption is made for the purpose of allowing the algorithm to be applicable for fields of movement where only local detection of obstacles is known perhaps through sensors or a vision system.

The motion is initiated by first determining a point in space which is situated an incremental distance from the present location and on a straight line between the object's present location and the end point. This next point is then run through the various collision avoidance algorithms to determine if it is achievable. If this location is safe, then the movement is performed and the next location is determined and checked. If the location is not safe, then an alternate location must be found.

The choice of the alternate location is limited to the immediate area and can become very complex for several

reasons. The need for an alternate location was determined due to a possible collision but the nature of the collision is unknown. It could be an overhang that the manipulators must regress from and go over. Or it could be a slot that the manipulator has fed the edge of the obstacle into, which again must be backed away from and jumped. For each individual type of situation, different recovery algorithms would need to be implemented.

Due to this complexity, a simpler general algorithm is chosen to determine the recovery path that the manipulator is to follow. This algorithm handles most situations but does have some difficulties with the most complex of problems. Once the desired next-point is deemed unsafe, the recovery algorithm is called. Here, the computer checks a series of locations to investigate if these locations are safe. The first safe location that is found defines the next location that the object is to move to and at that point, control is passed back to the normal path planning algorithm.

The choice of the appropriate series of locations mentioned above is a function of both the types of situations the manipulator is required to handle, and the speed desired. The speed is increased by having the points positioned with more distance between them. A less number of iterations will be required for points which are farther

apart, but poorer solutions (e.g., longer paths) will be provided than if the points were very close together; which is another trade-off in this process.

The types of situations will also dictate this series of locations. A field where only blocks orthogonal to and on the floor may require the recovery algorithm to go "up and over" the obstacle. It may be valid to state that in certain types of obstacle fields, the manipulators will be able to skirt counter-clockwise around the obstacle. This series thus becomes quite dependant upon the individual task at hand.

The specific recovery algorithm implemented in the computer simulation is set up with the criterion that movement is to occur generally along the x-axis in the negative direction. Thus, the algorithm stipulates that if the desired position is not available, then the next desirable position will be in the positive x-direction. If that position is also not available, a point down from the first alternate position is chosen. Finally, if this second position is also not available, then a position in the positive y-direction is chosen. More of the local alternate positions could be added so there would be no limitations in the types of movements, but there would be extra computation time required. Therefore, in trying to

increase the execution speed of the program, only these three alternate steps are defined.

5.5 Concluding Statements

The specific theory implemented in the computer simulation was discussed at length. Methods for path planning, collision avoidance, and transformation generation were presented which will provide the computer simulation with intelligence to move two robots mutually holding an object with an intelligence to move the object through a field in which obstacles are located. The obstacles may be of regular shape and size or may have overhangs. The object is represented as a rectangular box but may be of any size or shape which allows for a firm grip.

CHAPTER VI

COMPUTER SIMULATION AND GRAPHICS

6.1 Introduction

The previous chapter presented several algorithms and theories which can be utilized in controlling multiple robotic manipulators. These theories included appropriate transformation generation techniques, and path planning and collision avoidance algorithms. The purpose of this chapter is to describe the implementation of these various techniques in a computer program and to describe the actual processes which occur on the computer screen.

The use of computer simulation to analyze the motions of robots is a standard procedure in industry today for several reasons. A manager can review several different types and styles of robots and choose among them to find the one which suits the needs of the particular task at hand. Once the appropriate model is chosen, extensive off-line testing can be performed to identify conflict situations, areas of possible collisions, and the most time or energy efficient paths. Also, if the robot is to fail and collide with a person or object in the work envelope, there could be extensive injury or damage. If in the simulation a collision with an object takes place, lights may flash and buzzers may sound but no actual damage would

occur and nobody would be hurt. The ultimate advantage is that of cost. It is simply less expensive to view computer simulations than it is to actually run a machine in the testing phase of the research and purchase.

The computer simulation "CoordSim" developed for this thesis research is written in TurboPascal, a pascal programming language developed by Borland International for use on a personal computer. This language was chosen for its complete graphics capabilities, ease in program development, and available library of commands. Refer to appendix 1 for the program listing.

6.2 Program Features

CoordSim is set up as a computer generated simulation of two PUMA robots each of which partially share a portion of the other's workspace.

The screen output (figure 6.1) is set up to display most of the pertinent information which the programmer or user needs for analyzing the motion of the robots. The main screen depicts a vector model of the PUMA robots. The individual lines represent vectors which travel along the center-line of the robot arms. The scale of the robots is exactly that of the actual PUMA robot in order to maintain the relative size of the work envelope and the locations of the singularities.

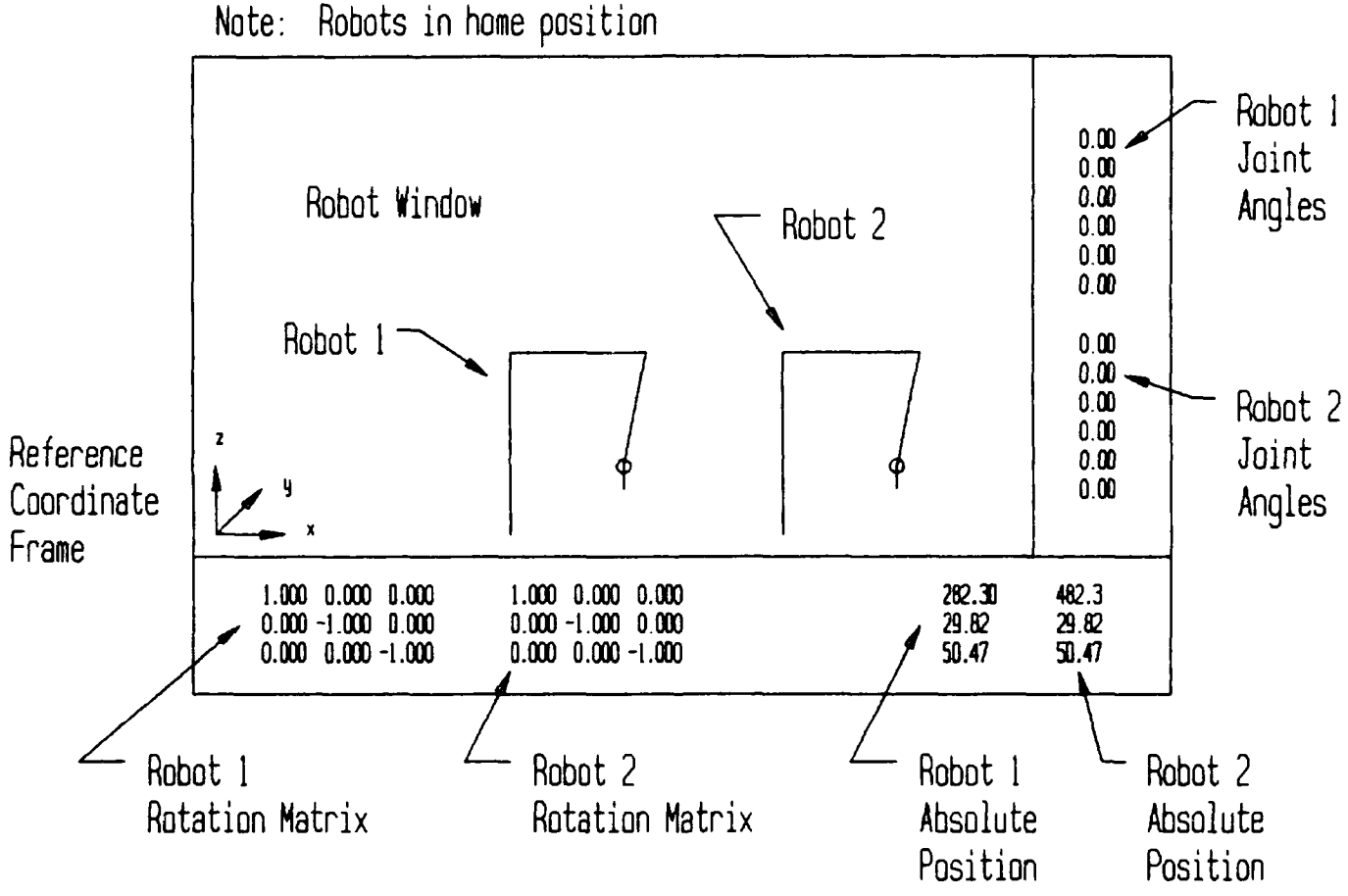


Figure 6.1
Program Output

Around the perimeter of th· screen are several sets of numbers. Starting with the bottom, left hand corner of the screen one finds the rotation matrix of the left hand robot, called Robot 1. The next matrix to the right is the appropriate rotation matrix of the right hand robot, called Robot 2. These rotation matrices describe the orientation of the hand coordinate system in reference to the base coordinate system. In the bottom, right hand corner are the absolute, world frame coordinates of the two robots. The origin of that frame is located in the plane of the screen at the bottom left hand side of the robot window. The user is looking at the x-z plane with the y-axis directed orthogonally into the screen. The final sets of numbers are the joint angles located on the right hand side of the screen. These numbers are output as degrees and represent the values of the joint angles of the six joints of each PUMA robot.

From the keyboard, the user can control several inherent functions of the robots (refer to appendix 2, User's Guide). These functions include increasing or decreasing any of the six joint angles of either robot as well as generating cartesian movement of the end effector frame of either robot. This way, the robots can be placed at any position and orientation in the appropriate work envelopes.

There are several characteristics of real-world robots which were not represented in this simulation. These characteristics are briefly outlined below:

1) There is no limit on the value of the joint angles. In other words, a particular joint can rotate around indefinitely with no effect on the generation of data.

2) No force, acceleration, or vibration considerations are made. The purpose of the simulation presented in this thesis was to test position control strategies and not velocity or force control functions.

3) There is absolute position control. Real world robots have characteristics such as compliance, weight, and inertia which affect the actual position of the robot. Therefore, the position which the robot thinks it occupies is not necessarily the position it actually occupies due to these inaccuracies. The computer generated robot has no weight or compliance. Thus, it can be assumed that the indicated position is completely accurate as obtained from the use

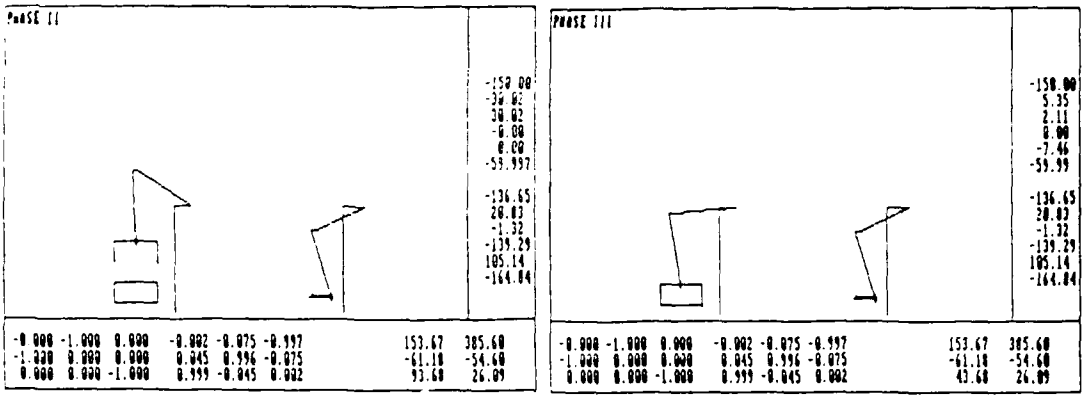
of kinematic and inverse kinematic calculations.

4) Since the speed of computations does not allow for a series of movie-like movements, each frame represents approximately a scaled ten millimeter displacement of the end effector, or at most a ten degree change of a particular joint angle. These values were chosen as they seem to provide a realistic, real-time movement on the screen.

6.3 Utilization of Modeling Techniques

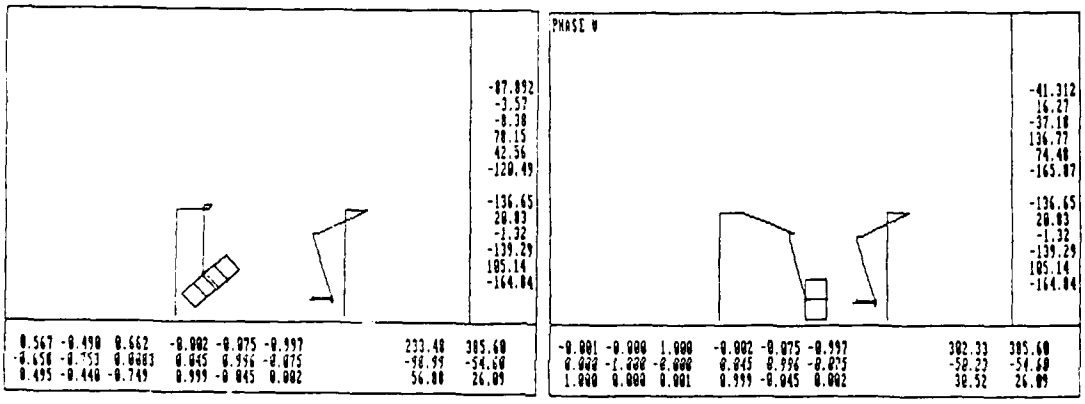
There are eight basic motions which this coordinated robot scheme has been programmed to perform. These functions utilize various algorithms which have been developed in previous chapters and illustrate the generated motion of the manipulators on the computer screen. The purpose of these functions is to index the level of success of these algorithms and test them under various situations. They are presented below in increasing order of complexity.

Routine 1 is a graphic illustration of the capabilities and range of motion of the manipulators (figure 6.2). In this series of movements, the robots are directed to perform a drilling process on a cubical object. Robot 1 first moves to a hover position over the cube while



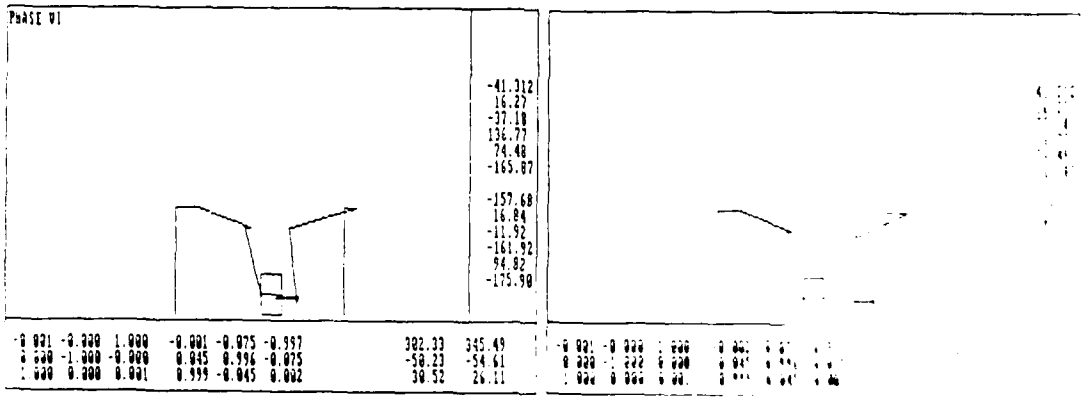
Frame 1

Frame 2



Frame 3

Frame 4

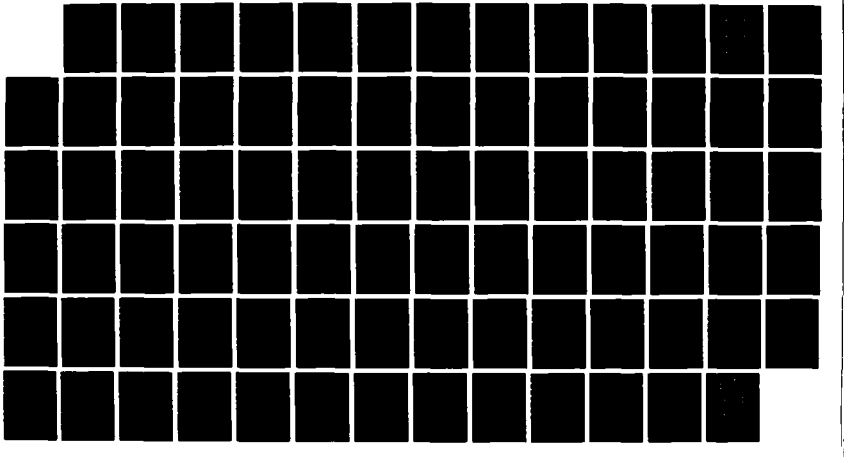


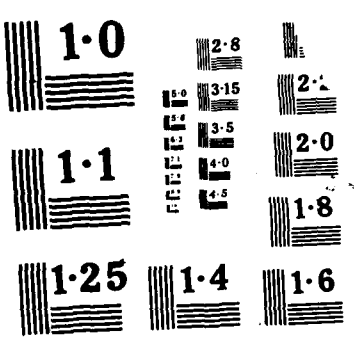
Frame 5

Figure 6.2
Routine 1

AD-A190 069 THE COORDINATION OF MULTIPLE ROBOTIC MANIPULATORS(U) 2/2
ARMY MILITARY PERSONNEL CENTER ALEXANDRIA VA R F YOUNG
11 DEC 87

UNCLASSIFIED F/G 13/8 NL



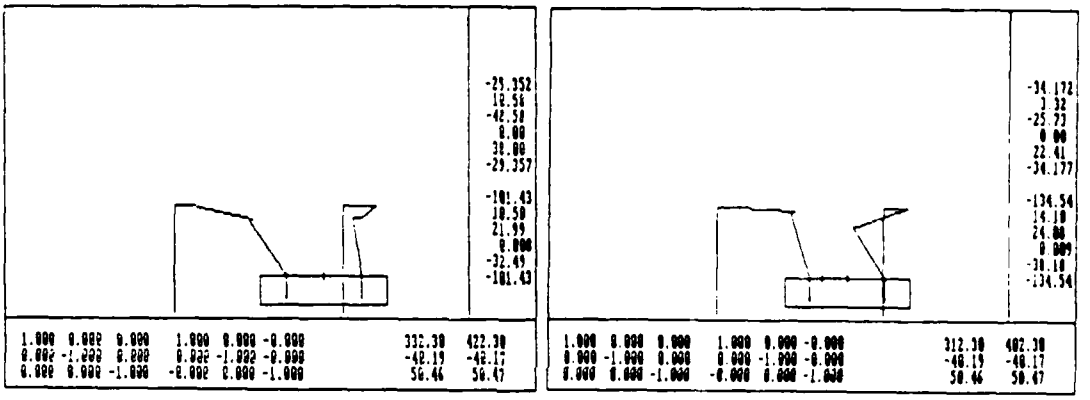

1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
2·0
1·8
1·4
1·6

robot 2 moves to its drilling position and waits (frame 1). Then, robot 1 drops down onto the cube and grasps it (frame 2), and returns to the hover position. Next, robot 1 moves (frame 3) to the vice position where it holds the cube for a drilling operation (frame 4). Robot 2 then drills two holes simultaneously into the cube (frame 5) and retracts the drill (frame 6).

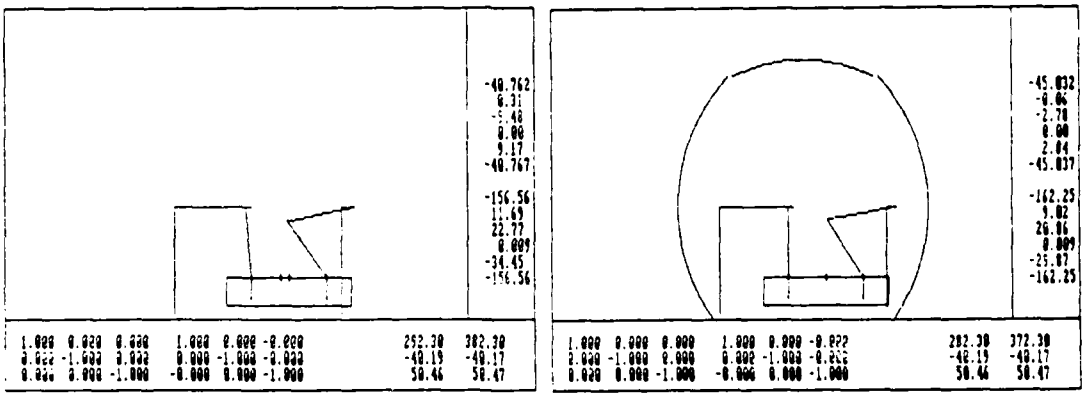
This routine also illustrates the use of coordinated robots where one robot is a tool and the other robot is a smart vice. The vice is able to assume many positions and orientations and thus makes a very versatile machine with which to perform intricate operations.

The execution of this routine utilizes the procedure RobMov as described below. The routine is designed to move the robots from their present location to a new location as defined by the new set of joint angles. Therefore, the motion of routine 1 is generated by specifying a series of points and then moving the robots to those points consecutively via RobMov. In this sense, the routine simulates the use of the teach pendant where each of the robots is taught a series of locations to travel to with the time the robots reach those through-points being closely controlled since the robots are to be coordinated.

Routine 2 introduces the type of coordination where the two robots are simultaneously holding an object (figure 6.3). The purpose of this series of movements is to illustrate the accuracy of the transformation generation theorem as presented by Lim and Chyung [13]. As they showed with their Rhino Robots, routine 2's series of frames indicates a progression of movement composed of a



Frame 1 Frame 2



Frame 3 Frame 4

Figure 6.3
Routine 2

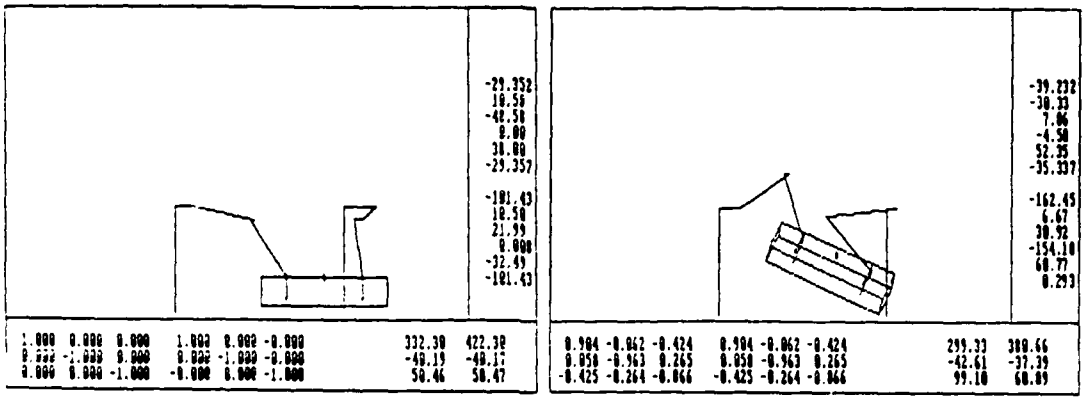
translation, and rotation only about an axis perpendicular to the x-y plane. The start and end points, and orientations are arbitrary values located within the object's work envelope.

This routine also graphically illustrates the object's work envelope as defined in chapter 2. In the last frame of the motion, three arcs are drawn to represent the volume from which the object's coordinate frame origin cannot exit.

The method used in routine 2 simply increments the position vector of $T_{r\bullet}$, then calculates the robots' transformations via NexTran, and finally draws the robots and the object in a loop for however many steps are desired.

Routine 3 represents motion which is extended to include any rotation of the object's coordinate frame. In this routine, the representation of PUMA joint angles is utilized to identify the orientation of the object and to calculate the rotation matrix of $T_{r\bullet}$ (refer to section 5.3 for a detailed discussion). Figure 6.4 illustrates the start and end point of the motion which includes a translation between arbitrary points as well as a change of thirty degrees in theta 5. As can be seen, this rotation is not simply a single change of the roll-pitch-yaw angles

but it effectively performs a combination of two or more of those angles.



Frame 1 Frame 2

Figure 6.4
Routine 3

The basis of this routine is similar to that of routine 2 with the addition of calculating the new rotation matrix of $T_{r\theta}$ while incrementing the position vector. Refer to figure 6.5 for the corresponding flowchart.

Routine 4 (figure 6.6) uses a roll-pitch-yaw angle change instead of the theta angle change in order to generate the orientation matrix of the object's coordinate frame. Here, a thirty degree increase in roll, followed by a twenty degree increase in yaw, were implemented with a translation. Again, the figure shows the start and end points.

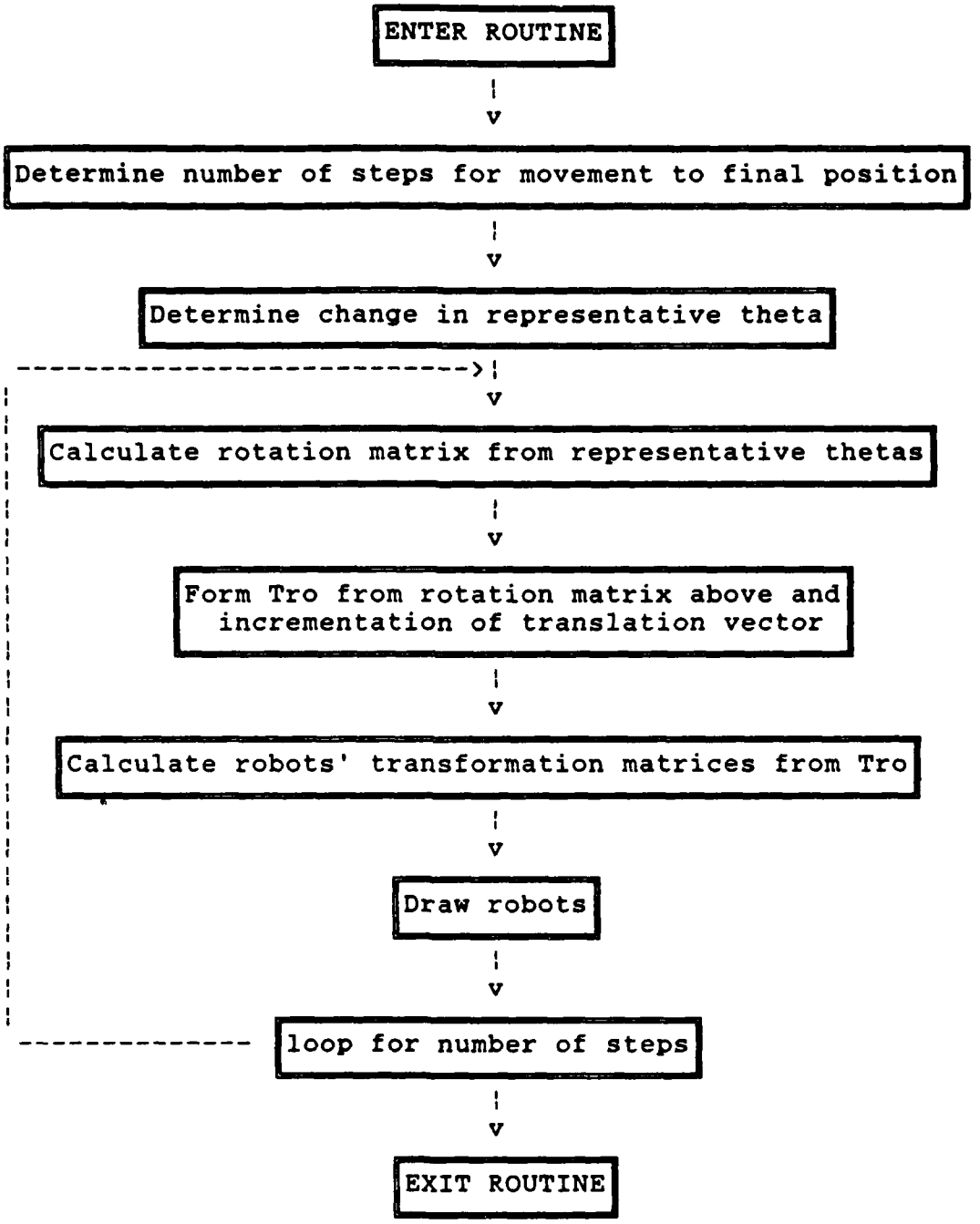
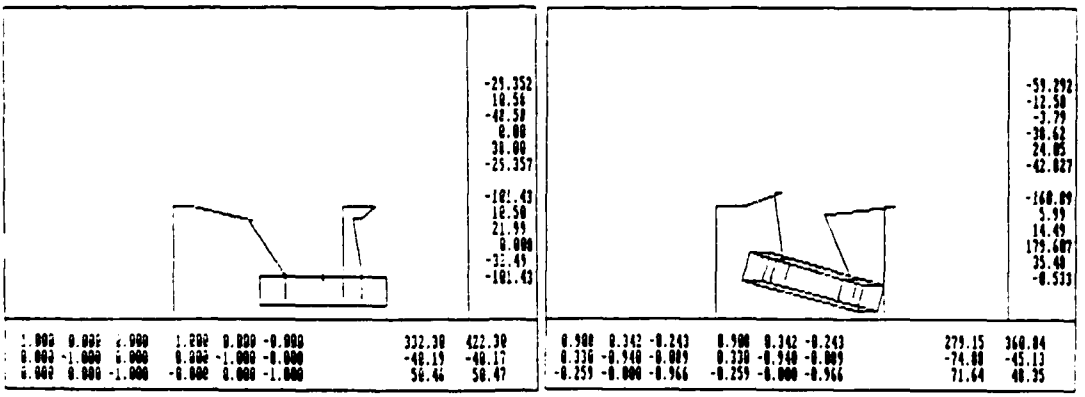


Figure 6.5
Flowchart of Routine 3



Frame 1 Frame 2

Figure 6.6
Routine 4

The procedure used in routine 4 follows the flow chart of routine 3 (figure 6.5) with the exception that instead of using the representative joint angles to calculate the rotation matrix of T_{r_0}, it uses the roll-pitch-yaw representation and equation (5.7).

Routine 5 (figure 6.7) demonstrates the use of the coordinated graph node search algorithm. The movement is similar to routine 2's except at each iteration the graph node search is performed to test whether or not the pre-determined nodes on each of the robot arms are likely to collide. This routine also illustrates the increased computation time required between the frames in order to calculate the distances between the various nodes.

Routine 6 illustrates a potential collision as discovered by the graph node search (figure 6.8). The motion is a translation plus a gross yaw rotation of Π.

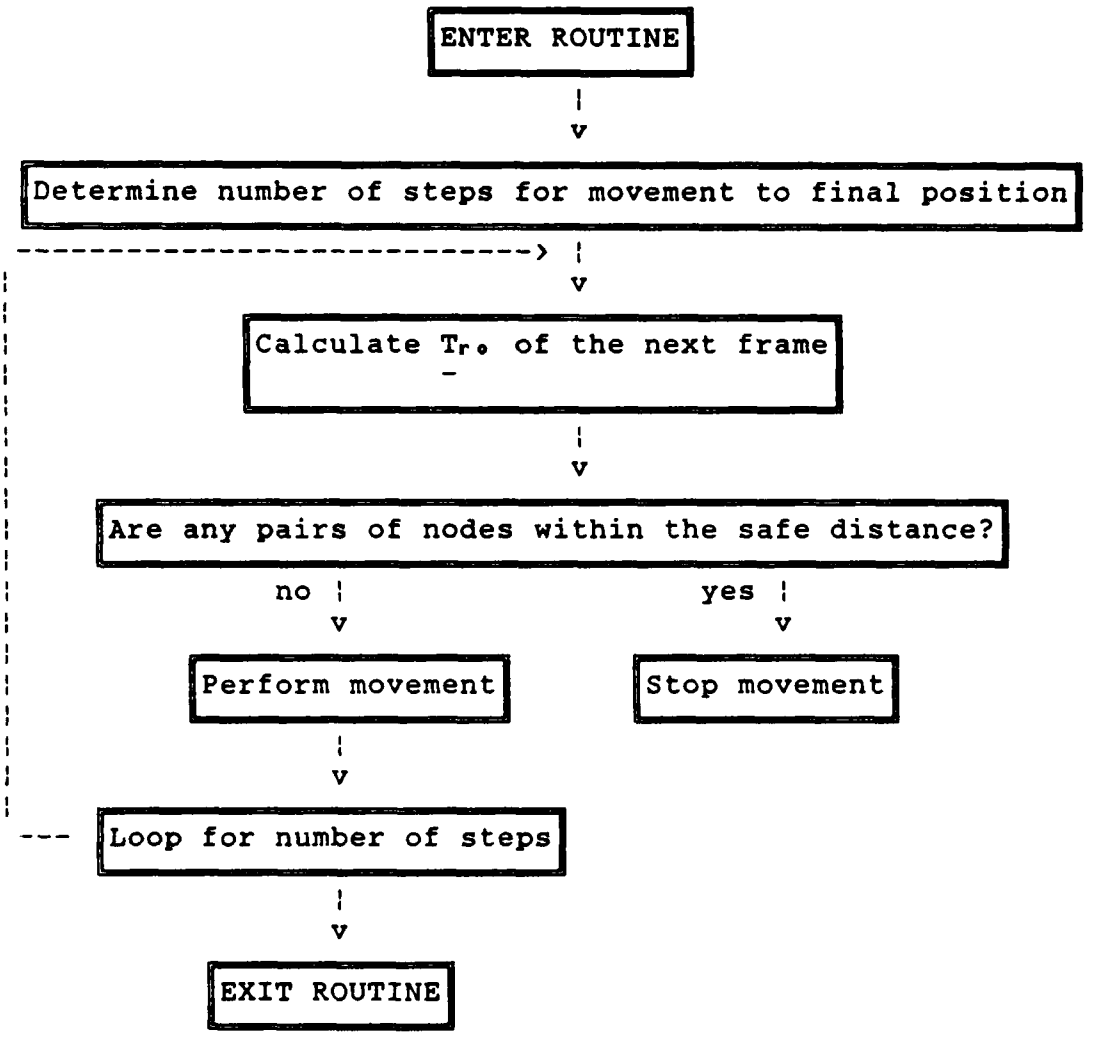
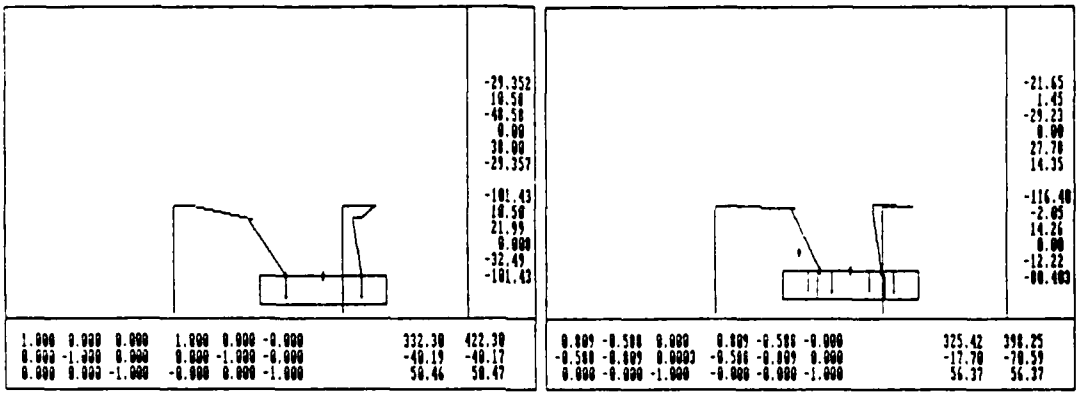


Figure 6.7
Flowchart of Routine 5

This yaw rotation causes the arms of the robots to spiral and twist upon themselves (a twisting collision). At a certain point in the progression of movement, the algorithm

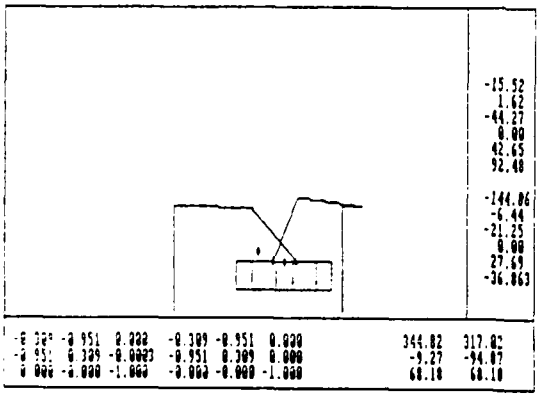
detects the potential collision and causes the movement to halt (frame 4).

Routines 7 and 8 introduce the implementation of the collision avoidance algorithm (refer to figure 6.11 for the flowchart). The difference between the two is the series

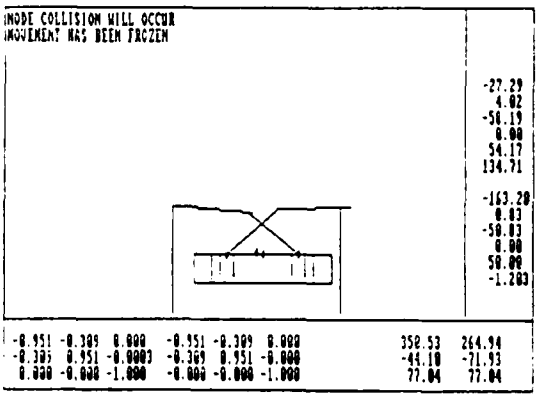


Frame 1

Frame 2

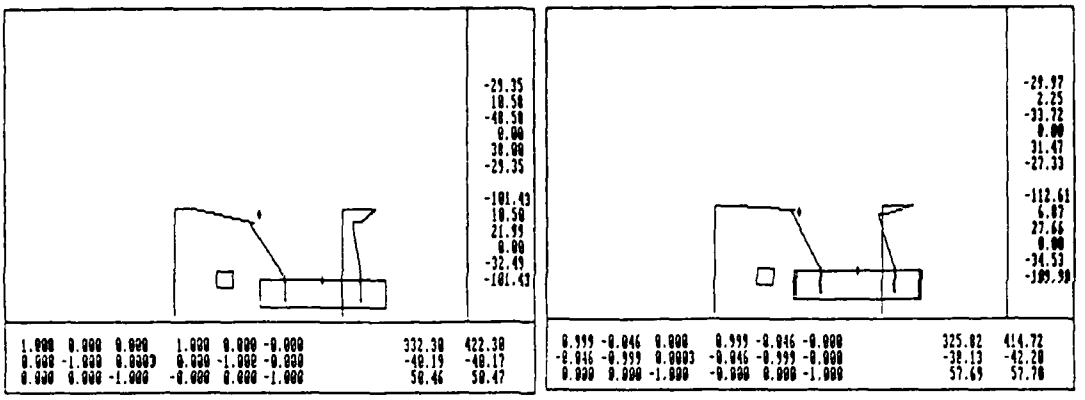


Frame 3



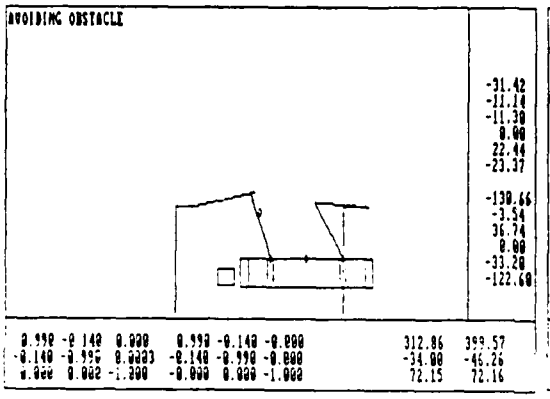
Frame 4

Figure 6.8
Routine 6

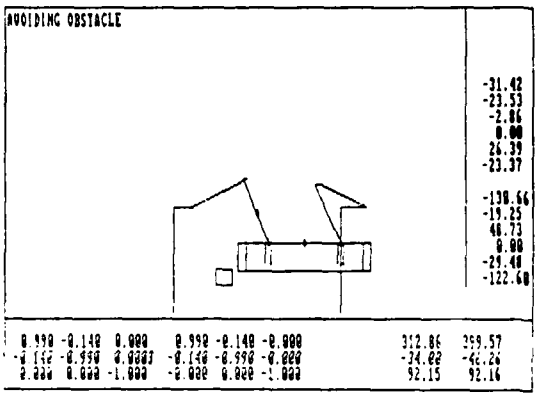


Frame 1

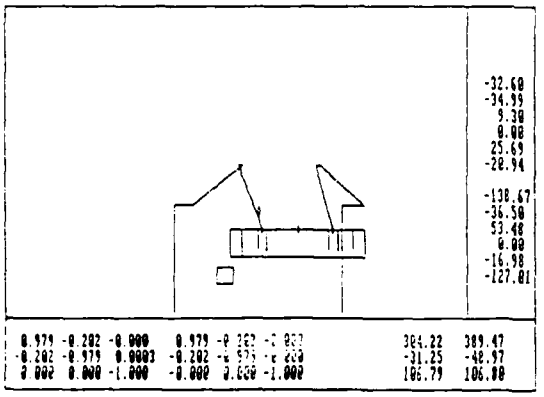
Frame 2



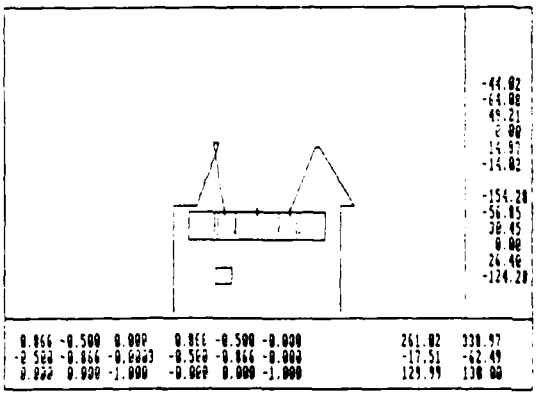
Frame 3



Frame 4



Frame 5

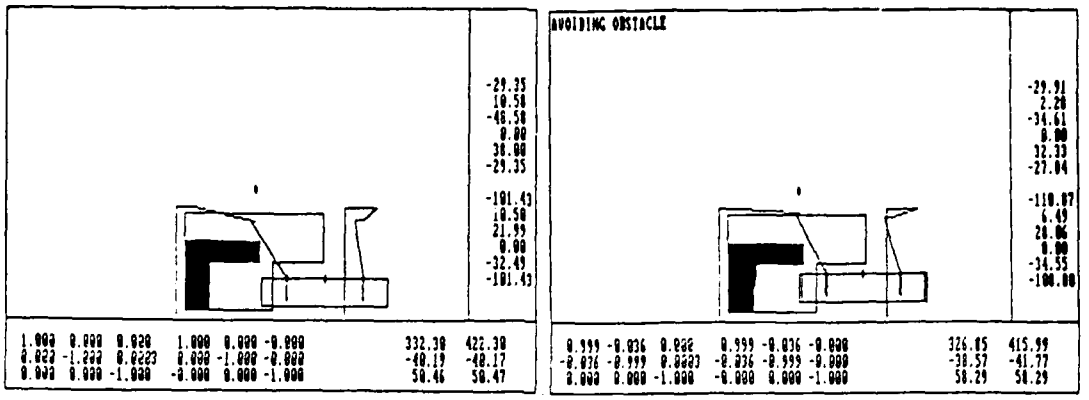


Frame 6

Figure 6.9
Routine 7

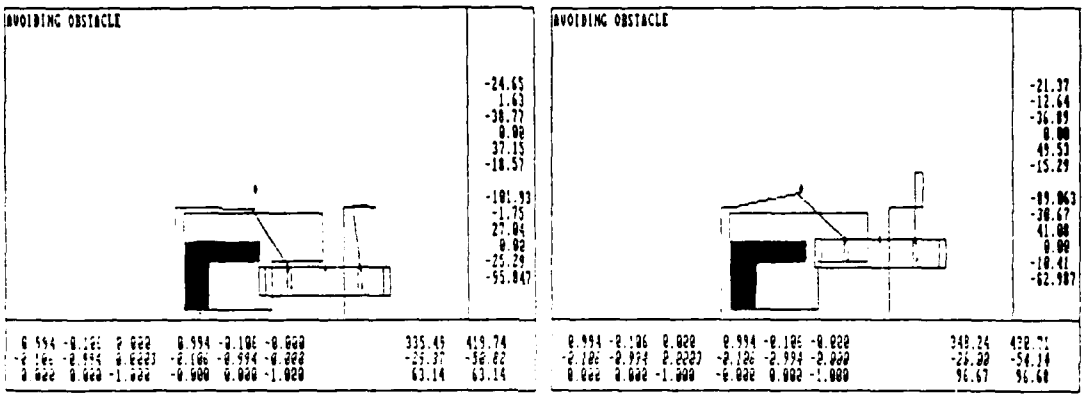
of recovery steps taken once a potential collision is identified. Routine 7 (figure 6.9) shows the object driving towards the goal position (frame 2) until an obstacle is identified (frame 3). It is assumed for this routine that all of the obstacles are box-like in shape and orthogonal to the floor so the recovery routine states that the manipulators are to move up and over the obstacle (frame 4). Therefore, each of the alternate points is simply 10 units upward along the z-axis. Once a clear path is found, the computer regenerates a route to the goal location, and the robot continues on with its motion (frame 5) until the goal is reached (frame 6).

Routine 8 implements a more intelligent recovery routine (figure 6.10) in that the obstacle shape is no longer limited to cubes. Here the possibility of overhanging obstacles is introduced and thus the assumption that simply moving up will provide a solution to the avoidance routine is no longer valid. Frame 1 illustrates the obstacle field. Note that the obstacle size has been enlarged in accordance with Configuration Mapping techniques. The shaded box is the actual obstacle and the box surrounding the obstacle indicates the boundary which cannot be crossed by the object frame's origin. This boundary is not increased uniformly around the obstacle since this motion as defined, only has an object yaw angle



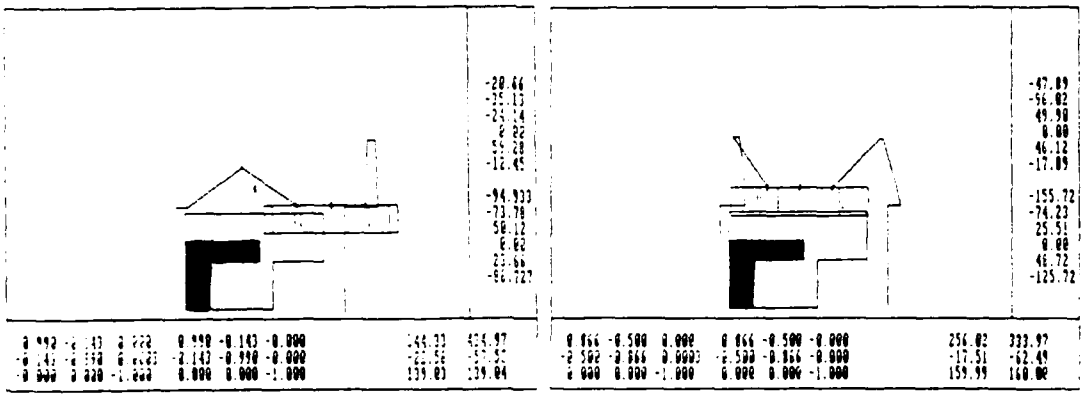
Frame 1

Frame 2



Frame 3

Frame 4



Frame 5

Frame 6

Figure 6.10
Routine 8

change. The purpose of this limitation is to provide a more vivid means to illustrate the output. This does not constrain the algorithm in any way.

The motion initiates (frame 1) and progresses until the next desired position is deemed unobtainable as it is occupied by the obstacle (frame 2). At this point, the computer searches successively to the following points:

1) Alternate Point 1 (AP1) is located 10 units in the negative x-direction and 5 units in the positive z-direction from the unobtainable point.

2) Alternate Point 2 (AP2) is located 15 units in the negative z-direction from AP1.

3) Alternate Point 3 (AP3) is located 10 units in the positive y-direction from AP2.

The alternate points are referenced from the unobtainable location as indicated above due to the fact that theoretically this unobtainable point is closer to the goal position and near the boundary of the obstacle. Thus, it provides a more desirable reference point from which movements may be calculated.

Frame 3 indicates that progression to AP2 provides movement under the obstacle and then frame 4 shows that

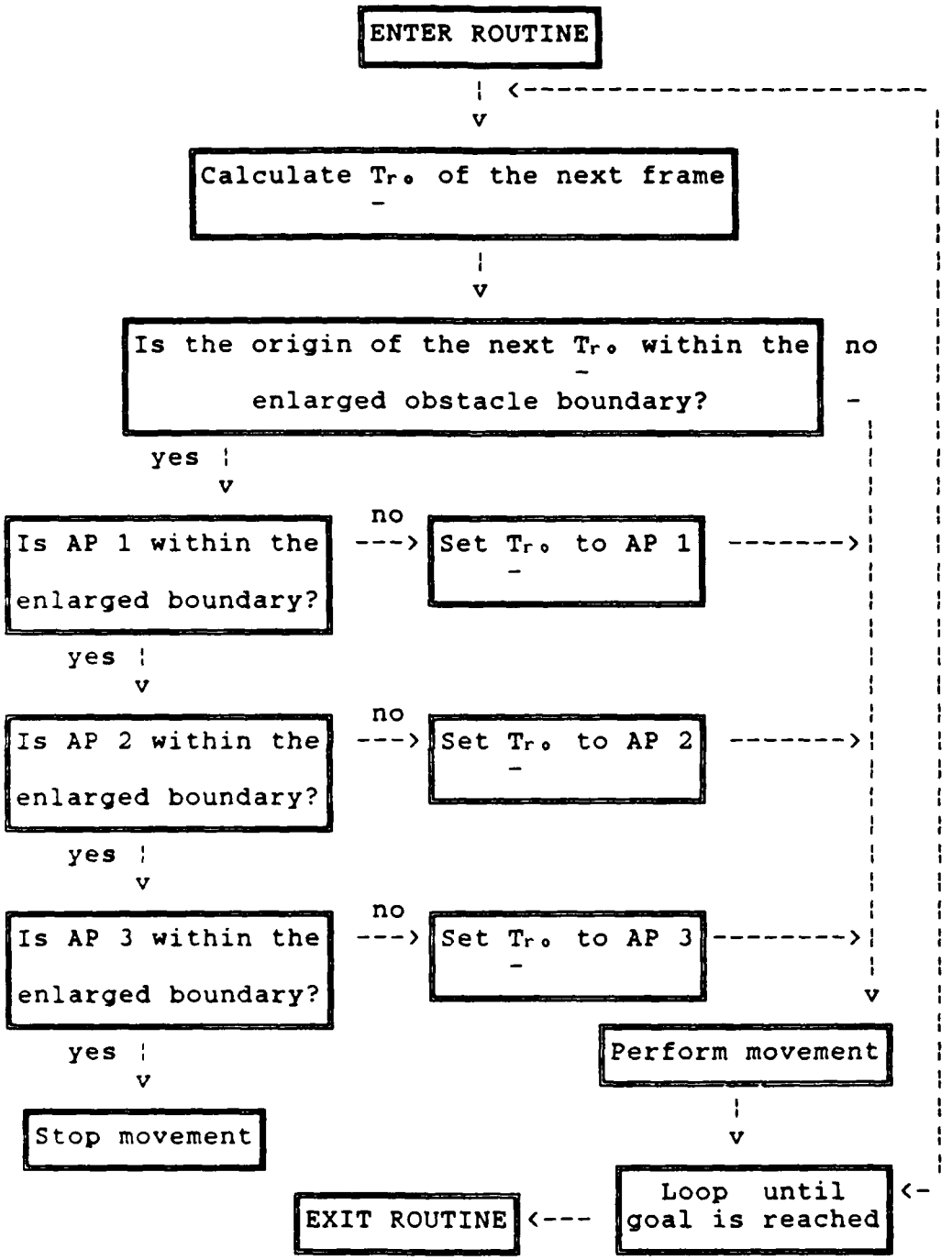


Figure 6.11
Flowchart of Routines 7 and 8

progression to AP1 allows movement up the side of the obstacle. Once the manipulators clear the obstacle (frame 5, the movement is again initiated towards the goal (frame 6).

Note that while the object is in the enlarged obstacle, no collision occurs unless the origin of the object's coordinate from crosses that boundary.

6.4 Subroutine Definitions

The following is a summary of the various functions, procedures (subroutines), and programs utilized by the simulation. Refer to appendix 1 for a complete program listing and variable list.

Function Atan2: TurboPascal does not have a math function for the four-quadrant inverse tangent function. Therefore, this function is written to input the numerator (num) and the denominator (den) as real numbers and returns the Atan2 value as a real number.

Function Pow: Again TurboPascal has no routine for finding a number's power greater than two. This function inputs a real number (x), and the real power (y) and returns a real which is x raised to the y power.

Procedure Envelope: This routine draws the object's work envelope onto the screen.

File Graph.p: This file is compiled with CoordSim in order to provide several TurboPascal graphics functions such as drawing lines (Draw) or filling an enclosed shape with a color (FillShape).

Procedure Coordinate: This routine inputs the joint angles (theta) and the transformation matrices (trans) of both robots and returns the world coordinates of the nodes on each of the robots (coord). The nodes are defined as in figure 5.4

Procedure Node: This routine inputs the reference coordinates of the nodes (coord) as found in procedure Coordinate and returns a Boolean value (collision) which when true, indicates that a node on robot 1 is coming close to contact with a node on robot 2. This routine is used in the graph node search algorithms.

Procedure CalcRot: This routine inputs the roll-pitch-yaw angles (rpy) and returns the appropriate rotation matrix (rot).

Procedure CalcTran: This routine inputs the joint angles of both robots (theta) and returns the appropriate transformation matrices of each of the robots (trans).

Procedure Step: This routine inputs a start and end position of an end-effector (start, finish) and returns an

integer which represents the approximate number of ten unit steps that there are between the start point and end point. The purpose is to have a uniform, ten step movement on the screen when a robot is to move.

Procedure InvKin: This routine inputs the transformation matrices and characteristics of both robots (trans, chars), performs inverse kinematic calculations, and then returns the joint angles (theta). The characteristics of a PUMA robot describe the arm up/down, wrist flip/no-flip, and shoulder left/right situations.

Procedure Border: This routine draws a border around the screen and divides the robot window from the numbers which are displayed on the screen.

Procedure DrawIquad: The graphics functions of TurboPascal reference the upper left hand corner of the screen as the origin with positive x values increasing to the right and positive y values increasing downward. This routine transforms the origin to the lower left hand corner of the screen and emulates the first quadrant of an x-y coordinate system. Also, this graphics screen is 640 pixels x 200 pixels; therefore, a point (100,100) will not be located equidistant from the lower and left hand side of the screen. This routine also numerically compensates to make the scale on the x-axis equal the scale on the y-axis

and produce a graphically correct picture for the user to see.

Procedure CircleIquad: This routine draws a circle on the screen in the coordinate system described above.

Procedure DrawFrame: This routine inputs the transformation matrices, joint angles, and node coordinates (trans, theta, coords) and draws the robots on the screen.

Procedure MulTran: This routine multiplies two four by four matrices (tran1, tran2) and returns the product (prod).

Procedure Nextran: This routine inputs the transformation of the mutually held object (Tro) and returns the two robots' transformations (trans).

Procedure InvTran: This routine inputs a transformation matrix (tran) and returns the inverse of that matrix (invtran).

Procedure DrawBox, DrawSquare, and DrawBox2: These routines draw the various objects which are being held by the robots. It also identifies the origin of the object's coordinate system with a circle.

Procedure ThetaIncr: This routine is similar to Steps except that the number of steps is a function of the largest change among the joint angles.

Procedures DrawObst and DrawObst1: These routines draw various obstacle which are used by the main program.

Procedure Robmov: This routine moves a robot automatically from its present location as identified by joint angles (theta) to a new location as identified by the new joint angles (thetaNEW) in a uniform motion.

With the definition of these various procedures contained within CoordSim, and the routines described in this chapter, the implementation of the algorithms in the code should be apparent.

6.5 Concluding Statements

This chapter has described the computer program CoordSim. Included in that description is an explanation of the purposes of the various procedures and routines located within the code and the manner in which the algorithms developed in past chapters have been implemented by the computer simulation.

The program has been tested under numerous situations and has proved successful under all of those tests. Thus, a complete, and viable simulation tool has been developed.

The following chapter discusses the level of success which was achieved by each of the routines as well as the theory it supports.

CHAPTER VII

RESULTS AND CONCLUSIONS

7.1 Discussion of Results

There are several problems present in industry today which at a first glance seem to be easily solvable with the use of robotic manipulators. However, due to limitations such as maximum payload, maximum reach of the individual robots, and the single robot work-cell's dynamic inadaptability, the use of a single robot can prove to be ineffective in certain work environments. The following section presents some of these situations and their possible solutions as derived using the various algorithms and theories contained in this thesis.

There are many examples of processes which require the manipulation of large, oblong, or heavy objects. Imagine attempting to pick up a 10 foot, 200 pound bar by one of the two ends--not an easy task by any means for one robot to perform due to the tremendous torque produced by the weight. If the robot had a maximum payload constraint of 150 pounds, then even trying to lift the bar in the center would prove impossible. This problem is easily solved through the use of two robots, each grasping the bar at an end. Assuming the bar to be a rigid structure, the moment at the robot hand disappears and a single robot need only

lift one-half the total weight of the bar. In this situation, the two robots with maximum payloads of 150 pounds each, now effectively have a 300 pound lifting capability which would be enough to lift the 200 pound shaft.

Also, with a two robot work-cell, one of the robots can function as a smart vice while the other performs some sort of operation on the held piece. The advantages of this type of work-cell are numerous. The smart vice can attain many more positions and orientations compared to a vice developed for a specific purpose. Moreover, the smart vice can easily be adapted for new processes which need to be implemented.

The uses of multiple robot systems are many. The problem is to control such systems easily and accurately. Routine 2 (chapter 6) presents and illustrates the successful implementation of a coordinated position control scheme. This routine simulates a process where, for example, a long, heavy shaft is to be loaded into a lathe. The robots are able to pick up the shaft and move it from a start point (perhaps a stock bin) to an end point (the lathe) by simply controlling the motion of a coordinate system located on the bar.

Many processes on assembly lines utilize single robots which have been "taught" a function. Teaching is a process whereby the robot is led through a motion by a human operator. The operator records joint velocities, through points, and other function-specific characteristics, and the robot cycles repeatedly through the motion to perform the task at hand (such as an assembly). Using the position control scheme proposed in this thesis, the same teaching process can be implemented to teach coordinated robots. In this case, the operator references the mutually held object's coordinate frame to define the motion of the system. If a bar is to be loaded into a lathe, the operator need only teach the path for the bar to take and the computer automatically calculates the joint angles and velocities given the bar path information. Routine 4 illustrates this scenario. The bars path is defined by a start point (a stock bin) and an end point (the lathe). The computer then takes the bar through this displacement with an appropriate change in the roll-pitch-yaw angle for proper loading into the lathe.

With the algorithms presented in this thesis, not only is it physically possible to lift this bar and control its motion via a teach pendant, but the bar can also be moved intelligently. This means that the two robots holding the bar can be moved from a starting location and orientation

to an end point and orientation through a computer generated path which accounts for obstacles located in the work envelope. This scenario is futuristic as it represents a problem of the work-cells of tomorrow. In this highly dynamic work-cell, the processes can be extremely complex. Thus, it may become quite ineffective to teach each individual motion for the robots to perform. The work-cell must be able to adapt to changing environments and perform functions given only a set of parameters. The cell's controlling computer should intelligently ponder the problem at hand, decide upon appropriate actions, and implement those actions accordingly.

The following is presented as an example scenario. The process controller has decided upon the task of this particular work-cell. This task is to move a long rod from a centralized stock bin to a cutting machine. The cutting machine is located opposite the stock bin with a pathway for Automatically Guided Vehicles (AGV's) running between the two. Thus, the robot's path contains obstacles. The computer must therefore control the coordinated robots through the motion and avoid the AGV's as they move along the pathway. Routine 8 (chapter 6) illustrates this type of movement. The robots successfully navigate from the start point (stock bin) to the end point (cutter) with the

avoidance of obstacles located in the field occurring dynamically with the motion.

Another situation may arise where the bar to be loaded into the cutter has been inadvertently placed into the stock bin backwards. The robot senses this mistake and tries to correct it by rotating the bar around so it can be loaded correctly. Unfortunately, this motion is impossible since the arms of the robots will twist upon themselves to the point of collision. Routine 6 (chapter 6) illustrates the detection of this type of problem through a graph node search scheme and prevents the robots from damaging themselves. With these algorithms, new motions can be introduced in the work-cell without any danger of collision as the computer contains the intelligence to anticipate and prevent various types of collisions from occurring.

There are many situations where processes are feasible only as completed using multiple robotic systems. This section presented some of those processes and the means by which the concepts presented in this thesis can be implemented to achieve those processes.

In summary, following are the major contributions of this thesis' research:

1) The resolved position control theory presented by Lim and Chyung [13] which was

implemented on five degree-of-freedom Rhino robots is proven to be valid and effective for six degree-of freedom PUMA robots.

2) Through contribution 1, the coordinated robots mutually holding an object are able to attain any position and orientation within the object's work envelope.

3) The object's work envelope is defined as the locus of points obtainable by the origin of the object coordinate frame. The method by which the work envelope can be derived is also documented.

4) The Artificial Potential Field Concept and Configuration Mapping are combined and altered to provide a collision avoidance algorithm for the coordinated robot scheme.

5) The problem of Twisting Collision is defined and potential solutions are presented.

6) The "Striving Technique" is introduced as a combination of collision avoidance and path planning algorithms. This technique provides intelligence for the coordinated

robots when moving from a start position and orientation to a final position and orientation through a field with obstacles.

7) The computer simulation CoordSim is presented to graphically illustrate the motions and implementations of the various algorithms.

7.2 Conclusions

This thesis focuses upon the feasibility, adaptability, and controllability of coordinated robotic systems. The discussion starts in chapter 2 with a literature review of the background of coordination of robots. Chapters 3 and 4 are concerned with a variety of path planning and collision avoidance algorithms and methods whereby single robot algorithms can be adapted to multiple robot systems. Chapter 5 sets up the modeling techniques and specific algorithms which are represented in the computer simulation. Finally, chapter 6 reviews the computer simulation--CoordSim--and discusses the levels of implementation of this software.

The "Striving Technique" is introduced as a means whereby a computer simulates an intelligence in order to move two PUMA robots, simultaneously holding a bar, through a field having obstacles. The computer is able to direct

the robots to avoid obstacles of any size or shape and to move between any two positions and orientations. In addition, the problem of twisting collision is addressed and solved through an adaptation of the Graph Node Search algorithm. Finally, the Coordinated Work Envelope is defined and a method is proposed to aid in the calculation of that work envelope.

7.3 Suggestions for Future Work

There are at least three major areas which would benefit from an extension of results presented in this thesis. The first would be to implement the various algorithms on real machines versus simulating their functions on a computer. The advantages of this are many. Variables such as inertia, friction, vibration, and timing would not be ignored and as they are real-life problems, they could be addressed, tested, and identified. Also, the use of real machines would unequivocally demonstrate the application of the proposed position control algorithms and their implementation towards real-world problems.

Secondly, a type of motion which presents itself as a complex problem is that of "snaking". This is a motion where the bar must weave in between several obstacles located in the field of movement. In other words, implementation of a simple translation in order to avoid

collision would not be adequate. A rotation would also be needed. Research into this type of motion would greatly increase the capabilities of the motion itself due to its great increase in flexibility.

Finally, it becomes obvious that recovery algorithms designed to solve specific problems soon become cumbersome as there are many situations which could be presented to a system. While they may be very effective for specific tasks where there is a limit on the motions possible, they prove invalid for the types of systems where complete independence is desired. Therefore, it is believed that the direction for future work should concentrate more on a general solution which is job-independent. In this way, any conceivable motion or process could be solved. Obviously, this type of general solution is a major hurdle to be overcome and will require much ingenuity, time, and effort to arrive at potential solutions.

REFERENCES

[1] Morris, H. "Controlling Multiple Robot Arms", Control Engineering, (9/86), pp. 144-147.

[2] Skirkhodaie, A., Taban, S., and Soni, A. "AI Assisted Multi-Arm Robotics", IEEE International Conference on Robots and Automation, (3/87), v. 3, pp. 1672-1676.

[3] Maimon, O. and Nof, S. "Analysis of Multi-Robot Systems", IIE Transactions, (9/86), v. 18, n. 3, pp. 226-234.

[4] Saigo, M. and Sadao, F. "Coordinating a Dual-Arm Assembly Robot", Robotics Engineering, (11/86), v. 8, n. 11, pp. 8-12.

[5] Hemami, A. "Control and Programming of a Two-Arm Robot", Technical Paper (Society of Manufacturing Engineers), (6/85), paper ms 85-600, pp. 2565-2586.

[6] Maimon, O. and Nof, S. "Coordination of Robots Sharing Assembly Tasks", Transactions of the ASME, (12/85), v. 107, n. 4, pp. 299-307.

[7] Callan, J. "The Simulation and Programming of Multiple-Arm Robot Systems", Robotics Engineering, (4/86), v. 8, n. 4, pp. 26-29.

[8] Red, W. and Cao, H. "Configuration Maps for Robot Path Planning in Two Dimensions", Transactions of the ASME, (12/85), v. 107, n. 4, pp. 292-298.

[9] Kokaji, S. "Collision-Free Control of a Manipulator with a Controller Composed of Sixty-Four Microprocessors", IEEE Control Systems Magazine, (10/86), v. 6, n. 5, pp. 9-14.

[10] Khatib, O. "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots", The International Journal of Robotics Research, (spring/86), v. 5, n. 1, pp. 90-98.

[11] Nageswara, S., Iyengar, S., Jorgensen, C., and Weisbin, C. "Robot Navigation in an Unexplored Terrain", Journal of Robotic Systems, (winter/86), v. 3, n. 4, pp. 389-407.

[12] Kambhampati, S. and Davis, L. "Multiresolution Path Planning for Mobile Robots", IEEE Journal of Robotics and Automation, (9/86), v. RA-2, n. 3, pp. 135-145.

[13] Lim, J. and Chyung, D. "Resolved Position Control for Two Cooperating Robot Arms", Robotica, (3/87), v. 5, part 1, pp. 9-15.

[14] Gilbert, E. and Johnson, D. "Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles", IEEE Journal of Robotics and Automation, (3/85), v. RA-1, n. 1, pp. 21-30.

[15] Lumelsky, V. "Dynamic Path Planning for a Planar Articulated Robot Arm Moving Amidst Unknown Obstacles", Automatica, (2/87), v. 23, n. 5, pp. 551-570.

[16] Zapata, R., Fournier, A., and Dauchez, P. "True Cooperation of Robots in Multi-Arms Tasks", IEEE International Conference of Robotics and Automation, (3/87), v. 3, pp. 1255-1260.

[17] Craig, J. "Introduction to Robotics, Mechanics, and Control", Addison-Wesley Publishing Company, Inc, (1986), p. 41.

[18] Chimes, P. "Multiple-Arm Robot Control Systems", Robotics Age, (10/85), pp. 5-10.

[19] Hawker, R., Nagel, R., Roberts, R., and Odrey, N. "Multiple Robotic Manipulators", Byte, (1/86), pp. 203-219.

[20] Lee, C. "Robot Arm Kinematics", in Tutorial on Robotics: IEEE Computer Society Press, (1/83), pp. 45-72.

[21] Grossman, D., Evans, R., and Summers, P. "The Value of Multiple Independent Robot Arms", Robotics and Computer-Integrated Manufacturing, (8/85), v. 2, n. 2, pp. 135-142.

[22] Lee, B. and Lee, C. "Collision-Free Motion Planning of Two Robots", IEEE Transactions on Systems, Man, and Cybernetics, (2/87), v. smc-17, n. 1, pp. 21-32.

[23] Acker, F. and Ince, I. "Troikabot--A Multi-Armed Assembly Robot", Tecnical Paper (Society of Manufacturing Engineers), (6/85), paper ms85-589, pp. 2357-2376.

[24] Maimon, O. "A Generic Multirobot Control Experimental System", Journal of Robotic Systems, (winter/86), v. 3, n. 4, pp. 451-466.

[25] Mayer, G. and Wood, E. "Multiple-Arm Control and Assembly Operation", Robotics Engineering, (4/86), v. 8, n. 4, pp. 18-25.

[26] Wong, E. and Fu, K. "A Hierarchical Orthogonal Space Approach to Three-Dimensional Path Planning", IEEE Journal of Robotics and Automation, (3/86), v. RA-2, n. 1, pp. 42-53.

APPENDIX 1

PROGRAM LISTING

```
Program CoordSim;

        { This program is a simulation of coordinated PUMA
          robot arms }

  {$C-}  { allows for keypressed function }

{****************************}

        { include TurboPascal graphics functions }

  procedure Graphics;
          external 'GRAPH.BIN';
  procedure HiRes;
          external Graphics[6];
  procedure HiResColor(Color: Integer);
          external Graphics[9];
  procedure Palette(N: Integer);
          external Graphics[12];
  procedure GraphBackground(Color: Integer);
          external Graphics[15];
  procedure GraphWindow(X1,Y1,X2,Y2: Integer);
          external Graphics[18];
  procedure Plot(X,Y,Color: Integer);
          external Graphics[21];
  procedure Draw(X1,Y1,X2,Y2,Color: Integer);
          external Graphics[24];
  procedure Circle(X,Y,Radius,Color: Integer);
          external Graphics[33];
  procedure FillShape(x,y,fillcol,bordercol: integer);
          external Graphics[48];
  procedure ClearScreen;
          external Graphics[60];

{****************************}

  label
    FLAG1, FLAG2, FLAG3, FLAG4, FLAG5, FLAG6, FLAG7, FLAG8;

  type  { identify types of matricies }
    matrix2x4x4 = array [1..2,1..4,1..4] of real;
    vector2x6 = array [1..2,1..6] of real;
    vector2x3 = array [1..2,1..3] of real;
    vector3 = array [1..3] of real;
    vector2 = array [1..2] of real;
    matrix3x3 = array [1..3,1..3] of real;
    matrix6x3 = array [1..6,1..3] of real;
    matrix2x16x3 = array [1..2,1..16,(x,y,z)] of real;
    matrix4x4 = array [1..4,1..4] of real;

  var  { define variables }
```

```
    del_rpy,          { change in rpy angle }
    rpy,              { rpy angles }
    del               { change in position }
                : vector3;
    del_theta         { change in theta }
                : array [1..6] of real;
    trans,            { transformation matrix from robot
                        base to hand }
    Trb,              { trans from reference to base }
    Tinvrb,           { inverse trans from ref to base }
    NexTrans,         { Next location's trans }
    Transinit,        { initial trans from base to hand }
    Tro,              { trans from ref to object }
    TroEND,           { final trans from reference
                        to object }
    Tinvinitro,       { inv, initial trans from ref
                        to object }
    T2,               { intermediate, dummy trans }
    Tincr,            { a dummy var for drawbox }
    RotTro1           { rotation matrix of Tro }
                : matrix2x4x4;
    RotTro            { rotation matrix of Tro }
                : matrix3x3;
    theta             { joint angle vector }
    thetaTRO,         { thetas which generate Tro }
    thetaNEW          { new joint angle vector }
                : vector2x6;
    coord             { node coords matrix }
                : matrix2x16x3;
    chars             { shoulder, arm and wrist char.'s }
                : vector2x3;
    i,j,k,l,m,n,      { counters }
    numsteps,         { number of steps between start
                        and finish }
    RobotID           { identifies between robots }
                : integer;
    movementx,        { change in x motion }
    movementy,        { change in y motion }
    movementz,        { change in z motion }
    px,py,pz          { position coordinates }
                : real;
    inkey             { key read from keyboard input }
                : char;
    grip,             { tells if robot has grip on object }
    collision         { toggle for graph node search exec. }
                : boolean;


{*****************************}

  FUNCTION Atan2 (num, den : real) : real; begin
```

```
        { This program inputs the numerator and
          demomenator of an inv. tan function and
          returns the atan2 function. }

    if (num = 0.0000) and (den = 0.0000) then
      write('zeros in atan2 function')

        { Y-axis }
    else if (den = 0.0) and (num > 0.0) then atan2 := pi/2

    else if (den = 0.0) and (num < 0.0) then atan2 := -pi/2

        { Vector close to pos x-axis.  Adjust 0.0001 to
          vector length }
    else if (num < 0.0001) and (num > -0.0001) and
            (den >= -1.0) then
      atan2 := 0.0

        { Vector close to neg x-axis }
    else if (num < 0.0001) and (num > -0.0001) and
               (den < -1.0) then
      atan2 := pi

        { first quad }
    else if (num > 0.0) and (den > 0.0) then
      atan2 := arctan(num/den)

        { second quad }
    else if (num > 0.0) and (den < 0.0) then
      atan2 := pi - arctan(abs(num/den))

        { fourth quad }
    else if (num < 0.0) and (den > 0.0) then
      atan2 := arctan(num/den)

        { third quad }
    else if (num < 0.0) and (den < 0.0) then
      atan2 := -pi + arctan(num/den)

    else write('** error in atan2 function **');
  end;  { routine }

{*****************************}

  FUNCTION Pow(x, y : real) : real ; begin

        { This function returns a number to a power }

    if (x > 0.0) then
      pow := exp(y*ln(abs(x)))
```

```
    else if (x = 0.0) then
      pow := 0.0
    else if (frac(abs(y)) = 0.0) then begin
      if (odd(abs(round(y)))) then
        pow := -exp(y*ln(abs(x)))
      else
        pow := exp(y*ln(abs(x)));
      end  { else }
      else
        write('bad numbers in pow routine');
  end; { function }

{*****************************}

  PROCEDURE Envelope;

          {  This routine calulates the work envelope and
             draws it }

    var
      x, y : integer;

    begin { routine }
      for y := 0 to 300 do begin
        x := round (400-sqrt(sqr(250.0)-sqr(y-137.7)));
        Plot (x, round (153-0.4*y),1);
        x := round (200+sqrt(sqr(250.0)-sqr(y-137.0)));
        Plot (x, round (153-0.4*y),1);
      end;  { for y }

      for x := 215 to 385 do begin
        y := round(137.0+sqrt(sqr(183.0)-sqr(x-300.0)));
        Plot(x, round(153-(0.4*y)),1);
      end;  { for y }
    end;  { routine }

{*****************************}

  PROCEDURE Coordinate (var coord : matrix2x16x3;
                        trans : matrix2x4x4;
                        theta : vector2x6);

        { This routine calculates the intermediate
          coordinates of the nodes }

    const
      scale = 0.20;  { scale factor between true robot
                       values in mm and screen scale }
      scale2 = 25.0;      { scale factor between rotation
                            vectors and hand size }
```

```
var i :
  integer;

begin  { routine }

  coord [1][1][x] := 200.0; { robot base coordinates }
  coord [1][1][y] := 0.0;
  coord [1][1][z] := 5.0;
  coord [2][1][x] := 400.0;
  coord [2][1][y] := 0.0;
  coord [2][1][z] := 5.0;

  for i := 1 to 2 do begin

    coord [i][2][x] := coord [i][1][x];
    coord [i][2][y] := coord [i][1][y];
    coord [i][2][z] := (scale*660.4) + coord [i][1][z];

    coord [i][3][x] := coord [i][2][x] -
                       sin (theta [i][1])*
                       scale*200.0;
    coord [i][3][y] := coord [i][2][y] +
                       cos (theta [i][1])*scale*200;
    coord [i][3][z] := coord [i][2][z];

    coord [i][4][x] := coord [i][3][x] +
                       cos (theta [i][1])*
                       cos (theta [i][2])*scale*431.8;
    coord [i][4][y] := coord [i][3][y] -
                       sin (theta [i][1])*
                       cos (theta [i][2])*scale*431.8;
    coord [i][4][z] := coord [i][3][z] -
                       sin (theta [i][2])*scale*431.8;

    coord [i][5][x] := coord [i][4][x] + scale*50.91*
                       sin (theta [i][1]);
    coord [i][5][y] := coord [i][4][y] -
                       cos (theta [i][1])*scale*50.91;
    coord [i][5][z] := coord [i][4][z];

    coord [i][6][x] := i*200 + trans [i][1][4];
    coord [i][6][y] := trans [i][2][4];
    coord [i][6][z] := 5 + scale*660.4 +
                       trans [i][3][4];

    coord [i][7][x] := coord [i][6][x] +
                       scale2*trans[i][1][2];
    coord [i][7][y] := coord [i][6][y] +
                       scale2*trans[i][2][2];
    coord [i][7][z] := coord [i][6][z] +
                       scale2*trans[i][3][2];
```

```
          coord [i][8][x] := coord [i][7][x] +
                             scale2*trans[i][1][3];
          coord [i][8][y] := coord [i][7][y] +
                             scale2*trans[i][2][3];
          coord [i][8][z] := coord [i][7][z] +
                             scale2*trans[i][3][3];

          coord [i][9][x] := coord [i][6][x] -
                             scale2*trans[i][1][2];
          coord [i][9][y] := coord [i][6][y] -
                             scale2*trans[i][2][2];
          coord [i][9][z] := coord [i][6][z] -
                             scale2*trans[i][3][2];

          coord [i][10][x] := coord [i][9][x] +
                              scale2*trans[i][1][3];
          coord [i][10][y] := coord [i][9][y] +
                              scale2*trans[i][2][3];
          coord [i][10][z] := coord [i][9][z] +
                              scale2*trans[i][3][3];

        end;  { i }
      end; { routine }

{*****************************}

  PROCEDURE Node (var collision : boolean;
                  coord : matrix2x16x3);

          { This routine performs the graph node search }

      var
        dist : real;
        i,j,k : integer;

      begin

        collision := false;   { default value }

           { calc intermediate nodes for accuracy in node
             search }
        for k := 1 to 2 do begin
          for j := 1 to 2 do begin  { node 11 and 12 }
            coord [k][10+j][x] := (coord [k][2][x] -
                                   coord [k][1][x])*
                                   j/3.0 + coord [k][1][x];
            coord [k][10+j][y] := (coord [k][2][y] -
                                   coord [k][1][y])*
                                   j/3.0 + coord [k][1][y];
            coord [k][10+j][z] := (coord [k][2][z] -
```

```
                                coord [k][1][z])*
                                j/3.0 + coord [k][1][z];
    end; { for j }
    for j := 1 to 2 do begin    { nodes 13 and 14 }
      coord [k][12+j][x] := (coord [k][4][x] -
                             coord [k][3][x])*j/3.0 +
                             coord [k][3][x];
      coord [k][12+j][y] := (coord [k][4][y] -
                             coord [k][3][y])*j/3.0 +
                             coord [k][3][y];
      coord [k][12+j][z] := (coord [k][4][z] -
                             coord [k][3][z])*j/3.0 +
                             coord [k][3][z];
    end;  { for j }
    for j := 1 to 2 do begin    { nodes 15 and 16 }
      coord [k][14+j][x] := (coord [k][6][x] -
                             coord [k][5][x])*j/3.0 +
                             coord [k][5][x];
      coord [k][14+j][y] := (coord [k][6][y] -
                             coord [k][5][y])*j/3.0 +
                             coord [k][5][y];
      coord [k][14+j][z] := (coord [k][6][z] -
                             coord [k][5][z])*j/3.0 +
                             coord [k][5][z];
    end;  { for j }
  end;  { for k }

    { do graph node search }
  gotoxy (2,2); write ('GRAPH NODE SEARCH');
  for k := 1 to 16 do begin
    for j := 1 to 16 do begin
      dist := sqrt (sqr (coord [1][k][x] -
                         coord [2][j][x]) +
                         sqr (coord [1][k][y] -
                         coord [2][j][y]) +
                        sqr (coord [1][k][z] -
                         coord [2][j][z]));

   { range for collision }
      if (dist <= 35.0) then begin
        gotoxy (2,2);
        write ('NODE COLLISION WILL OCCUR');
        collision := true;
        exit;  { return to main }
      end; { if }
    end; { for j }
  end;  { for k }

      { delete words above }
  gotoxy (2,2); write ('                        ');
```

```
    end; { routine }

{*****************************}

  PROCEDURE CalcRot (var rot : matrix3x3 ;
                     rpy : vector3);

        { This routine calculates the rot mx given the rpy
          angles }
    begin

      rot [1][1] := cos(rpy [3])*cos(rpy [2]);
      rot [1][2] := cos(rpy [3])*sin(rpy [2])*
                    sin(rpy [1]) - sin(rpy [3])*
                    cos(rpy [1]);
      rot [1][3] := cos(rpy [3])*sin(rpy [2])*
                    cos(rpy [1]) + sin(rpy [3])*
                    sin(rpy [1]);

      rot [2][1] := sin(rpy [3])*cos(rpy [2]);
      rot [2][2] := sin(rpy [3])*sin(rpy [2])*
                    sin(rpy [1]) + cos(rpy [3])*
                    cos(rpy [1]);
      rot [2][3] := sin(rpy [3])*sin(rpy [2])*
                    cos(rpy [1]) - cos(rpy [3])*
                    sin(rpy [1]);

      rot [3][1] := -sin(rpy [2]);
      rot [3][2] := cos(rpy [2])*sin(rpy [1]);
      rot [3][3] := cos(rpy [2])*cos(rpy [1]);

    end; {routine}

{*****************************}

  PROCEDURE CalcTran (var trans : matrix2x4x4;
                      theta : vector2x6);

        { This routine calculates the trans matricies
          given the joint angles }

    const
     scale = 0.20;

    var
     i : integer;

    begin
      for i := 1 to 2 do begin

        trans [i][1][4] := cos (theta [i][1])*(scale*
```

```
                    431.8*cos (theta [i][2]) -
                    scale*20.32*cos
                    (theta [i][2] +
                    theta [i][3]) -
                    scale*433.07*sin (
                    theta [i][2] +
                    theta [i][3])) -
                    scale*149.09*sin (
                    theta [i][1]);
trans [i][2][4] := sin (theta [i][1])*(scale*431.8*
                    cos (theta [i][2]) -
                    scale*20.32*cos (theta [i][2] +
                    theta [i][3]) -
                    scale*433.07*sin (
                    theta [i][2] +
                    theta [i][3])) +
                    scale*149.09*cos (
                    theta [i][1]);
trans [i][3][4] := scale*20.32*sin (theta [i][2] +
                    theta [i][3]) -
                    scale*431.8*sin (
                    theta [i][2]) -
                    scale*433.07*
                    cos (theta [i][2] +
                    theta [i][3]);

trans [i][1][1] := cos (theta[i][1])*
                    (cos (theta[i][2] +
                    theta[i][3])*(cos
                    (theta[i][4])*
                    cos (theta[i][5])*
                    cos (theta[i][6]) -
                    sin (theta[i][4])*
                    sin (theta[i][6])) -
                    sin (theta[i][2] +
                    theta[i][3])*sin (theta[i][5])*
                    cos (theta[i][6])) +
                    sin (theta[i][1])*
                    (sin (theta[i][4])*
                    cos (theta[i][5])*
                    cos (theta[i][6]) +
                    cos (theta[i][4])*
                    sin (theta[i][6]));
trans [i][2][1] := sin (theta[i][1])*
                    (cos (theta[i][2] +
                    theta[i][3])*(cos(theta[i][4])*
                    cos (theta[i][5])*
                    cos (theta[i][6]) -
                    sin (theta[i][4])*
                    sin (theta[i][6])) -
                    sin (theta[i][2] +
```

```
                        theta[i][3])*
                        sin (theta[i][5])*
                        cos (theta[i][6])) -
                        cos (theta[i][1])*
                        (sin (theta[i][4])*
                        cos (theta[i][5])*
                        cos (theta[i][6]) +
                        cos (theta[i][4])*
                        sin (theta[i][6]));
trans [i][3][1] := -sin (theta[i][2] +
                        theta[i][3])*
                        (cos (theta[i][4])*
                        cos (theta[i][5])*
                        cos (theta[i][6]) -
                        sin (theta[i][4])*
                        sin (theta[i][6])) -
                        cos (theta[i][2] +
                        theta[i][3])*sin (theta[i][5])*
                        cos (theta[i][6]);

trans [i][1][2] := cos (theta[i][1])*
                        (-cos (theta[i][2] +
                        theta[i][3])*
                        (cos (theta[i][4])*
                        cos (theta[i][5])*
                        sin (theta[i][6]) +
                        sin (theta[i][4])*
                        cos (theta[i][6])) +
                        sin (theta[i][2] +
                        theta[i][3])*
                        sin (theta[i][5])*
                        sin (theta[i][6])) +
                        sin (theta[i][1])*
                        (-sin (theta[i][4])*
                        cos (theta[i][5])*
                        sin (theta[i][6]) +
                        cos (theta[i][4])*
                        cos (theta[i][6]));
trans [i][2][2] := sin (theta[i][1])*
                        (-cos (theta[i][2] +
                        theta[i][3])*
                        (cos (theta[i][4])*
                        cos (theta[i][5])*
                        sin (theta[i][6]) +
                        sin (theta[i][4])*
                        cos (theta[i][6])) +
                        sin (theta[i][2] +
                        theta[i][3])*
                        sin (theta[i][5])*
                        sin (theta[i][6])) -
                        cos (theta[i][1])*
```

```
                        (-sin (theta[i][4])*
                        cos (theta[i][5])*
                        sin (theta[i][6]) +
                        cos (theta[i][4])*
                        cos (theta[i][6]));
    trans [i][3][2] := sin (theta[i][2] + theta[i][3])*
                        (cos (theta[i][4])*
                        cos (theta[i][5])*
                        sin(theta[i][6]) +
                        sin (theta[i][4])*
                        cos (theta[i][6])) +
                        cos (theta[i][2] +
                        theta[i][3])*
                        sin(theta[i][5])*
                        sin (theta[i][6]);

    trans [i][1][3] := -cos (theta[i][1])*
                        (cos (theta[i][2] +
                        theta[i][3])*
                        cos (theta[i][4])*
                        sin (theta[i][5]) +
                        sin (theta[i][2] +
                        theta[i][3])*
                        cos (theta[i][5])) -
                        sin (theta[i][1])*
                        sin (theta[i][4])*
                        sin (theta[i][5]);
    trans [i][2][3] := -sin (theta[i][1])*
                        (cos (theta[i][2] +
                        theta[i][3])*
                        cos (theta[i][4])*
                        sin (theta[i][5]) +
                        sin (theta[i][2] +
                        theta[i][3])*
                        cos (theta[i][5])) +
                        cos (theta[i][1])*
                        sin (theta[i][4])*
                        sin (theta[i][5]);
    trans [i][3][3] := sin (theta[i][2] +
                        theta[i][3])*cos (theta[i][4])*
                        sin (theta[i][5]) -
                        cos (theta[i][2] +
                        theta[i][3])*
                        cos (theta[i][5]);

    trans [i][4][1] := 0.0;
    trans [i][4][2] := 0.0;
    trans [i][4][3] := 0.0;
    trans [i][4][4] := 1.0;

end { i }
```

```
    end; { routine }

{*****************************}

  PROCEDURE Step (var numsteps : integer;
                  start,finish : matrix2x4x4);

        { This routine determines a uniform number of
          steps to take from a starting position to a
          final position }

    var
      dist : real;  { distance between start and finish }
      i    : integer;  { counter }
    begin
      dist := 0.0;  {initialize }
      for i := 1 to 3 do  { calculate distance }
        dist := dist + sqr (start [1][i][4] -
                            finish [1][i][4]);
      dist := sqrt (dist);

      numsteps := round (dist / 10.0);
         { where 10.0 is a predefined distance per step }

      if (numsteps = 0) then   { case of pure rotation }
        numsteps := 5;

    end; { routine }

{*****************************}

  PROCEDURE InvKin (var theta : vector2x6;
                    trans : matrix2x4x4;
                    chars : vector2x3 );

        { This routine inputs the transformation matrix
          and the char matrix to return the joint
          angles  }

    var
      inter1, inter2,
      inter3, inter4 : real; { intermediate answers }
      param : array [1..4,1..4] of real; { robot params }

    begin { procedure InvKin }

      param [3][3] := 149.09*0.2; { set four parameters }
      param [3][2] := 431.8*0.2; { times the scale factor }
      param [4][2] := -20.32*0.2;
      param [4][3] := 433.07*0.2;
```

```
for i := 1 to 2 do begin
  theta [i][1] := atan2 (trans [i][2][4],
                  trans [i][1][4]) -
                  atan2 (param [3][3],
                 chars [i][1]*
                  sqrt (sqr (trans [i][1][4]) +
                  sqr (trans [i][2][4]) -
                  sqr (param [3][3])));

  inter1 := (sqr (trans [i][1][4]) +
             sqr (trans [i][2][4]) +
             sqr (trans [i][3][4]) -
             sqr (param [3][2]) -
             sqr (param [4][2]) -
             sqr (param [3][3]) -
             sqr (param [4][3])) /
             (2.0* param [3][2]);

  theta [i][3] := atan2 (param [4][2],
                  param [4][3]) -
                  atan2 (inter1,
                  chars [i][2]*
                  sqrt (sqr (param [4][2]) +
                  sqr (param [4][3]) -
                  sqr (inter1)));

  inter2 := atan2 (trans [i][3][4]* (-param [4][2] -
                  param [3][2]*
                  cos (theta [i][3])) -
                  (cos (theta [i][1])*
                  trans [i][1][4] +
                  sin (theta [i][1])*
                  trans [i][2][4])*
                  (param [4][3] - param [3][2]*
                  sin (theta [i][3])),
                 trans [i][3][4]* (param [3][2]*
                  sin (theta [i][3])-
                  param [4][3]) + (param [4][2] +
                  param [3][2]*
                  cos (theta [i][3]))*
                  (cos (theta [i][1])*
                  trans [i][1][4] +
                  sin (theta [i][1])*
                  trans [i][2][4]));

  theta [i][2] := inter2 - theta [i][3];

  inter3 := -trans [i][1][3]* sin (theta [i][1]) +
              trans [i][2][3]* cos (theta [i][1]);

  inter4 := -trans [i][1][3]* cos (theta [i][1])*
```

```
                cos (theta [i][2] +
                theta [i][3]) - trans [i][2][3]*
                sin (theta [i][1])*
                cos (theta [i][2] + theta [i][3]) +
                trans [i][3][3]* sin (theta [i][2] +
                theta [i][3]);

if (abs (inter3) < 0.001) and
    (abs (inter4) < 0.001) then
  theta [i][5] := 0.0        { Case of singularity
                               of theta 5.  Assume
                               theta [4] equals its
                               old value }
else begin
  theta [i][4] := atan2 (inter3, inter4);

  theta [i][5] := atan2 (trans [i][3][3]*
                    sin (theta [i][2] +
                    theta [i][3])*
                    cos (theta [i][4])-
                    trans [i][1][3]*
                    (cos (theta [i][1])*
                    cos (theta [i][2] +
                    theta [i][3])*
                    cos (theta [i][4]) +
                    sin (theta [i][1])*
                    sin (theta [i][4])) -
                    trans [i][2][3]*
                    (sin (theta [i][1])*
                    cos (theta [i][2] +
                    theta [i][3])*
                    cos (theta [i][4]) -
                    cos (theta [i][1])*
                    sin (theta [i][4])),
                   -trans [i][1][3]*
                    cos (theta [i][1])*
                    sin(theta [i][2] +
                    theta [i][3]) -
                    trans [i][2][3]*
                    sin (theta [i][1])*
                    sin (theta [i][2] +
                    theta [i][3]) -
                    trans [i][3][3]*
                    cos (theta [i][2] +
                    theta [i][3]));
end; { else if }

theta [i][6] := atan2 (-trans [i][1][1]*
                  (cos (theta [i][1])*
                  cos (theta [i][2] + theta [i][3])*
                  sin (theta [i][4]) -
```

```
                        sin (theta [i][1])*
                        cos (theta [i][4])) -
                        trans [i][2][1]*
                        (sin (theta [i][1])*
                        cos (theta [i][2] +
                        theta [i][3])* sin (
                        theta [i][4]) +
                        cos (theta [i][1])*
                        cos (theta [i][4])) +
                        trans [i][3][1]*
                        sin (theta [i][2] +
                        theta [i][3])* sin (theta [i][4]),
                        trans [i][1][1]*
                        ((cos (theta [i][1])*
                        cos (theta [i][2] + theta [i][3])*
                        cos (theta [i][4]) +
                        sin (theta [i][1])*
                        sin (theta [i][4]))*
                        cos (theta [i][5]) -
                        cos (theta [i][1])*
                        sin (theta [i][2] +
                        theta [i][3])*
                        sin (theta [i][5])) +
                        trans [i][2][1]*
                        (cos (theta [i][5])*
                        (sin (theta [i][1])*
                        cos (theta [i][2] +
                        theta [i][3])* cos (
                        theta [i][4]) -
                        cos (theta [i][1])*
                        sin (theta [i][4])) -
                        sin (theta [i][1])*
                        sin (theta [i][2] +
                        theta [i][3])*
                        sin (theta [i][5])) -
                        trans [i][3][1]*(sin (
                        theta [i][2]+
                        theta [i][3])* cos (theta [i][4])*
                        cos (theta [i][5]) +
                        cos (theta [i][2] +theta [i][3])*

                        sin (theta [i][5])));

    if (chars [i][3] = -1) then begin { flip wrist
                                        characteristic }
      theta [i][4] := theta [i][4] + pi;
      theta [i][5] := -theta [i][5];
      theta [i][6] := theta [i][6] + pi;
    end; { if }
  end; { for i }
end;  { routine }
```

```
{*****************************}

  PROCEDURE Border (UpperLeftX, UpperLeftY, LowerRightX,
                    LowerRightY,DivisionY, DivisionX :
                    integer) ; begin

        { This procedure draws the border of the screen  }

    draw (UpperLeftX, UpperLeftY,
          LowerRightX, UpperLeftY, 1);
    draw (LowerRightX, UpperLeftY,
          LowerRightX, LowerRightY, 1);
    draw (LowerRightX, LowerRightY,
          UpperLeftX, LowerRightY, 1);
    draw (UpperLeftX, LowerRightY,
          UpperLeftX, UpperLeftY, 1);
    draw (UpperLeftX, DivisionY,
          LowerRightX, DivisionY, 1);
    draw (DivisionX, UpperLeftY,
          DivisionX, DivisionY, 1);

  end; { routine }

{*****************************}

  PROCEDURE DrawIquad ( x1, y1, x2, y2 : real);

        { This procedure converts coords in quad I to
              screen coords for printing }

    begin
      draw (round(x1), round(153 - (0.4*y1)),
            round(x2), round(153 - (0.4*y2)), 1);
    end; { routine }

{*****************************}

  PROCEDURE CircleIquad (x, y : real ; radius, color :
                         integer);

        { This routine draws a circle in screen coords
              given quad I coords }

    begin
      circle (round (x), round (153 - (0.4*y)),
              radius, color);
    end; { routine }

{*****************************}
```

```
PROCEDURE Drawframe (trans : matrix2x4x4; Theta :
                     vector2x6; coord : matrix2x16x3);

      { This routine draws the robot framework given the
        vector of thetas and the chars.  It updates the
        trans matrix.}

  var
    i,j : integer;
    dist : real;

  begin { drawframe }    { The variable coord [][][] is in
                           SCREEN COORDS }

    for i := 1 to 2 do begin

       { output WRIST coords to screen }

      GotoXY ((60 + (i-1)*9),22) ;
       write (coord [i][6][x] : 7 : 2) ;
      GotoXY ((60 + (i-1)*9),23) ;
       write (coord [i][6][y] : 7 : 2) ;
      GotoXY ((60 + (i-1)*9),24) ;
       write (coord [i][6][z] : 7 : 2) ;

       { output ORIENTATION matricies to screen }

      GotoXY ((2 + (i-1)*23),22) ;
       write (trans [i][1][1] : 7 : 3);
       write (trans [i][1][2] : 7 : 3);
       write (trans [i][1][3] : 7 : 3);
      GotoXY ((2 + (i-1)*23),23) ;
       write (trans [i][2][1] : 7 : 3);
       write (trans [i][2][2] : 7 : 3);
       write (trans [i][2][3] : 7 : 3);
      GotoXY ((2 + (i-1)*23),24) ;
       write (trans [i][3][1] : 7 : 3);
       write (trans [i][3][2] : 7 : 3);
       write (trans [i][3][3] : 7 : 3);

      for j := 1 to 6 do begin
        GotoXY (73, 5 + j + (i-1)*7);
        write (theta [i][j]*180/pi : 6 : 2);
      end; { for j }
    end;  { for i }

    clearscreen;          { clear previous picture from
                            screen }

    for i := 1 to 2 do begin
```

```
        drawIquad (coord [i][1][x],coord [i][1][z],
                   coord [i][2][x],coord [i][2][z]);
        drawIquad (coord [i][2][x],coord [i][2][z],
                   coord [i][3][x],coord [i][3][z]);
        drawIquad (coord [i][3][x],coord [i][3][z],
                   coord [i][4][x],coord [i][4][z]);
        drawIquad (coord [i][4][x],coord [i][4][z],
                   coord [i][5][x],coord [i][5][z]);
        drawIquad (coord [i][5][x],coord [i][5][z],
                   coord [i][6][x],coord [i][6][z]);
        circleIquad (coord [i][6][x],
                     coord [i][6][z], 2, 1);
        drawIquad (coord [i][6][x],coord [i][6][z],
                   coord [i][7][x],coord [i][7][z]);
        drawIquad (coord [i][7][x],coord [i][7][z],
                   coord [i][8][x],coord [i][8][z]);
        drawIquad (coord [i][6][x],coord [i][6][z],
                   coord [i][9][x],coord [i][9][z]);
        drawIquad (coord [i][9][x],coord [i][9][z],
                   coord [i][10][x],coord [i][10][z]);
        circleIquad (coord [i][10][x],
                     coord [i][10][z], 1, 1);

      end;   { for i }
    end;    { routine }

{*****************************}

  PROCEDURE MulTran ( var tran1, tran2, prod :
                      matrix2x4x4);

        { This routine multiplies two 2x4x4 matricies :
          the trans matricies }

    var
      i, j, k, l : integer;  { counters }

    begin
      for k := 1 to 2 do begin
        for i := 1 to 4 do begin
          for j := 1 to 4 do begin
            prod [k][i][j] := 0.0;
            for l := 1 to 4 do
              prod [k][i][j] := prod [k][i][j] +
                                tran1 [k][i][l]*
                                tran2 [k][l][j];
          end;  { for j }
        end;  { for i }
      end;  { for k }
    end; { routine }
```

```
{*****************************}

  PROCEDURE NexTran (var NexTrans : matrix2x4x4 ; Trb,
                     Tinvrb, Transinit, Tro, Tinvinitro :
                     matrix2x4x4);

        { This routine inputs the next Tro and returns the
          2 robot's trans }

    var
      prod1, prod2, prod3 : matrix2x4x4;
      i, j : integer;

    begin { procedure }

      multran (Tinvrb, Tro, prod1);
      multran (prod1, Tinvinitro, prod2);
      multran (prod2, Trb, prod3);
      multran (prod3, Transinit, NexTrans);

   end;  { routine }

{*****************************}

  PROCEDURE InvTran (var tran, invtran : matrix2x4x4 );

        { This routine calculates the inverse of the trans
          matricies using the formulas developed in the
          Lee Tutorial pg 51  }
    var
      i, j, k : integer;
      sum : real;

    begin  { routine }
      for i := 1 to 2 do begin
        for j := 1 to 3 do begin        { rot inverse }
          for k := 1 to 3 do
            invtran [i][j][k] := tran [i][k][j];
        end;  { for j }
      end;  { for i }

      for i := 1 to 2 do begin
        for j := 1 to 3 do begin     { trans inverse }
          sum := 0.0;
          for k := 1 to 3 do
            sum := sum + tran [i][k][j]*tran [i][k][4];
          invtran [i][j][4] := -sum;
        end; { for j }
        for j := 1 to 3 do
          invtran [i][4][j] := 0.0;
        invtran [i][4][4] := 1.0;
```

```
    end; { for j }
  end;  { routine }

{*****************************}

  PROCEDURE DrawBox ( Tro : matrix2x4x4 );

        { This routine draws the box to be held by the
          robots. }

    const
      scale1 = 75;  { n scale factor }
      scale2 = 35; {s and a scale factor }

    var
      boxcoord : array [1..8, (x,z)] of real;
      i, j     : integer;

    begin

      for i := 1 to 3 do begin
        for j := 2 to 3 do
          Tro [1][i][j] := Tro [1][i][j] * scale2;
        Tro [1][i][1] := Tro [1][i][1] * scale1;
      end;  { for i }

      boxcoord [1][x] := Tro [1][1][4] + Tro [1][1][1] +
                         Tro [1][1][2];
      boxcoord [1][z] := Tro [1][3][4] + Tro [1][3][1] +
                           Tro [1][3][2];

      boxcoord [2][x] := Tro [1][1][4] + Tro [1][1][1] -
                         Tro [1][1][2];
      boxcoord [2][z] := Tro [1][3][4] + Tro [1][3][1] -
                         Tro [1][3][2];

      boxcoord [3][x] := Tro [1][1][4] - Tro [1][1][1] -
                         Tro [1][1][2];
      boxcoord [3][z] := Tro [1][3][4] - Tro [1][3][1] -
                         Tro [1][3][2];

      boxcoord [4][x] := Tro [1][1][4] - Tro [1][1][1] +
                         Tro [1][1][2];
      boxcoord [4][z] := Tro [1][3][4] - Tro [1][3][1] +
                         Tro [1][3][2];

      boxcoord [5][x] := boxcoord [1][x] + Tro [1][1][3];
      boxcoord [5][z] := boxcoord [1][z] + Tro [1][3][3];

      boxcoord [6][x] := boxcoord [2][x] + Tro [1][1][3];
      boxcoord [6][z] := boxcoord [2][z] + Tro [1][3][3];
```

```
      boxcoord [7][x] := boxcoord [3][x] + Tro [1][1][3];
      boxcoord [7][z] := boxcoord [3][z] + Tro [1][3][3];

      boxcoord [8][x] := boxcoord [4][x] + Tro [1][1][3];
      boxcoord [8][z] := boxcoord [4][z] + Tro [1][3][3];

      drawIquad (boxcoord [1][x],boxcoord [1][z],
                 boxcoord [2][x],boxcoord [2][z]);
      drawIquad (boxcoord [2][x],boxcoord [2][z],
                 boxcoord [3][x],boxcoord [3][z]);
      drawIquad (boxcoord [3][x],boxcoord [3][z],
                 boxcoord [4][x],boxcoord [4][z]);
      drawIquad (boxcoord [4][x],boxcoord [4][z],
                 boxcoord [1][x],boxcoord [1][z]);
      drawIquad (boxcoord [1][x],boxcoord [1][z],
                 boxcoord [5][x],boxcoord [5][z]);
      drawIquad (boxcoord [2][x],boxcoord [2][z],
                 boxcoord [6][x],boxcoord [6][z]);
      drawIquad (boxcoord [3][x],boxcoord [3][z],
                 boxcoord [7][x],boxcoord [7][z]);
      drawIquad (boxcoord [4][x],boxcoord [4][z],
                 boxcoord [8][x],boxcoord [8][z]);
      drawIquad (boxcoord [5][x],boxcoord [5][z],
                 boxcoord [6][x],boxcoord [6][z]);
      drawIquad (boxcoord [6][x],boxcoord [6][z],
                 boxcoord [7][x],boxcoord [7][z]);
      drawIquad (boxcoord [7][x],boxcoord [7][z],
                 boxcoord [8][x],boxcoord [8][z]);
      drawIquad (boxcoord [8][x],boxcoord [8][z],
                 boxcoord [5][x],boxcoord [5][z]);
      circleIquad (Tro [1][1][4], Tro [1][3][4], 2, 1);

    end; { routine }

{*****************************}

  PROCEDURE ThetaIncr (theta, thetanew : vector2x6;
                       var incr : vector2x6;
                       var steps : integer);

        { This routine calculates the increment for the
          thetas given the initial and final theta
          vectors.  The increment of the largest
          change will be approx 5 degrees.  }

    var
      i, j  : integer;
      maxdiff : real;
      diff : vector2x6;
```

```
    begin  { routine }
      maxdiff := 0.0;
      for i := 1 to 2 do begin
        for j := 1 to 6 do begin
          diff [i][j] := thetanew [i][j] - theta [i][j];
          if abs (diff [i][j]) > maxdiff then
            maxdiff := abs (dif_ [i][j]);
        end;  { for j }
      end;  { for i }

      steps := round (maxdiff / (10*pi/180));
      for i := 1 to 2 do begin
        for j := 1 to 6 do
          incr [i][j] := diff [i][j]/steps;
      end; { for i }
    end;  { routine }

{*****************************}

  PROCEDURE DrawObst;

        { This routine draws the obstacle for avoidance }

    begin  { routine }
      drawIquad (250,60,270,60);
      drawIquad (270,60,270,40);
      drawIquad (270,40,250,40);
      drawIquad (250,40,250,60);
    end;  { routine }

{*****************************}

  PROCEDURE DrawObst1;

        { Draw obstacle with shadow }

    begin  { routine }
      drawIquad (210,130,375,130);
      drawIquad (375,130,375,70);
      drawIquad (375,70,315,70);
      drawIquad (300,70,240,70);
      drawIquad (240,70,240,10);
      drawIquad (240,10,210,10);
      drawIquad (210,10,210,130);
      drawIquad (210,95,300,95);
      drawIquad (300,95,300,70);
      drawIquad (315,70,315,10);
      drawIquad (315,10,240,10);

      fillshape (215,120,1,1);  { fill obstacle (units in
                                  screen coord }
```

```
  end; { routine }

{*****************************}

  PROCEDURE DrawBox2 (trans : matrix2x4x4);

        { This routine draws a box for one robot to hold }

    const
      scale1 = 25;  { n, s, and a scale factor }

    var
      boxcoord : array [1..8, (x,z)] of real;
      i, j     : integer;

    begin { routine }

      for i := 1 to 3 do begin
        for j := 1 to 3 do
          Trans [1][i][j] := Trans [1][i][j]*scale1;
      end;

      trans [1][1][4] := trans [1][1][4] + 200.0;
      trans [1][3][4] := trans [1][3][4] + 137.08;


      boxcoord [1][x] := Trans [1][1][4] +
                         Trans [1][1][1]+
                         Trans [1][1][2];
      boxcoord [1][z] := Trans [1][3][4] +
                         Trans [1][3][1]+
                         Trans [1][3][2];

      boxcoord [2][x] := Trans [1][1][4] +
                         Trans [1][1][1] -
                         Trans [1][1][2];
      boxcoord [2][z] := Trans [1][3][4] +
                         Trans [1][3][1] -
                         Trans [1][3][2];

      boxcoord [3][x] := Trans [1][1][4] -
                         Trans [1][1][1] -
                         Trans [1][1][2];
      boxcoord [3][z] := Trans [1][3][4] -
                         Trans [1][3][1] -
                         Trans [1][3][2];

      boxcoord [4][x] := Trans [1][1][4] -
                         Trans [1][1][1] +
                         Trans [1][1][2];
```

```
        boxcoord [4][z] := Trans [1][3][4] -
                           Trans [1][3][1] +
                           Trans [1][3][2];

        boxcoord [5][x] := boxcoord [1][x] + Trans [1][1][3];
        boxcoord [5][z] := boxcoord [1][z] + Trans [1][3][3];

        boxcoord [6][x] := boxcoord [2][x] + Trans [1][1][3];
        boxcoord [6][z] := boxcoord [2][z] + Trans [1][3][3];

        boxcoord [7][x] := boxcoord [3][x] + Trans [1][1][3];
        boxcoord [7][z] := boxcoord [3][z] + Trans [1][3][3];

        boxcoord [8][x] := boxcoord [4][x] + Trans [1][1][3];
        boxcoord [8][z] := boxcoord [4][z] + Trans [1][3][3];

        drawIquad (boxcoord [1][x],boxcoord [1][z],
                   boxcoord [2][x],boxcoord [2][z]);
        drawIquad (boxcoord [2][x],boxcoord [2][z],
                   boxcoord [3][x],boxcoord [3][z]);
        drawIquad (boxcoord [3][x],boxcoord [3][z],
                   boxcoord [4][x],boxcoord [4][z]);
        drawIquad (boxcoord [4][x],boxcoord [4][z],
                   boxcoord [1][x],boxcoord [1][z]);
        drawIquad (boxcoord [1][x],boxcoord [1][z],
                   boxcoord [5][x],boxcoord [5][z]);
        drawIquad (boxcoord [2][x],boxcoord [2][z],
                   boxcoord [6][x],boxcoord [6][z]);
        drawIquad (boxcoord [3][x],boxcoord [3][z],
                   boxcoord [7][x],boxcoord [7][z]);
        drawIquad (boxcoord [4][x],boxcoord [4][z],
                   boxcoord [8][x],boxcoord [8][z]);
        drawIquad (boxcoord [5][x],boxcoord [5][z],
                   boxcoord [6][x],boxcoord [6][z]);
        drawIquad (boxcoord [6][x],boxcoord [6][z],
                   boxcoord [7][x],boxcoord [7][z]);
        drawIquad (boxcoord [7][x],boxcoord [7][z],
                   boxcoord [8][x],boxcoord [8][z]);
        drawIquad (boxcoord [8][x],boxcoord [8][z],
                   boxcoord [5][x],boxcoord [5][z]);

    end;  { routine }

{*****************************}

  PROCEDURE DrawSquare;

        { This procedure draws the initial position
          for the box to be held }

    begin  { routine }
```

```
      drawIquad (128.66,43.65,178.66,43.65);
      drawIquad (178.66,43.65,178.66,18.65);
      drawIquad (178.66,18.65,128.66,18.65);
      drawIquad (128.66,18.65,128.66,43.65);
    end;  { routine }

{*****************************}

  PROCEDURE RobMov (var theta : vector2x6;
                    thetaNEW : vector2x6;
                    var trans : matrix2x4x4;
                    var coord : matrix2x16x3;
                    var chars : vector2x3;
                    grip : boolean);

        { This routine moves the robots from a present
          location (theta) to a new location (thetaNEW) }
    var
       i, j, k, steps : integer;
       angle, max_angle : real;
       del_angle : array [1..2,1..6] of real;

    begin

      angle := 0.0;            { calc maximum angle change }
      max_angle := 0.0;
      for i := 1 to 2 do begin
        for j := 1 to 6 do begin
          angle := abs (theta [i][j] - thetaNEW [i][j]);
          if (angle > max_angle) then
            max_angle := angle;
        end;  { for j }
      end;  { for i }

          { determine num of steps }
      steps := round (max_angle/(20.0*pi/180));
      for i := 1 to 2 do begin   { calc angle change per
                                           step }
        for j := 1 to 6 do
          del_angle [i][j] := (thetaNEW [i][j] -
                               theta [i][j])/steps;
      end; { for i }

      for i := 1 to steps do begin
        for j := 1 to 2 do begin     { perform movement }
          for k := 1 to 6 do
            theta [j][k] := theta [j][k] +
                            del_angle [j][k];
        end; { for j }

        calctran (trans,theta);
```

```
      coordinate (coord,trans,theta);
      drawframe (trans, theta, coord);

      if (grip) then
        drawbox2 (trans)
      else
        drawsquare;

    end;  { for i }

  end;  { routine }


begin { Program }

  hires;                    { set screen to high resolution }
  hirescolor (15);          { set color for drawing  }
  palette(3);               { allows for draw command to
                              still be active with
                              graph.p included     }

  chars [1][1] := 1;
   chars [2][1] := 1;    { shoulder left condition }
  chars [1][2] := 1;
   chars [2][2] := 1;    { arm - up condition }
  chars [1][3] := 1;
   chars [2][3] := 1;    { no-flip condition in wrist }

  for i := 1 to 2 do begin
    theta [i][1] := 0.0;  { initialize HOME thetas }
    theta [i][2] := 0.0;
    theta [i][3] := 0.0;
    theta [i][4] := 0.0;
    theta [i][5] := 0.0;
    theta [i][6] := 0.0;
  end; { for i }

        { initialize matricies to zero }
  for i := 1 to 2 do begin
    for j := 1 to 4 do begin
      for k := 1 to 4 do begin
        Trb [i][j][k] := 0.0;
        Tinvrb [i][j][k] := 0.0;
        Nextrans [i][j][k] := 0.0;
        Tro [i][j][k] := 0.0;
        Tinvinitro [i][j][k] := 0.0;
        TroEND [i][j][k] := 0.0;
      end;  { for k }
    end;  { for j }
  end;  { for i }
```

```
        { set all diagonal elements to one }
    for i := 1 to 2 do begin
      for j := 1 to 4 do begin
        Trb [i][j][j] := 1.0;
        Tinvrb [i][j][j] := 1.0;
      end;  { for j }
    end;  { for i }

        { set other constant elements of T  matricies }
    Trb [1][1][4] := 200.0;
    Trb [1][3][4] := 137.08;

    Tinvrb [1][1][4] := -200.0;
    Tinvrb [1][3][4] := -137.08;

    Trb [2][1][4] := 400.0;
    Trb [2][3][4] := 137.08;

    Tinvrb [2][1][4] := -400.0;
    Tinvrb [2][3][4] := -137.08;

    Border (5,5,635,195,160,555);  { draw border }
    GraphWindow (6, 6, 554, 159);  { setup working window }

        { draw initial configuration }
    CalcTran (trans,theta);
    Coordinate (coord,trans,theta);
    DrawFrame (trans, theta, coord);

    RobotID := 1;  { initialize robot choice toggle }


    while inkey <> #27 do begin
FLAG1:
      read (kbd, inkey);
      if (inkey = #27) and keypressed then
        read (kbd, inkey);
      case inkey of
          #120    :   goto FLAG7;  { alt-keys functions }
          #121    :   goto FLAG2;
          #122    :   goto FLAG2;
          #123    :   goto FLAG2;
          #124    :   goto FLAG2;
          #125    :   goto FLAG2;
          #126    :   goto FLAG2;
          #127    :   goto FLAG2;

          #113    :   begin
                        if (RobotID = 1) then RobotID := 2
                        else RobotID := 1;
                      end;
```

```
#59     :   begin  { F1 }
              theta [RobotID][1] :=
               theta [RobotID][1] +
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;
#104    :   begin  { ALT-F1 }
              theta [RobotID][1] :=
               theta [RobotID][1]-
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#60     :   begin  { F2 }
              theta [RobotID][2] :=
               theta [RobotID][2]+
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#105    :   begin  { ALT-F2 }
              theta [RobotID][2] :=
               theta [RobotID][2]-
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#61     :   begin  { F3 }
              theta [RobotID][3] :=
               theta [RobotID][3]+
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#106    :   begin  { ALT-F3 }
              theta [RobotID][3] :=
               theta [RobotID][3]-
               10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
```

```
                DrawFrame (trans, theta, coord);
              end;

#62     :     begin  { F4 }
                theta [RobotID][4] :=
                 theta [RobotID][4]+
                 10*pi/180;
                CalcTran (trans,theta);
                Coordinate (coord,trans,theta);
                DrawFrame (trans, theta, coord);
              end;

#107    :     begin  { ALT-F4 }
                theta [RobotID][4] :=
                 theta [RobotID][4]-
                 10*pi/180;
                CalcTran (trans,theta);
                Coordinate (coord,trans,theta);
                DrawFrame (trans, theta, coord);
              end;

#63     :     begin  { F5 }
                theta [RobotID][5] :=
                 theta [RobotID][5]+
                 10*pi/180;
                CalcTran (trans,theta);
                Coordinate (coord,trans,theta);
                DrawFrame (trans, theta, coord);
              end;

#108    :     begin  { ALT-F5 }
                theta [RobotID][5] :=
                 theta [RobotID][5]-
                 10*pi/180;
                CalcTran (trans,theta);
                Coordinate (coord,trans,theta);
                DrawFrame (trans, theta, coord);
              end;

#64     :     begin  { F6 }
                theta [RobotID][6] :=
                 theta [RobotID][6]+
                 10*pi/180;
                CalcTran (trans,theta);
                Coordinate (coord,trans,theta);
                DrawFrame (trans, theta, coord);
              end;

#109    :     begin  { ALT-F6 }
                theta [RobotID][6] :=
                 theta [RobotID][6]-
```

```
                10*pi/180;
              CalcTran (trans,theta);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#71     :   begin  { HOME }
              trans [RobotID][1][4] :=
                    trans [RobotID][1][4] - 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#73     :   begin  { PGUP }
              trans [RobotID][1][4] :=
                    trans [RobotID][1][4] + 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#75     :   begin  { LT ARROW }
              trans [RobotID][2][4] :=
               trans [RobotID][2][4] - 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#77     :   begin  { RT ARROW }
              trans [RobotID][2][4] :=
               trans [RobotID][2][4] + 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#79     :   begin  { END }
              trans [RobotID][3][4] :=
               trans [RobotID][3][4] - 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
              DrawFrame (trans, theta, coord);
            end;

#81     :   begin  { PGDN }
              trans [RobotID][3][4] :=
               trans [RobotID][3][4] + 10.0;
              InvKin (theta, trans, chars);
              Coordinate (coord,trans,theta);
```

```
                        DrawFrame (trans, theta, coord);
                      end;

          #68     :   halt;  { F10 - terminate program }

      end;  {case}
    end; {while}

{*****************************}

{********  ROUTINE 1 **}

          { drilling operation }
FLAG7:

    gotoxy (2,2); write ('PHASE I');
     { hover }                              { ready to drill }
    thetaNEW [1][1] := -2.618;    thetaNEW [2][1] := -2.385;
    thetaNEW [1][2] := -0.524;    thetaNEW [2][2] :=  0.363;
    thetaNEW [1][3] :=  0.524;    thetaNEW [2][3] := -0.023;
    thetaNEW [1][4] :=  0.0;      thetaNEW [2][4] := -2.431;
    thetaNEW [1][5] :=  0.0;      thetaNEW [2][5] :=  1.835;
    thetaNEW [1][6] := -1.047;    thetaNEW [2][6] := -2.877;
    RobMov (theta,thetaNEW,trans,coord,chars,false);

    gotoxy (2,2); write ('PHASE II');  { pick up }

    for 1 := 1 to 5 do begin
      trans [1][3][4] := trans [1][3][4] - 10.0;
      InvKin (theta,trans,chars);
      Coordinate (coord,trans,theta);
      DrawFrame (trans,theta,coord);
      DrawSquare;
    end;  { for 1 }

    gotoxy (2,2); write ('PHASE III');
       { hover again }                      { same }
    thetaNEW [1][1] := -2.618;    thetaNEW [2][1] := -2.385;
    thetaNEW [1][2] := -0.524;    thetaNEW [2][2] :=  0.363;
    thetaNEW [1][3] :=  0.524;    thetaNEW [2][3] := -0.023;
    thetaNEW [1][4] :=  0.0;      thetaNEW [2][4] := -2.431;
    thetaNEW [1][5] :=  0.0;      thetaNEW [2][5] :=  1.835;
    thetaNEW [1][6] := -1.047;    thetaNEW [2][6] := -2.877;
    RobMov (theta,thetaNEW,trans,coord,chars,true);

    gotoxy (2,2); write ('PHASE IV');
       { move to vice position }            { same }
    thetaNEW [1][1] := -0.721;    thetaNEW [2][1] := -2.385;
    thetaNEW [1][2] :=  0.284;    thetaNEW [2][2] :=  0.363;
    thetaNEW [1][3] := -0.649;    thetaNEW [2][3] := -0.023;
    thetaNEW [1][4] :=  2.387;    thetaNEW [2][4] := -2.431;
```

```
thetaNEW [1][5] :=  1.300;   thetaNEW [2][5] :=  1.835;
thetaNEW [1][6] := -2.895;   thetaNEW [2][6] := -2.877;
RobMov (theta,thetaNEW,trans,coord,chars,true);

gotoxy (2,2); write ('PHASE V');
    { stay in vice position }          { drill }
thetaNEW [1][1] := -0.721;   thetaNEW [2][1] := -2.752;
thetaNEW [1][2] :=  0.284;   thetaNEW [2][2] :=  0.294;
thetaNEW [1][3] := -0.649;   thetaNEW [2][3] := -0.208;
thetaNEW [1][4] :=  2.387;   thetaNEW [2][4] := -2.826;
thetaNEW [1][5] :=  1.300;   thetaNEW [2][5] :=  1.655;
thetaNEW [1][6] := -2.895;   thetaNEW [2][6] := -3.070;
RobMov (theta,thetaNEW,trans,coord,chars,true);

gotoxy (2,2); write ('PHASE VI');
   { stay in vice position }      { retract drill }
thetaNEW [1][1] := -0.721;   thetaNEW [2][1] := -2.385;
thetaNEW [1][2] :=  0.284;   thetaNEW [2][2] :=  0.363;
thetaNEW [1][3] := -0.649;   thetaNEW [2][3] := -0.023;
thetaNEW [1][4] :=  2.387;   thetaNEW [2][4] := -2.431;
thetaNEW [1][5] :=  1.300;   thetaNEW [2][5] :=  1.835;
thetaNEW [1][6] := -2.895;   thetaNEW [2][6] := -2.877;
RobMov (theta,thetaNEW,trans,coord,chars,true);

goto FLAG1;
     { end of routine drill }

FLAG2:  { routine IN MAIN to lead robots through specified

          movement }

     { set thetas to an appropriate starting point }
theta [1][1] := -0.5123; theta [2][1] := -1.7703;
theta [1][2] := 0.1847 ; theta [2][2] := 0.1833;
theta [1][3] := -0.8479; theta [2][3] := 0.3838;
theta [1][4] := 0.0;     theta [2][4] := 0.0;
theta [1][5] := 0.6632;  theta [2][5] := -0.5671;
theta [1][6] := -0.5123; theta [2][6] := -1.7703;

     { robot initial postition }
CalcTran (trans,theta);
Coordinate (coord,trans,theta);
DrawFrame (trans, theta, coord);

     { set transinit mx to trans mx }
transinit := trans;

for k := 1 to 2 do begin
  for i := 1 to 3 do begin       { initialize Tro }
    for j := 1 to 3 do
      Tro [k][i][j] := trans [1][i][j];
```

```
        Tro [k][4][i] := 0.0;
      end;  { for i }
      Tro [k][1][4] := 377.3;
      Tro [k][2][4] := -40.18;
      Tro [k][3][4] := 50.47;
      Tro [k][4][4] := 1.0;
    end;  { for k }

        { calc Tinvinitro for nextran }
    InvTran (Tro, Tinvinitro);

    DrawBox (Tro);   { draw initial box }

{******** ROUTINE 2 **}

    if inkey = #121 then begin  { alt-2 key }

        { linear movement without node search
          - no obstacles }

      px := Tro [1][1][4] - 50.0;

        { run through movement routine }
      for i := 1 to 5 do begin
        for j := 1 to 2 do
          Tro [j][1][4] := Tro [j][1][4] - 10.0;

        { calculate next trans and draw robots }
        Nextran (Trans, Trb, Tinvrb,
                 Transinit, Tro, Tinvinitro);
        InvKin (theta, trans, chars);
        Coordinate (coord,trans,theta);
        DrawFrame (trans, theta, coord);
        DrawBox (Tro);
          { indicate Tro }
        CircleIquad (Tro [1][1][4],Tro [1][3][4],2,1);
        CircleIquad (px, Tro [1][3][4],2,1);

      end; { for i }
    end  { else if }

{********* ROUTINE 3 **}

    else if inkey = #122 then begin    { alt-3 key }

        { increase theta 5 of 30 degrees with trans being
          incremented }

      for i := 1 to 2 do begin       { set final position }
        TroEND [i][1][4] :=  340.0;
        TroEND [i][2][4] := -40.0;
```

```
  TroEND [i][3][4] :=  80.0;
end;  { for i }

    { calc Tinvinitro for nextran }
InvTran (Tro, Tinvinitro);

Step (numsteps,Tro,TroEND);  { determine number of
                                steps for move }
    { calc the increment on r-p-y }
del_theta [5] := 30.0*pi/180/numsteps;
    { del x }
del [1] := (TroEND [1][1][4] -
            Tro [1][1][4])/numsteps;
    { del y }
del [2] := (TroEND [1][2][4] -
            Tro [1][2][4])/numsteps;
    { del z }
del [3] := (TroEND [1][3][4] -
            Tro [1][3][4])/numsteps;

for i := 1 to numsteps do begin

   { calc rotation mx of Tro}
  ThetaTRO := theta;
  ThetaTRO [1][5] := Theta [1][5] + del_theta [5];
  CalcTran (RotTro1, ThetaTRO);

  for j := 1 to 2 do begin     { set ROT mx of Tro }
    for k := 1 to 3 do begin
      for l := 1 to 3 do
        Tro [j][k][l] := RotTro1 [1][k][l];
    end; { for k }
    for k := 1 to 3 do  { set translation mx of Tro }
      Tro [j][k][4] := Tro [j][k][4] + del[k];
  end; { for j }

   { calc the robot's next trans mx's }
  Nextran (trans, Trb, Tinvrb,
           Transinit, Tro, Tinvinitro);

   { calc robot positions and draw }
  InvKin (theta, trans, chars);
  Coordinate (coord,trans,theta);
  DrawFrame (trans,theta,coord);

  DrawBox (Tro);  { draw in object }

   { indicate end position and reference point }
  CircleIquad (Tro [1][1][4],tro [1][3][4],2,1);
  CircleIquad (TroEND [1][1][4],
               TroEND [1][3][4],2,1);
```

```
      end; { i }
    end  { if }

{********* ROUTINE 4 **}

    else if inkey = #123 then begin      { alt-4 key }

        { translation with roll (+30) and yaw (+20) }

      rpy [1] := -pi;
      rpy [2] := 0.0;
      rpy [3] := 0.0;

      for m := 1 to 2 do begin

        if (m=1) then begin  { PHASE I of movement }

         { initialize del_rpy }
          for i := 1 to 3 do
            del_rpy [i] := 0.0;

          for i := 1 to 2 do begin  { set final position }
            TroEND [i][1][4] :=  320.0;
            TroEND [i][2][4] := -60.0;
            TroEND [i][3][4] :=  60.0;
          end;  { for i }

          { determine number of steps for move }
          Step (numsteps,Tro,TroEND);
          { calc the increment on r-p-y }
          del_rpy [3] := 20.0*pi/180/numsteps;
          del_rpy [2] := 15.0*pi/180/numsteps;

        end { if }

        else if (m=2) then begin  { PHASE II of movement }

          for i := 1 to 3 do
            del_rpy [i] := 0.0;  { initilize del_rpy }

          for i := 1 to 2 do begin  { set final position }
            TroEND [i][1][4] :=  290.0;
            TroEND [i][2][4] := -80.0;
            TroEND [i][3][4] :=  70.0;
          end;  { for i }

          { determine number of steps for move }
          Step (numsteps,Tro,TroEND);

        end; {if}
```

```
          { del x }
        del [1] := (TroEND [1][1][4] -
                    Tro [1][1][4])/numsteps;
          { del y }
        del [2] := (TroEND [1][2][4] -
                    Tro [1][2][4])/numsteps;
          { del z }
        del [3] := (TroEND [1][3][4] -
                    Tro [1][3][4])/numsteps;

        for i := 1 to numsteps do begin

          { calc rotation mx of Tro}
          for j := 1 to 3 do  { incr rpy angles }
            rpy [j] := rpy [j] + del_rpy [j];

          CalcRot (RotTro,rpy); { calc rotation mx of Tro }

          for j := 1 to 2 do begin  { set ROT mx of Tro }
            for k := 1 to 3 do begin
              for l := 1 to 3 do
                Tro [j][k][l] := RotTro [k][l];
            end;  { for k }

           { set translation mx of Tro }
            for k := 1 to 3 do
              Tro [j][k][4] := Tro [j][k][4] + del[k];
         end;  { for j }

         { calc the robot's next trans mx's }
         Nextran (trans, Trb, Tinvrb, Transinit, Tro,
                  Tinvinitro);

         { draw frame }
         InvKin (theta, trans, chars);
         Coordinate (coord,trans,theta);
         DrawFrame (trans,theta,coord);
         DrawBox (Tro);
         CircleIquad (Tro [1][1][4],tro [1][3][4],2,1);
         CircleIquad (TroEND [1][1][4],
                      TroEND [1][3][4],2,1);

       end; { i }
     end; { m }
   end { else if }

{********* ROUTINE 5 **}

   else if inkey = #124 then begin    { alt-5 key }
```

```
        { linear translation with trans
          being incremented , node search, no obstacles }

      px := Tro [1][1][4] - 50.0;  { final location }

        { run through movement routine }
      for i := 1 to 5 do begin
        for j := 1 to 2 do  { increment Tro }
          Tro [j][1][4] := Tro [j][1][4] - 10.0;

        { draw the frame }
        Nextran (Trans, Trb, Tinvrb,
                 Transinit, Tro, Tinvinitro);
        InvKin (theta, trans, chars);
        Coordinate (coord,trans,theta);

        Node (collision,coord); { perform the node search }

          { a collision detected by Node}
        if (collision) then begin
            gotoxy (2,3);
            write ('MOVEMENT HAS BEEN FROZEN');
            exit;
        end;  { if }

          { draw the frame }
        DrawFrame (trans, theta, coord);
        DrawBox (Tro);
        CircleIquad (Tro [1][1][4],Tro [1][3][4],2,1);
        CircleIquad (px,Tro [1][3][4],2,1)

      end; { for i }
    end  { else if }

{******** ROUTINE 6 **}

    else if inkey = #125 then begin      { alt-6 key }

        { yaw 180 degrees for node search failure }

      rpy [1] := -pi;
      rpy [2] := 0.0;
      rpy [3] := 0.0;

      for i := 1 to 3 do  { initialize del_rpy }
        del_rpy [i] := 0.0;

      for i := 1 to 2 do begin       { set final position }
        TroEND [i][1][4] :=  300.0;
        TroEND [i][2][4] := -60.0;
        TroEND [i][3][4] :=  80.0;
```

```
end;  { for i }

numsteps := 10;

   { calculate the change in variable }
del_rpy [3] := -180.0*pi/180/numsteps;
del [1] := (TroEND [1][1][4] -
             Tro [1][1][4])/numsteps;
del [2] := (TroEND [1][2][4] -
             Tro [1][2][4])/numsteps;
del [3] := (TroEND [1][3][4] -
             Tro [1][3][4])/numsteps;

for i := 1 to numsteps do begin

               { calc rotation mx of Tro}
  for j := 1 to 3 do  { incr rpy angles }
    rpy [j] := rpy [j] + del_rpy [j];

  CalcRot (RotTro,rpy); { calc rotation mx of Tro }

  for j := 1 to 2 do begin    { set ROT mx of Tro }
    for k := 1 to 3 do begin
      for l := 1 to 3 do
        Tro [j][k][l] := RotTro [k][l];
    end;  { for k }

    for k := 1 to 3 do { set translation mx of Tro }
      Tro [j][k][4] := Tro [j][k][4] + del[k];
  end;  { for j }

   { calc the robot's next trans mx's }
  Nextran (trans, Trb, Tinvrb,
           Transinit, Tro, Tinvinitro);
  InvKin (theta, trans, chars);
  Coordinate (coord,trans,theta);

  Node (collision,coord);  { perform node search }

  if (collision) then begin { stop movement if
                              collision is to occur }
    gotoxy (2,3);
    write ('MOVEMENT HAS BEEN FROZEN');
    goto FLAG1;
  end; { if }

  DrawFrame (trans,theta,coord);
  DrawBox (Tro);
  CircleIquad (Tro [1][1][4],tro [1][3][4],2,1);
  CircleIquad (TroEND [1][1][4],
   TroEND [1][3][4],2,1);
```

```
      end; { i }
    end { else if }


{********* ROUTINE 7 **}

    else if inkey = #126 then begin      { alt-7 key }

        { linear x movement with upward obstacle
          avoidance }

      DrawObst;    { draw in original obstacle }

      for i := 1 to 2 do begin      { set final Tro }
        TroEND [i][1][1] :=  0.866;
        TroEND [i][1][2] := -0.5;
        TroEND [i][1][3] := -0.0;
        TroEND [i][1][4] :=  300.0;
        TroEND [i][2][1] := -0.5;
        TroEND [i][2][2] := -0.866;
        TroEND [i][2][3] :=  0.0;
        TroEND [i][2][4] := -40.0;
        TroEND [i][3][1] := -0.0;
        TroEND [i][3][2] :=  0.0;
        TroEND [i][3][3] := -1.0;
        TroEND [i][3][4] :=  130.0;
        TroEND [i][4][4] :=  1.0;
      end;
         { indicate end point }
      CircleIquad (TroEND [1][1][4],TroEND [1][3][4],2,1);

         { calc the Trans of the end position
           (into nextrans)}
      Nextran (Nextrans, Trb, Tinvrb, Transinit, TroEND,
                    Tinvinitro);

FLAG3:
      STEP (numsteps, Tro, TroEND); { determine the number
                                      of steps }

      for i := 1 to 2 do begin   { calc. incremental step }
        for j := 1 to 4 do begin
          for k := 1 to 4 do
            Tincr [i][j][k] := (Nextrans [i][j][k] -
                                trans [i][j][k]) /
                                 numsteps;
        end;  { j }
      end;  { i }

      T2 := Trans;      { set T2  (an itermediate dummy) }
```

```
for i := 1 to numsteps do begin  { movement loop }

  for j := 1 to 2 do begin       { calc where desired
                                          location T2 is }
    for k := 1 to 4 do begin
      for l := 1 to 4 do
        T2 [j][k][l] := T2 [j][k][l] +
                        Tincr [j][k][l];
    end;  { for k }
  end;  { for j }

   { calc dummy Tro coords for NEXT step }
  px := 0.5*(200+T2 [1][1][4]+400+T2 [2][1][4]);
  py := 0.5*(T2 [1][2][4] + T2 [2][2][4]);
  pz := 0.5*(133.08+T2 [1][3][4]+133.08+
             T2 [2][3][4]);

   { check to see if bar is in obstacle's range }
  if (px > 170.0)  and (px < 350.0) and { enlarged
                                          obstacle }
     (py > -135.0) and (py < 35.0) and
     (pz > 30.0)   and (pz < 95.0) then begin
                                    { not as enlarged }

    gotoXY (2,2); write ('AVOIDING OBSTACLE');

    { increment z coordinate to go up and over }
    Trans [1][3][4] := Trans [1][3][4] + 10.0;
    Trans [2][3][4] := Trans [2][3][4] + 10.0;
    Tro [1][3][4] := Tro [1][3][4] + 10.0;
                             { reset Tro z coord }

    Tro [2][3][4] := Tro [1][3][4];

    InvKin (theta, trans, chars);
    Coordinate (coord,trans,theta);
    DrawFrame (trans, theta, coord);
    DrawObst;
    DrawBox (Tro);

   { indicate end position }
    CircleIquad (TroEND [1][1][4],
                 TroEND [1][3][4],2,1);

    goto FLAG3; { restart loop }
  end;

   { calc trans and draw robots }
  InvKin (theta, T2, chars);
```

```
          CalcTran (trans,theta);
          Coordinate (coord,trans,theta);
          DrawFrame (trans, theta, coord);
          DrawObst;
           { indicate end position }
          CircleIquad (TroEND [1][1][4],
                       TroEND [1][3][4],2,1);

          Tro := trans;  { reset Tro }

          for j := 1 to 2 do begin   { adjust Tro }
            Tro [j][1][4] := 0.5*(coord [1][6][x] +
                                  coord [2][6][x]);
            Tro [j][2][4] := 0.5*(coord [1][6][y] +
                                  coord [2][6][y]);
            Tro [j][3][4] := 0.5*(coord [1][6][z] +
                                  coord [2][6][z]);
          end;  { for j }

          DrawBox (Tro);

        end; { for i }
      end  { else if }

{******** ROUTINE 8 **}

      else if inkey = #127 then begin      { alt-8 key }

           { movement withe intelligent obstacle avoidance }

        DrawObst1; { draw in original obstacle with
                     overhang }

        for i := 1 to 2 do begin      { set final Tro }
          TroEND [i][1][1] :=  0.866;
          TroEND [i][1][2] := -0.5;
          TroEND [i][1][3] := -0.0;
          TroEND [i][1][4] :=  295.0;
          TroEND [i][2][1] := -0.5;
          TroEND [i][2][2] := -0.866;
          TroEND [i][2][3] :=  0.0;
          TroEND [i][2][4] := -40.0;
          TroEND [i][3][1] := -0.0;
          TroEND [i][3][2] :=  0.0;
          TroEND [i][3][3] := -1.0;
          TroEND [i][3][4] :=  160.0;
          TroEND [i][4][4] :=  1.0;
        end;

           { indicate end point }
        CircleIquad (TroEND [1][1][4],TroEND [1][3][4],2,1);
```

```
        { calc the Trans of the end position
          (into nextrans)}
      Nextran (Nextrans, Trb, Tinvrb, Transinit, TroEND,
                    Tinvinitro);

FLAG6:
      STEP (numsteps, Tro, TroEND); { determine the
                                      number of steps }

      for i := 1 to 2 do begin  { calc. incremental step }
        for j := 1 to 4 do begin
          for k := 1 to 4 do
           Tincr [i][j][k] := (Nextrans [i][j][k] -
                                    trans [i][j][k]) /
                                 numsteps;
        end;  { for j }
      end;  { for i }

      T2 := Trans;  { set T2 (an itermediate dummy) }

      for i := 1 to numsteps do begin  { movement loop }

        for j := 1 to 2 do begin  { calc where desired
                                    location T2 is }
          for k := 1 to 4 do begin
            for l := 1 to 4 do
              T2 [j][k][l] := T2 [j][k][l] +
                              Tincr [j][k][l];
          end;  { for k }
        end;  { for j }

         { calc dummy Tro coords for NEXT step }
        px := 0.5*(200+T2 [1][1][4]+400+T2 [2][1][4]);
        py := 0.5*(T2 [1][2][4] + T2 [2][2][4]);
        pz := 0.5*(137.08+T2 [1][3][4]+137.08+
                   T2 [2][3][4]);

        movementx := 0.0; { initialize variables for loop }
        movementy := 0.0;
        movementz := 0.0;

        for j := 1 to 5 do begin
          if (j = 2) then  begin  { Alternate Position 1 }
            px := px + 10.0;
            pz := pz + 5.0;
            movementx := 10.0;
            movementz := 5.0;
          end  { if }

{ Alternate Position 2 }
```

```
            else if (j = 3) then  begin
                                    pz := pz - 15.0;
              movementz := movementz - 15.0;
            end  { else if }

{ Alternate Position 3 }
             else if (j = 4) then  begin
               py := py + 10.0;
               movementy := 10.0;
            end  { else if }

            else if (j = 5) then begin  { stop movement }
              gotoxy (2,2);
              write ('movement frozen');
              exit;
            end;  { else if }

           { check to see if bar is in obstacle's range }
            if (px >= 208.0)  and (px <= 382.0) and
               (py >= -135.0) and (py <= 35.0) and
               (pz >= 66.0)   and (pz <= 135.0) then
              goto FLAG4
                    { enlarged obstacle }

            else if (px >= 208) and (px <= 320) and
                    (py >= -135) and (py <= 35) and
                    (pz >= 8) and (pz <= 72) then goto FLAG4

            else begin
              if (j = 1) then goto FLAG8
              else goto FLAG5;
            end;  { else }
FLAG4:
          end;  { for j }

FLAG5:
          gotoXY (2,2); write ('AVOIDING OBSTACLE');

           { incr trans to intelligent movement }
          Trans [1][1][4] := T2 [1][1][4] + movementx;
          Trans [2][1][4] := T2 [2][1][4] + movementx;
          Trans [1][2][4] := T2 [1][2][4] + movementy;
          Trans [2][2][4] := T2 [2][2][4] + movementy;
          Trans [1][3][4] := T2 [1][3][4] + movementz;
          Trans [2][3][4] := T2 [2][3][4] + movementz;

          InvKin (theta, trans, chars);   { draw robots }
          Coordinate (coord,trans,theta);
          DrawFrame (trans, theta, coord);

          Tro := trans;  { reset Tro }
```

```
        for j := 1 to 2 do begin  { adjust Tro }
          Tro [j][1][4] := 0.5*(coord [1][6][x] +
                                coord [2][6][x]);
          Tro [j][2][4] := 0.5*(coord [1][6][y] +
                                coord [2][6][y]);
          Tro [j][3][4] := 0.5*(coord [1][6][z] +
                                coord [2][6][z]);
        end;  { for j }

        DrawObst1;
        DrawBox (Tro);
         { indicate end position }
        CircleIquad (TroEND [1][1][4],
                     TroEND [1][3][4],2,1);

        goto FLAG6; { restart loop }


FLAG8:
        InvKin (theta, T2, chars);  { draw robots }
        CalcTran (trans,theta);
        Coordinate (coord,trans,theta);
        DrawFrame (trans, theta, coord);
        DrawObst1;
         { indicate end position }
        CircleIquad (TroEND [1][1][4],
         TroEND [1][3][4],2,1);

        Tro := trans;  { reset Tro }

        for j := 1 to 2 do begin   { adjust Tro }
          Tro [j][1][4] := 0.5*(coord [1][6][x] +
                                coord [2][6][x]);
          Tro [j][2][4] := 0.5*(coord [1][6][y] +
                                coord [2][6][y]);
          Tro [j][3][4] := 0.5*(coord [1][6][z] +
                                coord [2][6][z]);
        end;  { for j }

        DrawBox (Tro);
      end; { for j }

    end  { else if }

    else
      goto FLAG1;  { default if wrong alt code is entered }

    goto FLAG1; { after a routine is finished }

end.  { CoordSim }
```

APPENDIX 2

USER'S GUIDE TO COORDSIM

A2.1 Introduction

The following is a user's guide which describes the use of the program "CoordSim". CoordSim is a computer simulation written in TurboPascal (refer to appendix 1) which graphically illustrates the coordination of two PUMA robots.

Figure 6.1 is a sample output of the screen. It shows the location of the rotation matrices, joint angles, and absolute positions of each robot along with a stick figure which represents the center-line of the robot

A2.2 Starting the Program

1) Boot up the IBM PC with DOS version 2.0 or later.

2) Place the floppy disk containing the program "CoordSim.com" in drive A.

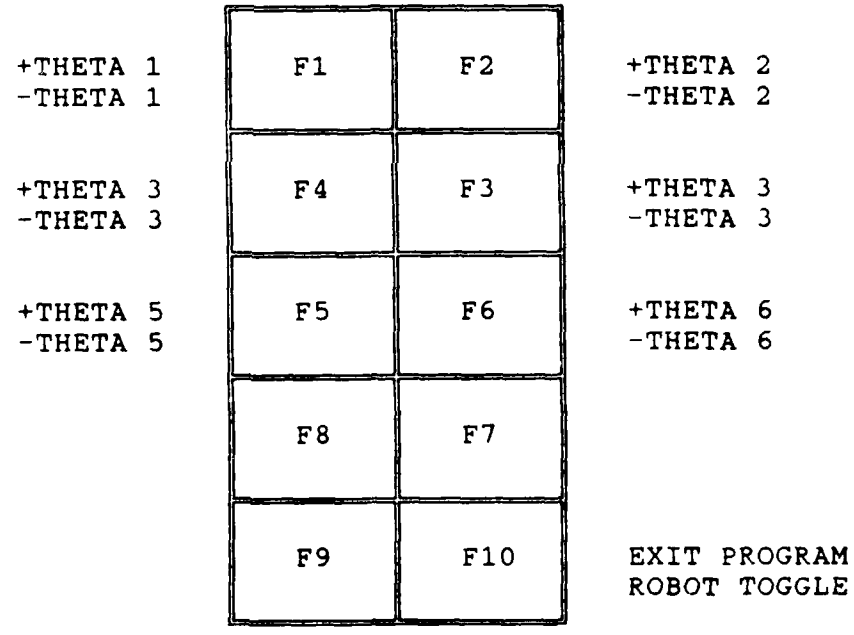
3) At the "A>" DOS prompt type "CoordSim".

4) The program automatically loads and initializes the robots' locations to a home position.

5) Keyboard input is now available as per the following sections.

A2.3 Teach Pendant

CoordSim has the ability to control a variety of motions of each robot. Initially, the keyboard controls robot 1. The first six function key control the six joint angles of the robot (refer to figure A2.1). Pressing F1 increases the theta 1 by 10 degrees, F2 increases the theta 2 by 10 degrees and so on up to F6 for theta 6. A decrease in angle is obtained by pressing ALT-F1 for theta 1, ALT-F2 for theta 2 and so on up to ALT-F6 for theta 6. The toggle to pass control to the other robot is the ALT-F10 key.



* Press function key alone to perform the upper function.
* Press ALT-function key for lower function.

Figure A2.1
Function Key Identification

Along with joint angle control is cartesian movement control. In this case, the numeric keypad controls the cartesian movement of the end effector. At the keypad, the "7" and the "9" move the end-effector in the negative and positive x-direction. The "4" and "6" control the negative and positive y-direction. Finally, the "1" and "3" control the negative and positive z-direction.

A2.4 Routines

The eight routines described in chapter 6 are started by pressing the ALT key along with the number of the specific routine desired. For example, routine 4 would be started by pressing the ALT-4 key. Each of these routines can be started at any time while in CoordSim and the teach mode is still effective after a routine is completed.

A2.5 General Information

The program is exited by pressing the F10 key.

END

DATE

FILMED

5-88

DTIC