University of Manitoba

Winnipeg

An Efficient Collision Detector for Automatic Path Planning for Manipulators

Presented to the University of Manitoba in partial completion
of the requirements for a Master of Science degree
in Electrical Engineering

by

Gordon D. Sawatzky

ISBN 0-315-33860-1

AN EFFICIENT COLLISION DETECTOR FOR AUTOMATIC PATH PLANNING FOR MANIPULATORS

BY

GORDON D. SAWATZKY

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

ABSTRACT

In this thesis an efficient approach to collision detection is proposed. Also presented is the incorporation of this collision detector in two methods for automatic robot path planning. The collision detector is based on representing objects by a hierarchy of bounding spheres. Features of the collision detector are: one, that computation time depends on the proximity of the objects, and two, that the method concentrates its efforts on the parts of objects most likely to have geometric interference as determined by collision information.

ACKNOWLEDGEMENT

I would like to express my gratitude to the Systems Laboratory of the NRC (National Research Council of Canada) in Ottawa, Ontario for having sponsored my graduate thesis work. I am especially grateful to Dr. Hany I. El-Zorkany of the NRC for the opportunity to have pursued my thesis work at NRC. Dr. El-Zorkany suggested the thesis topic and guided and encouraged me from beginning to end of the work.

I am indebted to Dr. Steve Onyshko of the University of Manitoba. Dr. Onyshko was my advisor throughout my graduate studies at Manitoba as well as while working at NRC.

Also special thanks to John Buckley and Ramiro Liscano, fellow researchers at the National Research Council of Canada whose discussions of the problem were very helpful in producing this work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The desire of humans to have machines that aid them in dangerous, tedious or superhuman tasks has existed for a long time. The use of hand tools to process food, build shelters and kill animals preceded the advent of power tools. Animals, water, wind, and fire were utilized as power sources for machines that extended the ability of humans to perform work. Use of external power sources requires a method of control. The first type of control systems were supervisory control. For example, the inputs to an ox pulling a plow have to be determined, or the sails have to be set on a sailboat. With the advent of mechanical and electrical feedback control systems, fully automatic machines were possible. Larger scale tasks that could not be performed previously such as mass transportation, large scale manufacturing or large scale processing are now possible. Hybrid control systems consisting of a human supervisor and an array of feedback control systems enabled one to accomplish very complex tasks and increase one's productivity. The introduction of digital computers into modern control systems dramatically increases the flexibility and capability of machines. Computers are present in the low level control loops and in the higher level supervisory control decisions. One of the costs for the added flexibility of machines controlled by computers is programming. The development of computers which automatically program machines is required to increase the level of automation. Such computers will require extensive "world" knowledge to complete their task.

Today a vast array of machines, tools and power sources are available to achieve new goals. Mechanical manipulators coupled with computer control i.e. robots, are one of the more recent developments available. The term robot, which had its origins from the play "Rossum's Universal Robots" by the

Czech dramatist Karel Capek in 1921, describes a set of machines. The size of this set depends upon the particular definition of robot. One possible definition states that a robot is a mechanical manipulator whose degrees of freedom are computer controlled. This definition is very broad and would include many devices which may not be commonly considered as robots. One can further reduce the number of devices considered to be robots by adding the clause that the devices must be used to manipulate objects and that the devices be programmable.

1.1. Task level Robot systems

A relatively recent topic of research is the development of task level robot systems. While the definition of a task level robot system varies, such systems are typically characterized by being told what to do rather than being told the details of how to execute a task. This differs from supervisory robot systems which require a constant human presence in guiding the robot through its task. It also differs from programmable robots, where a human writes a program which contains the motion commands that enable a robot to complete a task. A block diagram of a potential task level system is shown in Figure 1.1. The block structure of a task level system will vary according the the class of tasks to be performed by a particular robot. Generally, a task level robot system can be divided into planning and control subsystems.

The planning subsystem translates a task, given as object oriented commands, into a suitable robot strategy. This strategy, which may be expressed as a robot program, is developed with the aid of a motion planner which ensures that the particular robot can safely execute the required motions. To develop a robot strategy, the planning subsystem requires world information in the form of environment data and/or rules about motion strategies. Geometric and location data of objects may be acquired from automatic sensors or human input.

At the lowest level of robot control systems are the servo controllers which require position, velocity and acceleration feedback of joint values. At the next level, trajectory generators generate set points for the servo feedback loops. Generally, trajectory generators require kinematic and trajectory models of the robot to calculate set points for the servo control loops. Some robot controllers allow for interfaces with vision, proximity, and/or force sensors. Robot motion will be executed depending on the state of the particular

TASK LEVEL ROBOT SYSTEM

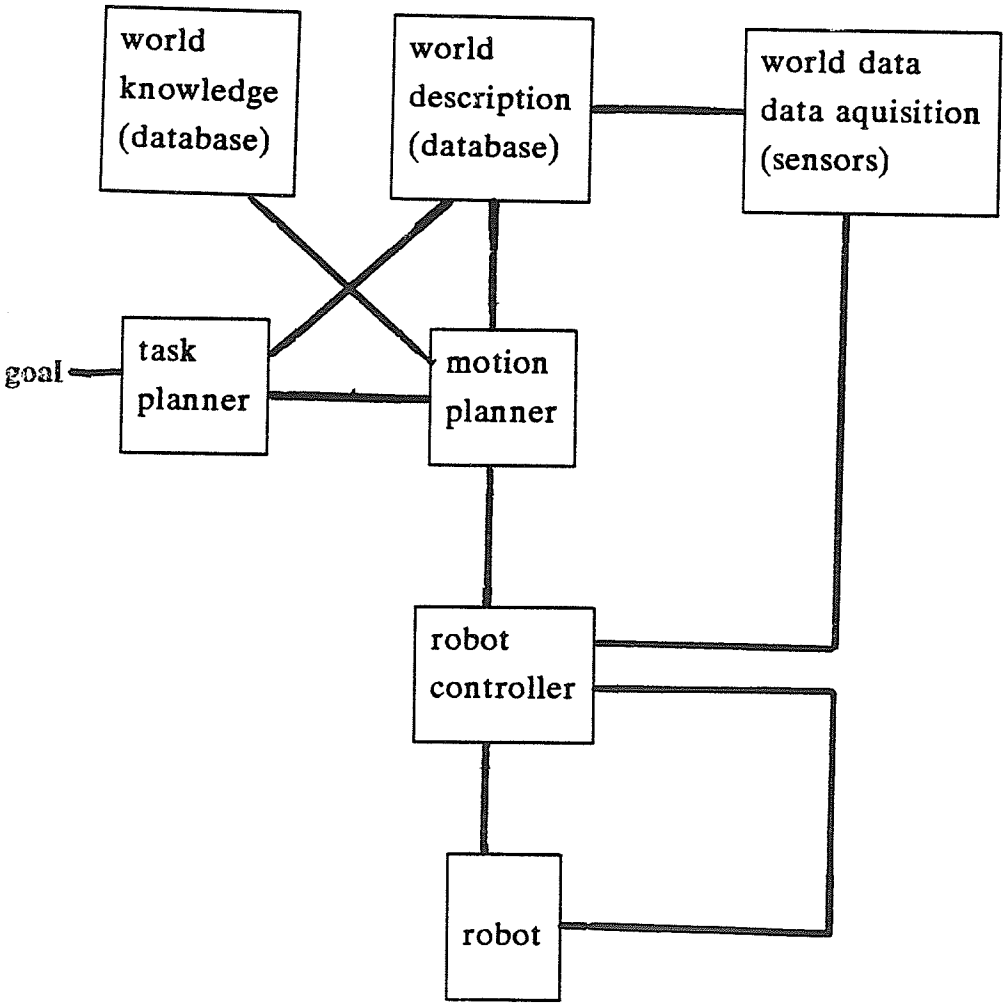


Figure 1.1 : Block diagram of a task level robot system

sensors.

1.2. Present Robot Programming Techniques

The programming of robots requires writing a program of motion commands and defining "robot locations", which describe a position and orientation of a robot end effecter. Presently, robot manufacturers have their own programming languages which vary in complexity. Programming constructs such as loops, conditionals, and subroutines are available in many languages. For a survey of robot programming languages the reader is referred to [3].

Often, generating a robot program is not difficult or time consuming compared to defining the robot locations. One of the most common ways to define robot locations is to manually specify the desired locations using a robot motion "teach pendant". Location data are specified by moving the robot to each location and recording the location data. In doing so the programmer also ensures that safe motions will result by including any intermediate or "via" points as required. One advantage of this method of programming is that the resulting robot motion is a "playback" of what is programmed. On the other hand, some disadvantages of this "teach through showing" approach are that it is often time consuming, possibly unsafe because it is done on the spot using the robot itself, and costly because the long programming time required disrupts the production process associated with the robot.

Off-line programming, where location data are obtained from a CAD (computer aided design) system, can overcome some of the difficulties mentioned above, however it poses some problems. One such problem is the discrepancy between the kinematic, trajectory, and geometric models of the robot, that are used in the programming phase, and the actual robot. Another problem is the need for the operator to visually determine safe robot motions from a two dimensional display of the robot and its environment. Safeness of robot motions could, however, be verified by testing for collisions in a proposed robot motion, using suitable models of the robot and its environment. To this end, a collision detector, which would be used as an aid to an off-line robot programmer, is proposed.

A more attractive possibility is to automatically generate safe motions between given task locations. Such a capability will be more useful as robot tasks grow in complexity, for example, in automated assembly. In such a system it is desired to program by specifying what is to be done rather than

giving the details of how it is to be done. Evidently, among other things, this calls for the ability to find collision free paths for the robot and its payload.

An important question to ask is the need for automatic path planning and the type of applications that will benefit most from such a development. Programming of existing robot applications, such as pick and place and contour following, would benefit from an automatic path planner. Future robot applications that will benefit from automatic path planning would include remote applications such space and sea, and harsh environments such as nuclear and chemical plants.

1.3. Scope of Thesis

The find-path problem can be defined as follows. Given two robot locations, which are sometimes referred to as configurations, find a collision free path between them or report that none exists.

Automatic path planning between two robot configurations can be considered with or without knowledge of objects in the robot's environment. This thesis will consider the case of the known environment. The solution of these two cases of path planning would likely be very different. The former dealing with a world description in a database and the latter dealing with real time interpretation of sensor information.

In this thesis an approach is proposed, implemented and tested for representing objects for purposes of collision detection. Also presented is the use of this collision detector in two approaches to the find-path problem. In particular, a generate and test approach is implemented on a PUMA 560 robot, and an approach previously presented by Gouzenes [9,10] is implemented and tested for examples of two dimensional robots as well as for the first three links of a PUMA 560 robot.

In order to develop and test collision detection algorithms as well as find-path algorithms, a general and flexible robot simulator was implemented. LISP was chosen as the development language in order to quickly build a prototype simulator. When a LMI LAMBDA LISP Machine became available, this proved to be a very productive programming environment. For future developments, this system can serve as an excellent environment to quickly try new ideas, algorithms, and models in robotics. To evaluate the algorithms and models developed, it was necessary to develop graphic output for illustration of the two dimensional experiments. A serial communication interface

between the LISP machine and the PUMA 560 robot was developed for testing the three dimensional models and find-path algorithms.

The thesis is organized as follows. First, a review of the literature on the robot "find-path" problem is presented. Details are given on the proposed object representation and the associated collision detector. Thirdly, incorporation of the proposed collision detector in two approaches to the solution for the find-path problem are presented. A discussion of the implementation of the collision detector, implementation of two find-path methods, and the results of experiments performed in two dimensions as well as experiments in three dimensions is included. Finally, potential extensions, a summary of what was done, and conclusions are presented.

In Appendix A, an overview of the robot simulation environment (PPLAN), which was developed for this thesis is presented. Appendix B is a manual containing brief descriptions of major functions in PPLAN and their usage. Source listings of PPLAN are available from the author upon request.

CHAPTER 2

Path Planning

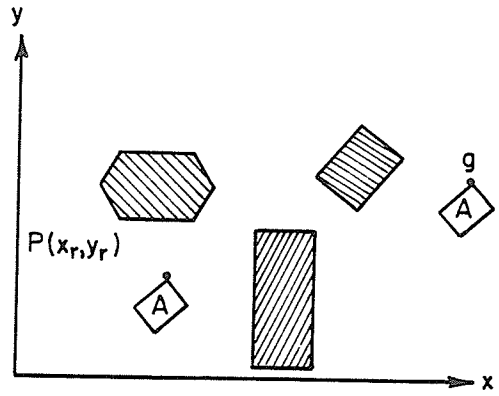
Before beginning a survey of find-path research, it is necessary to define terminology commonly used in the area.

2.1. Configuration space and free space

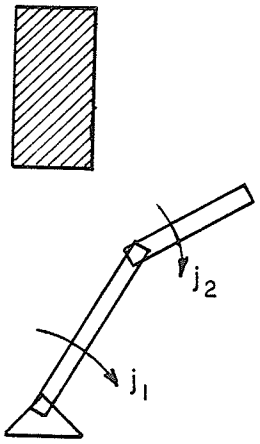
Configuration space of a robot, which will be referred to as C-space, is the set of all robot configurations. For an open link robot of n links, this space is n-dimensional and is formed by the Cartesian product,

$$C-space = j_1 \times j_2 \times j_3 \times \cdots \times j_n$$

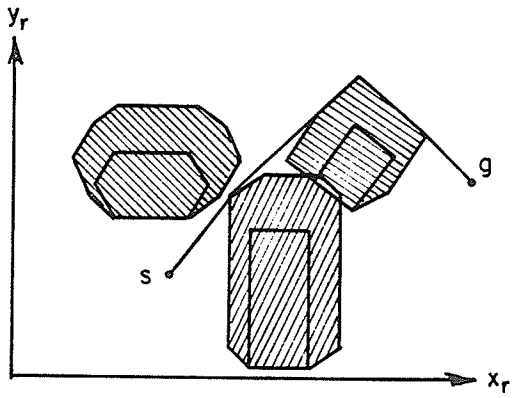
where j_i is the range of values for the i^{th} link parameter. The free space of a robot, which will be referred to as F-space, is a subset of C-space and is defined by all configurations of the robot for which the robot does not collide with other objects. F-space is the complement of the set of all points occupied by the obstacles in C-space. Any robot configuration is represented by a point in its C-space. Accordingly, in C-space the path planning problem becomes the simpler one of finding a continuous path for a point that is completely contained in F-space. Unfortunately, accurate representation of F-space for six degree of freedom robots is a rather difficult problem. This is a manifestation of the fact that obstacles with simple geometry in Cartesian space can become very complex shapes in C-space. Figure 2.1a shows that polygonal obstacles in Cartesian space become polygonal obstacles in C-space of an object which can only translate in the x or y direction. Figure 2.1b shows the complex shape of a simple Cartesian obstacle in the the C-space of a serial link robot which has



Cartesian space

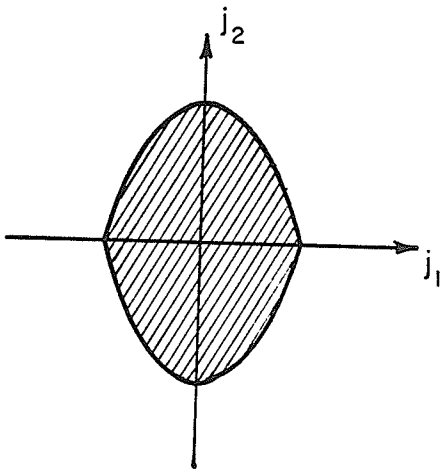


Cartesian space



C-space

a.



C-space

b.

Figure 2.1 Obstacle representation in C-space

two rotational joints.

2.2. Free Space Representation

F-space of robots can be represented in many ways. Each method of representation allows for different path finding algorithms. For example, F-space can be represented as a connected graph. In one representation, each node in the graph represents a region of F-space and each arc indicates adjacent regions. Characteristics of such regions should be such that movement within a region is trivial, determining adjacency of regions be easy, and movement between adjacent regions can be determined. In a second approach, each node represents a safe configuration and each arc represents a safe trajectory. For both approaches the solution for path planning then becomes a much simpler problem of finding a path between two nodes of a connected graph [21].

2.3. Find-path Survey

The find-path problem in robotics refers to the problem of generating a safe robot path between an initial and a final robot configuration. Schwartz and Sharir [19,20], determined an algorithm to solve the find-path problem which has computational time bounds that are exponential in terms of the degrees of freedom of the manipulator and polynomial in terms of geometric complexity. Computational time bounds are mathematical expressions which show how the computation time for an algorithm increases with the size of the problem. The algorithm was never implemented because of its apparent complexity. It is based on representing F-space with a graph of adjacent F-space regions generated by Tarski Sets. The complexity of the problem has directed recent research efforts to use approximations and to exploit available constraints to produce practical solutions. For example, Brooks [4,5] used the concept of "freeways" in C-space for moving polygons with rotation among polygons, constraints on allowed payload rotations, and motion decoupling between the upper and lower arms to obtain a solution for a PUMA robot for pick and place operations.

One of the simplest approaches to the find-path problem is the generate and test method. The basic idea is to hypothesize a path, say the shortest path to the destination, and then to determine if it is safe. If the path is safe it is taken as the solution, otherwise, a new path is proposed and the process

repeated. One way of generating a new path is to use information from the detected collision to identify a safe intermediate location and start the algorithm with the intermediate location as the destination. Of course, there is no guarantee that a path could be found. Furthermore, if one is found it is merely feasible, as opposed to an optimal collision free path. Moreover, this method is characterized by a local view. That is to say that such approaches consider only the objects colliding and do not take other objects into account when proposing a new path. Myers [14] proposed such an approach for simple tasks such as palletizing. A collision detector is a basic ingredient of such an approach and an efficient collision detector is a definite asset.

Find-path solutions with a global point of view tend to include a representation of F-space. Lozano-Perez [11,12,13], and Brooks and Lozano-Perez [6] transformed obstacles from their Cartesian representation into their C-space representation from which it was possible to obtain a representation of F-space. Gouzenes [9,10], in principle, discretized C-space with an n-dimensional grid, where n is the number of degrees of freedom of the robot, and used a collision detector to test at the nodes to generate a representation of F-space. Actually, collision detection is not performed at all grid points of C-space. Since swept volumes of robot links represent large regions of C-space, collision detection with swept volumes of robot links reported on large regions of C-space. This reduced the number of tests required but not necessarily their complexity. Even with the above advantage, the maximum number of collision checks grows exponentially with the number of degrees of freedom of the robot being considered, hence, an efficient collision detector is important in an implementation of Gouzenes' approach.

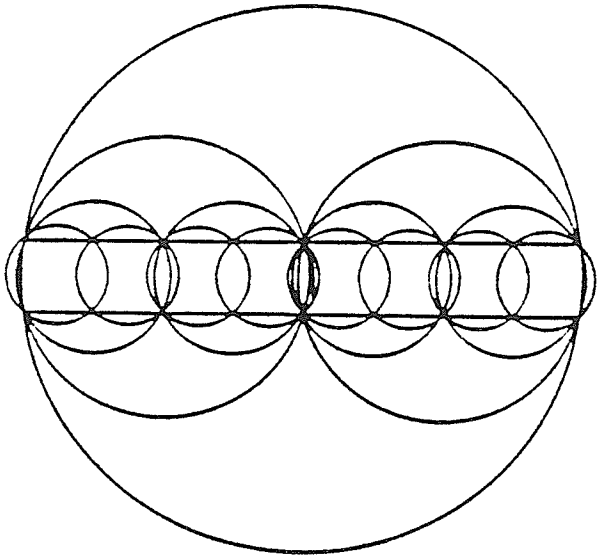
CHAPTER 3

Geometric representation for interference detection

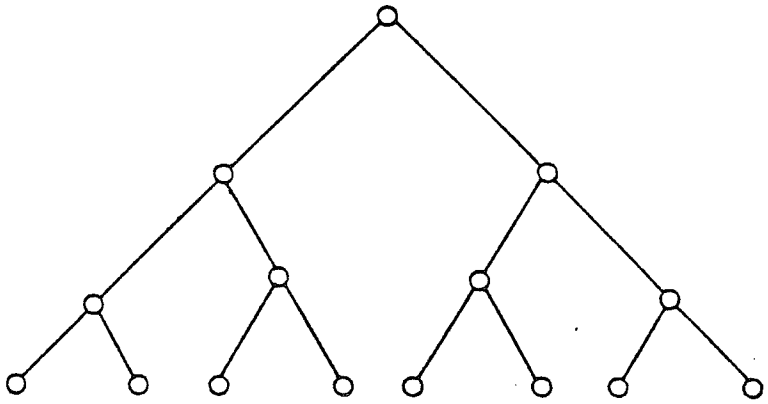
The choice of a geometric representation should reflect the goals and nature of their intended use. In the case of CAD, geometric accuracy is a predominant consideration since the models will be used in various engineering analyses such as stress analysis or for program generation in computer aided manufacturing. For interference analysis between objects in path planning, and in particular for gross motion planning, high precision of object representation is not required, because errors in object location and robot movement do not justify the cost of a precise representation. In fact, conservative representations are more advantageous as they provide some margin of safety. What is most often required is fast computation of interference detection rather than geometrical precision.

3.1. Proposed geometric representation scheme

There are many ways of representing three dimensional objects in a data structure. Some of the well known representation techniques are spatial occupancy, constructive solid geometry, swept volume, and boundary representations [18]. For purposes of interference detection, a representation is proposed where all objects, whether they represent robot links, swept volumes, obstacles or collections of obstacles are represented by a hierarchy of spheres. For illustration, consider the rectangular object in Figure 3.1a and its representation by a hierarchy of spheres (circles in two dimensional space) shown in Figure 3.1b. The union of circles from each level of the hierarchy bounds the object being modelled. At each node, the original object is divided and the resulting "sub-objects" are separately bounded by smaller circles. These two facts allow for effective interference detection between two objects



a. rectangular object and spheres



b. hierarchy of spheres

Figure 3.1 Hierarchy of spheres for an object

by comparing their hierarchical representations. Two objects are said to interfere if any of the leaves of their representations intersect. Two objects do not interfere if a set of spheres from one object, which bounds that object, does not interfere with any spheres from a set of spheres which bound the second object. Interference detection between spheres is done by comparing the distance between their centers and the sum of their radii. This is done very easily for both two dimensional and three dimensional spheres.

3.2. Proposed collision detector

Given the geometric representation of objects by a hierarchy of spheres, where the hierarchy is a binary tree, the collision detector called interferep-obj, a predicate function which reports on two objects x and y, proceeds as follows. If the most crude representations of x and y are not interfering, then x and y are not colliding and interferep-obj stops and reports appropriately. Otherwise, interferep-obj is applied to the more precise representations of x and y. If x and y have no more precise representations, then x and y are said to collide. For readers with a LISP background, the following is a description of the collision detection algorithm in "Pseudo LISP".

Given : x and y are binary tree representations of two objects
(interferep-sphere s1 s2) is a predicate function which
returns t if s1 and s2 collide, otherwise, nil is
returned.
s1,s2 are spheres defined by a center and radius

Using the basic functions of LISP

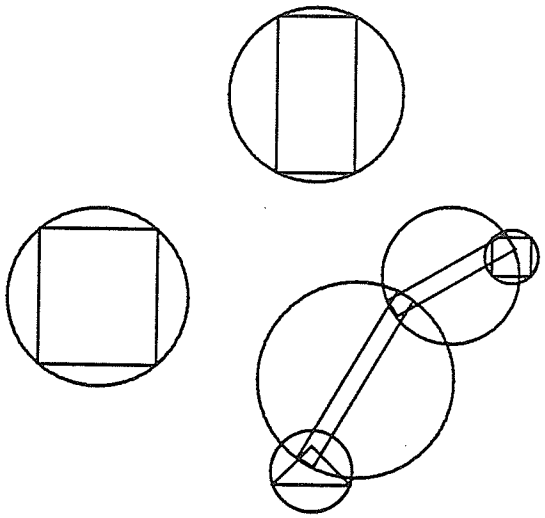
(car a) returns first element of a
(cadr a) returns second element of a
(caddr a) returns third element of a
(null a) is a predicate function which determines
if a is undefined
(cond (p1 f11 f12 ...)
(p2 f21 f22 ...)
(p3 f31 f32 ...)
.
.
(pn fn1 fn2 ...))
is a function which evaluates the forms after
a true predicate is found and returns the value
of the last form evaluated
; indicates a comment

The algorithm proceeds as follows

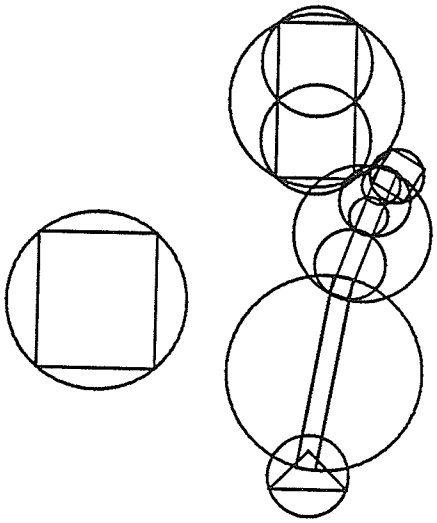
```
(interferep-obj (x y)
  (cond ((null x) ())   ; no representation for x ? return nil
        ((null y) ())   ; no representation for y ? return nil
         ; check crudest representations of x and y for
         ; interference
        ((interferep-sphere (car x) (car y))
          ; crudest representations are colliding,
          ; more precise representations are
          ; checked if any exist
          (cond ((null (cadr x)) ;more precise representation
                                 ;for x ?
                 (cond ((null (cadr y)) t) ;more precise
                                           ;rep. for y ?
                                           ;return t
                       ; otherwise check x against more
                       ; precise representation of y
                       (or (interferep-obj x (cadr y))
                           (interferep-obj x
                                           (caddr y))))))
                (t (cond ((null (cadr y)) ;more precise rep.
                                          ;for y ?
                         ; check y against more precise
                         ; representation of x
                         (or (interferep-obj (cadr x) y)
                             (interferep-obj (caddr x) y)))
                        ; check more precise representation of
                        ; x against precise representation of y
                        (t (or (interferep-obj (cadr x)
                                               (cadr y))
                               (interferep-obj (cadr x)
                                               (caddr y))
                               (interferep-obj (caddr x)
                                               (cadr y))
                               (interferep-obj (caddr x)
                                               (caddr y))))))))
          (t nil))) ; no collision so return nil
```

3.3. Algorithm Properties

Figure 3.2 illustrates the circles used in computing interference detection between a two dimensional robot and its environment when the robot is far from and near to obstacles. Comparison of these two cases highlights many properties of the interference detection algorithm. Note that the algorithm can terminate after comparing the top level spheres of each object when they are sufficiently far apart. Accordingly, computation time for interference



a. far



b. near

Figure 3.2 Interference detection between objects

detection between two objects will depend on their proximity. Also, notice that the entire representation of objects is rarely used because the interference detection algorithm "homes in" on potential collisions without considering the entire object. This idea can be extended by grouping collections of small objects into a single region and considering the objects enclosed only when necessary. In the example, one can imagine that the obstacles are really a collection of smaller objects. To restate the above properties, one can say that the algorithm and representation technique allow computation to occur only when necessary. For the above reasons a hierarchy of spheres can be a good geometric representation for purposes of interference detection between objects.

3.4. Comparison

A comparison between geometric interference detection algorithms that employ different object representations is difficult. Two important factors that should be considered in such a comparison are computational speed and precision of spatial representation. A simple comparison of two object interference algorithms can be made as follows. Consider, two methods R_1 and R_2 of representing objects. Let R_1 be a non-hierarchical representation, and let R_2 be a hierarchical representation. Associated with each method of representation is an interference detection algorithm, namely I_1 and I_2 which have computation times t_1 and t_2 respectively. Consider, performing interference detection between two objects A and B n number of times with both I_1 and I_2. Let d(A,B) be a distance function for the objects A and B. Assume that d(A,B) has a probability density function shown in Figure 3.3. Assume that t_1 and t_2 depend on d(A,B) as shown in Figure 3.4. In the above thought experiment, let T_1 and T_2 be the total computation times for I_1 and I_2 respectively. T_1 will not depend on d(A,B), whereas T_2 will be strongly dependent on d(A,B). If the shape of the object distance probability density function is favourable then R_2 and I_2 is the better representation and the better algorithm to use.

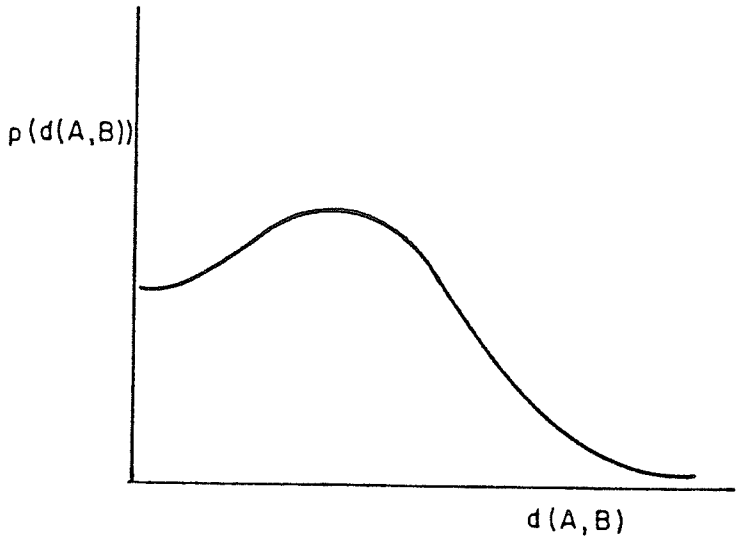


Figure 3.3 Probability density function for distance between objects A and B

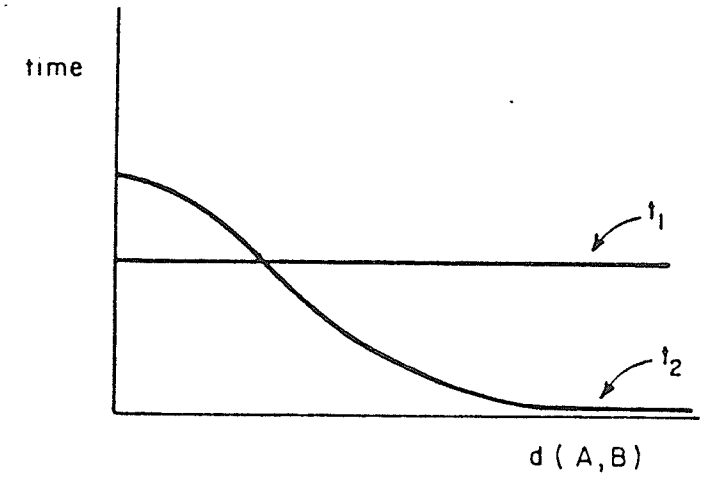


Figure 3.4 Computation time for interference detection algorithms

3.5. Generating representation

Currently, a programmer determines the spheres used to represent an object with the aid of algorithms which generate hierarchies of spheres from simple shapes. For example, one such algorithm returns a collection of spheres, organized as a binary tree, as a geometric representation for any rectangle. Figure 3.1 shows the binary tree of spheres generated for a rectangle. The algorithm works by finding a bounding sphere for a rectangle and then recursively sub-dividing that rectangle until the length to width ratio of the new rectangle is less than the square root of two. A similar algorithm was implemented for three-dimensional boxes with trapezoidal cross-sections. As an example of the modeling capability of a hierarchy of spheres, consider, the most precise representation of a representation used for a PUMA 560 robot shown in Figure 3.5. In the future, automatic generation of a good representation by bounding hierarchy of spheres from a CAD object representation would be a very useful tool.

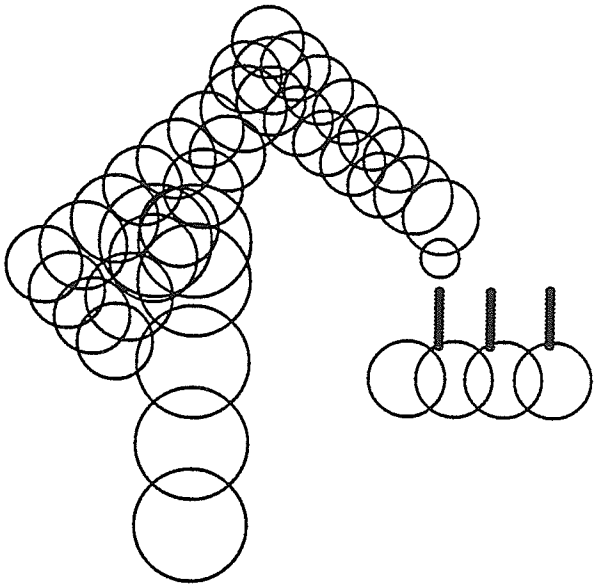


Figure 3.5 Representation of PUMA with spheres

CHAPTER 4

The use of the collision detector within find-path solutions

This chapter discusses the incorporation of the proposed geometric representation and the associated collision detector in two find-path solution methods. The first is a generate and test approach and the second is based on representation of F-space. In both methods collision detection is a fundamental part, and hence there is potentially a significant advantage in using an approach that is computationally efficient.

4.1. Generate and Test

Solution to the find-path problem with a generate and test approach requires an ability to check the safeness of a robot trajectory and to propose intermediate robot locations if a trajectory is unsafe. One technique of checking a trajectory is to determine the geometric volume that the robot sweeps out as it executes the trajectory and determine if this volume collides with obstacles. For robots with rotational joints this method is difficult because the volumes swept by robot motions are complex shapes, and hence difficult to represent. A second, simpler approach discretizes a robot trajectory and tests for safeness of each intermediate robot configuration. The size of the discretization steps should be small enough to reduce the chance of missing a collision. Evidently, in choosing a step size, there is a trade-off between the probability of missing a collision and the amount of computation. Alternatively, to guarantee a safe trajectory, robot links can be enlarged to bound the volume swept by all robot configurations in a discretization interval. An associated penalty, however, is that enlarging the robot may eliminate legitimate trajectories. For either technique an efficient collision detector is desirable.

When checking the safeness of robot trajectories, reduction of the number of collision checks can be achieved by using a variable number of checks for each link. For example, the end effecter of a serial link manipulator will often move much larger distances than the first link. In such a case the collision detector should dynamically determine the number of checks for each link depending on the distance travelled by each link.

The second aspect of the generate and test solution is to propose intermediate robot configurations when collisions are detected. For each particular robot the rules and heuristics used will vary according to the application. For example, one can choose to limit the choices for collision avoidance with a PUMA 560 robot to three strategies. Given that a collision is detected, an attempt to move over, inside, or outside of the obstacle that is in collision with the PUMA payload can be considered. An example of such an approach is presented in the next chapter.

4.2. F-space Representation

Global find-path solutions generally use a representation of F-space, which can be described by a collection of regions formed from the cells generated by imposing an n dimensional grid on C-space, where n is the number of degrees of freedom of the robot. Clearly, the number of cells in an n dimensional grid grows exponentially with the degrees of freedom of the robot. For example, if each dimension of C-space is divided into 32 steps then C-space for a three degree of freedom robot would consist of about 32,000 cells, and the number of cells contained in C-space for a six degree of freedom robot increases to about 1,000,000,000 cells. Obviously, checking each cell of C-space individually for robot safeness and storing only the cells that are part of F-space can be very demanding, especially for robots with more than three degrees of freedom. If however it is possible to effectively check large regions of C-space, the work required to determine F-space will be reduced. The procedure would be to check a large region of C-space, and if it is found to be part of F-space it is stored, otherwise, it is broken down and investigated further. Such a method has two benefits, one being that the number of checks is reduced and secondly that the number of regions used to represent F-space is less. The number of F-space regions can be further reduced if adjacent regions can be combined into larger regions of a similar shape. For example, the combined region should still be convex, otherwise the

regions should not be combined. Fewer regions of F-space will result in faster searches for paths contained in F-space.

As mentioned earlier, Gouzenes [9,10] recently proposed a method of obtaining a representation of F-space that is efficient in terms of the number of checks required to determine F-space and the number of regions used to represent F-space. Since collision detection is a major requirement, improvement in the performance of this method may be obtained by improving the computational efficiency of the collision checks required. The general versions of the algorithms to implement Gouzenes' method are briefly described and then illustrated by a simple example below. The reader is referred to [9,10] for further details.

Gouzenes' find-path solution contains four steps:

1 Test regions of C-space and if free store in a tree structure

2 From this tree structured representation of F-space find adjacencies of regions and construct a graph of connected regions.

3 Search the resulting graph for a connected sequence of regions from the region containing the initial robot configuration to the region containing the final robot configuration.

4 Generate a robot trajectory contained in the sequence of regions found.

This method is illustrated by way of a simple two dimensional example. Figure 4.1 shows an object which can move in the x or y direction in an environment with three stationary obstacles. Figure 4.2 shows the cells of F-space that are generated for very large discretization steps in x and y. The F-space representation is obtained by checking a large rectangle which bounds the volume swept out by the object for all values of y, for any x position of the object. This rectangle is checked for safeness at a series of positions along the x axis. If the large rectangle is free from collisions, it is stored as a region of F-space, otherwise the object is checked for collisions at a series of positions along the y axis. While checking the object along y, adjacent regions of F-space are combined so that the final regions are as large as possible. Clearly, the quality of the representation of F-space is determined by the size of the discretization steps used. This explains the very rough representation of F-space in Figure 4.2. Figure 4.3 shows the tree structured organization of F-

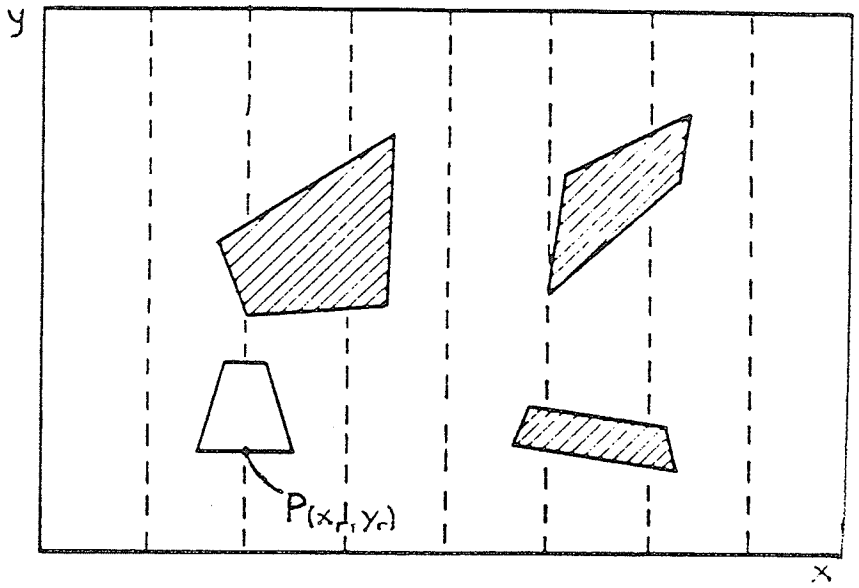


Figure 4.1 Moving object and obstacles

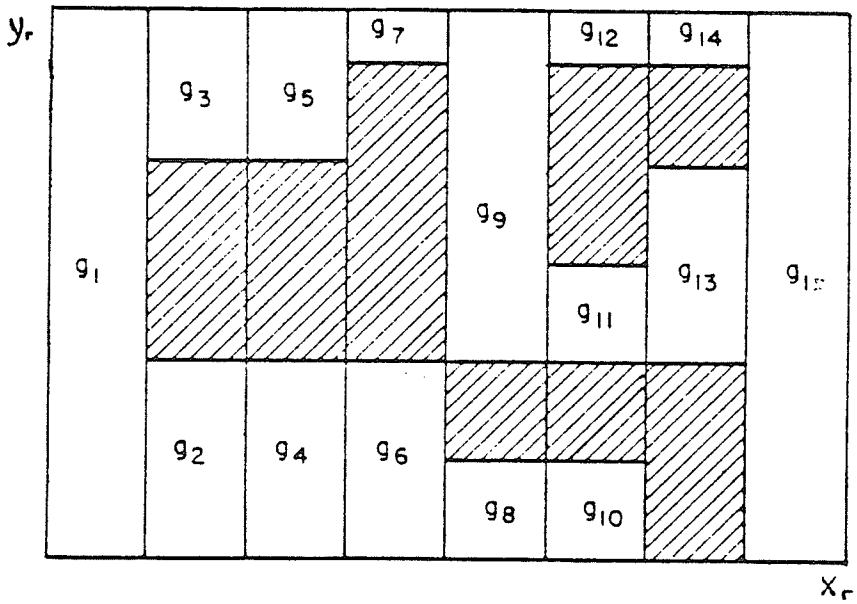


Figure 4.2 C-space representation of moving object

space regions. Regions of F-space are completely defined at the leaves of this tree, while each branch indicates a range of values for the object's reference point position. A cell is defined by the branches followed back to the root of the tree and by the complete range of positions not specified. This tree organization of the regions of F-space is used to aid the search for adjacent regions and later to find which region a given robot configuration is a member of.

Figure 4.4 shows the final graph of F-space regions for the mobile object of Figure 4.1. Each node in the graph represents one region of F-space. Each arc in the graph indicates adjacent regions, where adjacent means that the region boundaries overlap or touch at least at one point. From such a connectivity graph of F-space regions, it is relatively simple to find a feasible path between any two object positions, if one exists.

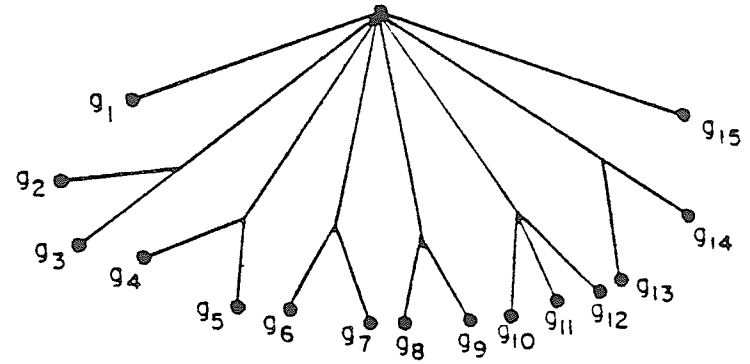


Figure 4.3 Tree representation of F-space

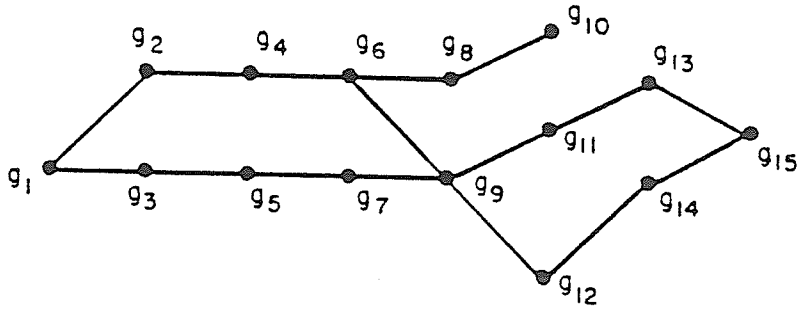


Figure 4.4 Graph of connected F-space regions

CHAPTER 5

Implementation details and experimental results

This chapter contains a description of the implementation of the find-path method of Gouzenes [9,10]. This description includes algorithms for: finding F-space regions, finding adjacency of F-space regions, and finding robot paths from an F-space representation. The remainder of this chapter is organized as follows. A description of the developed robot simulator is given. Next, all experimental results are described for both two-dimensional and three-dimensional examples. Finally, extensions of the algorithms developed are identified.

5.1. General Algorithms for Gouzenes' method

The method of generating a F-space representation outlined by Gouzenes [9,10] was implemented. The algorithm is the same as described by Gouzenes except that objects and virtual swept objects are represented by the proposed hierarchy of spheres. The goal of the algorithm is to check all configurations of a robot for geometric safeness and build a representation of F-space. Provided that a representation of a robot and obstacles in its environment exists, and that a collision detector that operates on the chosen object representations exists, the algorithm for finding and storing regions of F-space can be described as a recursive algorithm expressed in "Pseudo LISP", as follows.

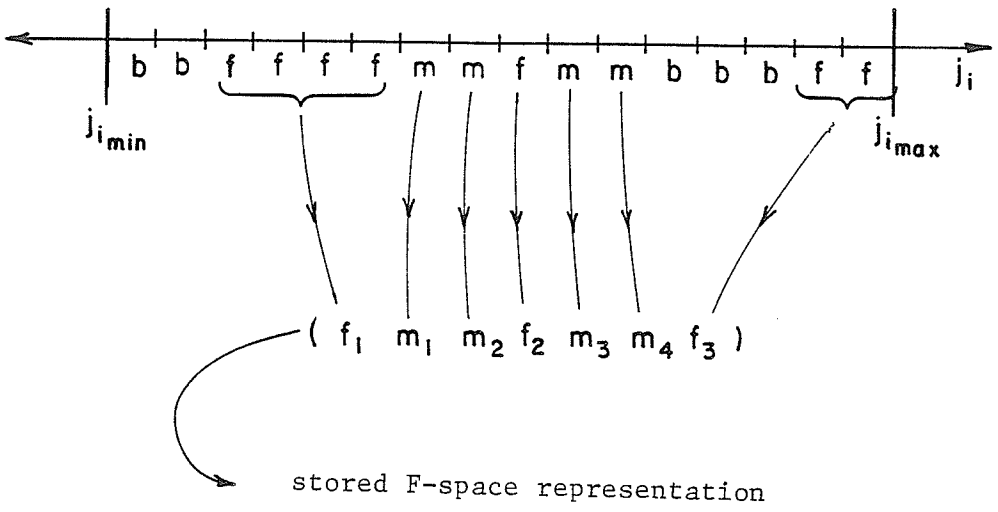
```
Given: - link is the name of a robot link
 - obstacles is a list of objects which
     can be considered as obstacles
 - link-swept-rep is the representation of
     the volume given by sweeping the remaining
     links over all their configurations
 - (interferep-obj a b), is a predicate function
     which returns t when two objects "a" and "b"
     interfere geometrically, otherwise nil
     is returned.


(F-space-decomp (link)
  (for all positions of the link
     (if (interferep-obj link obstacles)
           then move link to next position
     (else if (interferep-obj link-swept-rep
                        obstacles)
              then (F-space-decomp next-link))
        (else (add-cell-to-F-space-tree))))
```

For each degree of freedom of a robot, the algorithm classifies each interval in one of three ways. If the geometric representation of a link is colliding with any obstacles, then the interval is discarded without further investigation. If the geometric representation of all configurations of supported links is not colliding with any obstacles, a region of F-space has been found. Otherwise, the algorithm is called recursively with the name of the supported link. The efficiency of the final representation of F-space, in terms of number of F-space regions, is increased by combining F-space regions where appropriate. Although it is not explicitly stated in the abstract description of the algorithm, successive regions of F-space are combined at each link level. The list of F-space regions that would be found and stored for a link is shown in Figure 5.1. The resulting representation of F-space is a collection of F-space regions stored in a tree structure. The tree structure for a typical F-space representation is shown in Figure 5.2.

Performing an interference analysis between obstacles and a link positioned at the center of each discretization step does not guarantee that the entire C-space interval is free from collisions. One way to guarantee that an entire C-space interval is collision free is to determine the geometric shape of the volume swept by all configurations of the link in the discretization interval. The shape of this volume is generally not invariant with respect to its

b - blocked f - free m - mixed



f_i - F-space regions

m_i - lists of F-space regions that result from the recursive calling of the algorithm

Figure 5.1 Typical F-space regions for a robot link

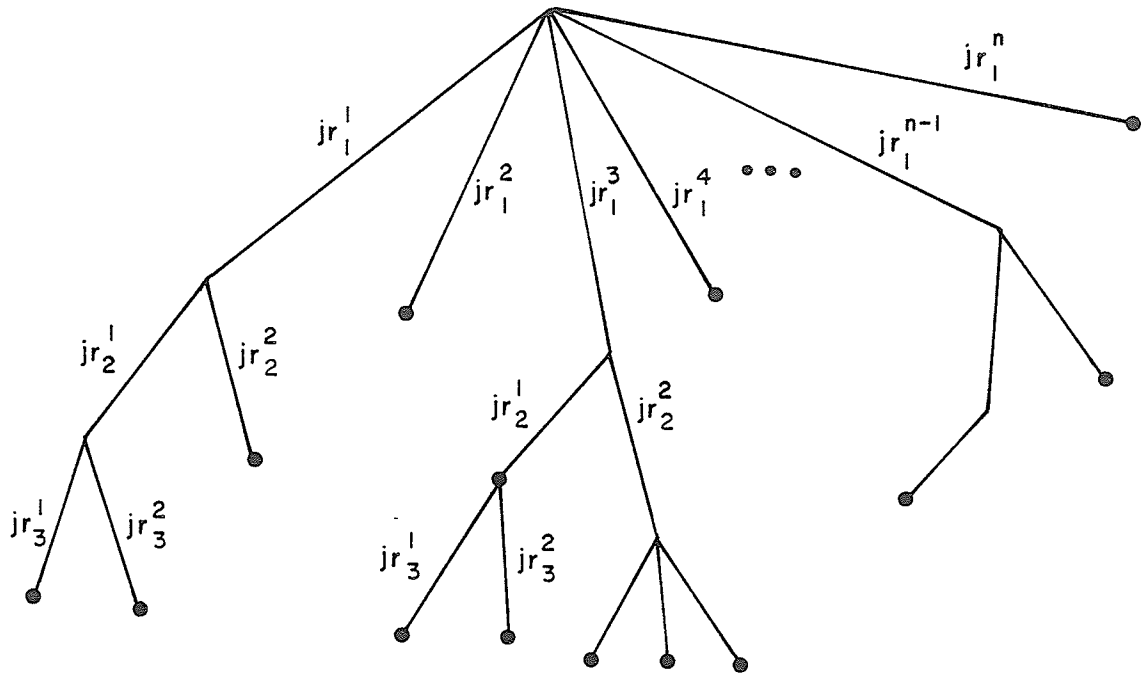


Figure 5.2 Tree organization of F-space regions

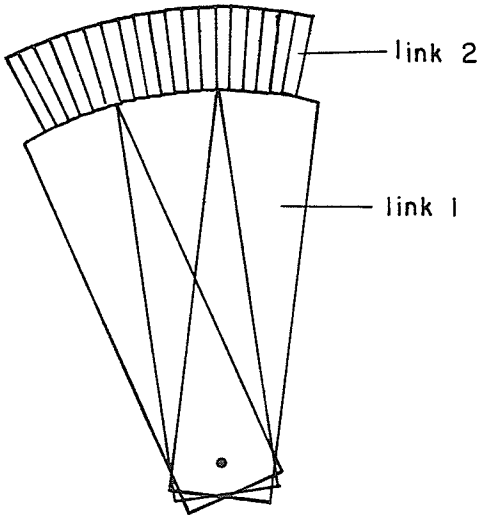


Figure 5.3 Different discretization intervals for robot links

position in C-space. Hence, in general, it is not possible to store this shape for collision detection and it would have to be regenerated for each interval. Another method is to determine a scale factor for enlarging a link in order to bound the swept volume of the link over any discretization interval. Alternatively, the obstacles could be enlarged by a sufficient amount. Both methods guarantee a report on complete intervals, but at the cost of extra computation and possibly a loss of F-space regions. Alternatively, one can allow the interference check to occur with the link positioned at the center of each interval and work with regions that represent either pure F-space or mixed C-space. Results indicate that this mixed space representation is quite useful. An example of using mixed space representation for the find-path problem is discussed later in this chapter.

5.1.1. Choosing a discretization step size

It is apparent, that the above algorithm requires a discretization step to be chosen for each link of the robot. Clearly, decreasing the discretization step size results in a more precise representation of F-space at the cost of more computation time. Choosing the discretization steps for the link parameters is done using knowledge of the geometry of the robot links. For example, rotational links with a large length to width ratio should have smaller steps than a rotational link with a lower length to width ratio (see Figure 5.3). Expression 5.1 is a simple relation which expresses this idea. In this expression n is the number of discretization levels, r is a length to width size factor of a link, dq is the total range for the joint parameter, and ceiling is a function which returns the next highest integer.

$$n = \text{ceiling}\ (r^{*}dq) \tag{5.1}$$

5.1.2. Finding adjacent regions of F-space

For purposes of finding safe paths it is useful to generate a connected graph representation of the F-space regions. To generate a connected graph it is necessary to find all adjacencies of F-space regions. Presently, the F-space regions are all hyperparallelpipeds in C-space. That is, they are defined by a set of intervals in each dimension of C-space. In general, a region of F-space

is defined by a set of link parameter ranges as shown below, where j_{r1} is a range of values for a link parameter.

$$(j_{r1}\, j_{r2} \cdots j_{rn})$$

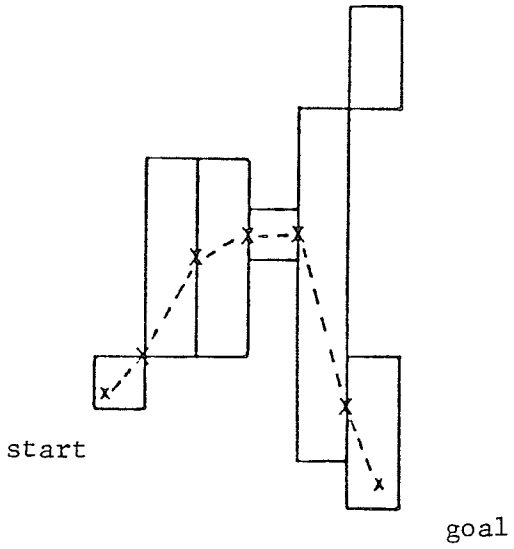
Two regions are adjacent if all link parameter ranges intersect at least for a single value. An efficient method for finding all adjacent regions is produced by using the fact that two regions can only be adjacent if they are members of adjacent branches in the tree organization of F-space regions. The algorithm developed uses the fact that two F-space regions can only be adjacent if they are adjacent at a higher level in the F-space tree structured representation. Region adjacencies are stored as a list of adjacent region names associated with each region. This type of representation is useful for breadth-first connected graph search techniques.

5.1.3. Finding paths from graph of F-space regions

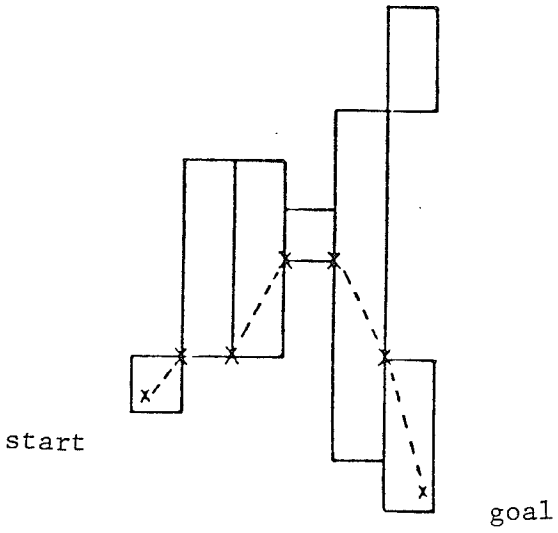
For the next step of this find-path solution, it is necessary to find which F-space regions contain the initial and final robot configurations and then find a connected sequence of regions between them. To find region membership of a robot configuration, one searches the tree organization of F-space regions until the correct one is found. To find a path between two regions there are several possible methods. Clearly, a breadth-first search of the connected graph of F-space regions will find the shortest path in terms of number of regions [21]. Due to the fact that regions are not of the same size, this sequence of regions may result in an unnecessarily long robot trajectory.

5.1.4. Generating robot path from a sequence of regions

To complete the find-path solution, one must generate a robot path from a sequence of F-space regions. Two methods of selecting a robot path were used, and both assume that the robot can approximately execute straight line motions in C-space. In the first method, one proposes a path consisting of straight lines between the centers of consecutive F-space region intersections. In the second method, one proposes a path that consists of a series of straight lines which connect the nearest point in consecutive F-space regions. Figures 5.4a and 5.4b illustrate the first and second methods on a similar sequence of regions. The first method generates the same path in either direction while the



a. path between consecutive region centers



b. path to nearest point of consecutive regions

Figure 5.4 Selecting robot paths from sequences of F-space regions

second method will generate a different path for both directions. The first method produces more conservative and possibly more expensive motion in terms of time and energy. Neither of the above algorithms produce an optimal path. As noted by Gouzenes [9,10], generating minimum time or minimum energy paths from a sequence of regions is an open problem.

For the case of regions that are not guaranteed to contain only free space, an unsafe robot trajectory may result. One way of circumventing this problem is to use the collision detector to test the safety of a proposed path. If a path is not safe, then one must either locally modify the part of the path which is causing the collision, or try a trajectory from the next shortest sequence of regions.

5.1.5. Execution of collision free paths with robots

In the preceding section it was assumed that a robot can execute straight line motion in its C-space. Typically, most robot control systems allow for a finite number of motion types between two configurations. Typically, a complex continuous trajectory for a robot is programmed using a series of straight line C-space motion commands between many intermediate points. Straight line Cartesian motion is generated with this type of strategy.

Assuming that each link of a robot is a solid body, the dynamics of robots can be represented by a set of n second order differential equations, where n is the number of joints or degrees of freedom. This set of differential equations is nonlinear and strongly coupled. Two methods for generating the set of differential equations are the Lagrange and the Newton-Euler techniques. If the dynamic model of the manipulator is known, this set of equations can be used to determine the torque inputs to the joint drive units for a particular set of positions, velocities, and accelerations for each joint.

Typically, robot controllers do not account for the complexity of their dynamic models. Hence even straight line motion in C-space will not be executed exactly the same throughout the working volume of an industrial robot. At low motion speeds the robot trajectory will be more accurate than at higher speeds. A better find-path system would include a complete model of the robot dynamics, control system and trajectory generation scheme. This thesis does not address these issues.

5.2. Robot Simulation system

A robot simulation system (PPLAN) was developed in LISP [25] with an LMI LAMBDA LISP Machine. While the cost of such a development system is high, this is offset by the increased productivity in terms of experimenting with new ideas. This simulator was required to verify and demonstrate the validity of the models and algorithms developed. One requirement of the simulator was to be able to animate the collision free paths of the two-dimensional experiments. This animation was required for purposes of verification of path safeness and demonstration purposes. This simulation required geometric models for purposes of display and collision detection. For some applications, the inverse kinematics must be defined for a robot and of course these are robot specific. Also, an interface to a robot is required and a serial communication interface to a PUMA 560 was developed for downloading collision free paths. Finally, an interface to the user was developed for purposes of rapid development, testing, and demonstration of models and algorithms. This interface included features such as: a graphic pointing device, multiple window display, and two-dimensional graphics for animation of robot paths. The proposed object representation, the associated collision detector, and its incorporation into the two find-path algorithms discussed above were implemented with the robot simulator.

5.2.1. Robot Modelling

Since a large class of robots can be represented with a tree structured database (see Figure 5.5), PPLAN was developed with such a data structure. A tree structured database consists of information nodes organized in tree structure. Each node is used to store information about one robot link. Presently, the information stored for each link includes: name of its support link, coordinate frame transformation relative to its support, name of its supported link(s), type of degree of freedom and a range of values, a geometric representation for collision detection, a geometric representation for display, a list of obstacles, a discretization interval size, and the name of an object which has a geometric representation of the volume swept out by all configurations of supported links. Other information which could be stored for each link might include; mass, moment of inertia, and other geometric representations. This data structure can represent any robot which does not contain closed mechanical loops. Included in this class of robots are serial link manipulators

typical of many industrial robots. Such robots are usually further classified as spherical, cylindrical or Cartesian robots, depending on whether link connections are rotational or translational.

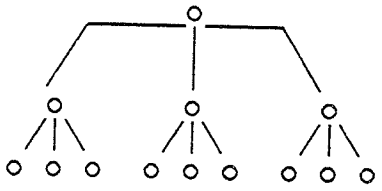


Figure 5.5 A tree structured database for robot representation

5.2.2. Kinematics

In addition to the C-space robot description, robot configurations can be described by the Cartesian location and orientation of a robot link, generally its end effecter. The Cartesian location of a robot is usually the preferred description as this is the world which we are most familiar with. Robot kinematics generally refers to the relationship between C-space and the Cartesian space descriptions of robots. Forward kinematics refers to the calculation of world coordinates of points on a robot from a set of joint values. Inverse kinematics refers to determining the joint values that correspond to a Cartesian position and orientation of the robot end effecter. Forward kinematics is an injective mapping and is simple to calculate. A good description for methods of calculating the forward kinematics can be found in Paul [17]. Generally, Cartesian robot locations do not uniquely describe the robot's configuration. In mathematical terminology the mapping from Cartesian space to C-space is surjective. Hence inverse kinematics solutions are usually robot specific and involve extra information or assumptions regarding the robot's configuration.

A bijective or one-to-one mapping for robot kinematics is often possible if configuration subspace flags are included with the Cartesian location. For example, consider a two-dimensional robot with two rotary joints shown in Figure 5.6. For any point in this robot's workspace we can obtain the same end effecter position with a right arm or left arm configuration. The right and

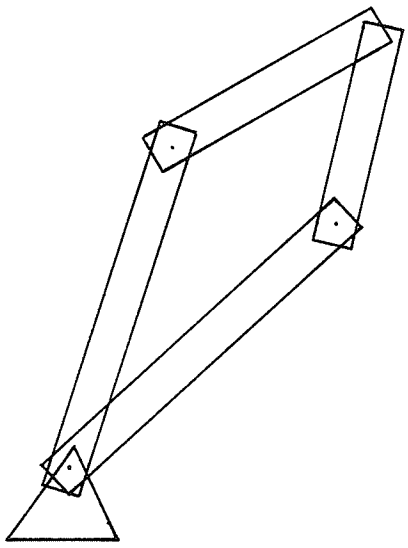


Figure 5.6 Right and left configurations
for an end effecter location

left arm configuration subspaces have a bijective mapping to the Cartesian description of the robot end effecter. For robots with six rotational joints such as a PUMA robot, there are eight configuration subspaces which must be considered in the inverse kinematic solution.

Finding the world coordinates of a point in an object's coordinate frame is determined by successively transforming the point by the coordinate frame transformations of all the supporting objects. Inverse kinematic solutions are generally robot specific and would normally have to be developed for a specific problem. For generate and test findpath experiments, the inverse kinematic solution for a PUMA 560 robot was implemented. Originally, this inverse kinematic solution was implemented in FORTRAN by J. Lauzon for the NRC, but was translated into LISP and incorporated into the PPLAN simulator by the author.

5.3. Experimental Results

After developing a suitable robot simulation environment, experiments could be performed on the proposed geometric representation and collision detector as well as its use within find-path algorithms. Experiments were performed for three two-dimensional path planning examples, as well as for three-dimensional examples with a PUMA 560 robot. Find-path experiments with the PUMA 560 were performed, both with a generate and test approach and a F-space representation approach.

5.3.1. Two-dimensional examples

Figure 5.7 shows three, two-dimensional examples and the paths that were generated by using the representation of F-space which was in turn generated with the proposed object representation and associated interference detector. The computation time for generation of the F-space representation was on the order of one minute for all three examples. This compares very favourably with times reported in [9,10] which are on the order of hours in essentially similar examples. It should be emphasized that work reported in [9,10] used facilities that are not optimized for LISP and this is probably responsible, to a large extent, for the long computing times reported. As was previously mentioned, it is possible to work with a description of mixed space representation of C-space. Notice that the path in the third example has a collision which can be detected with the collision detector. After detecting a

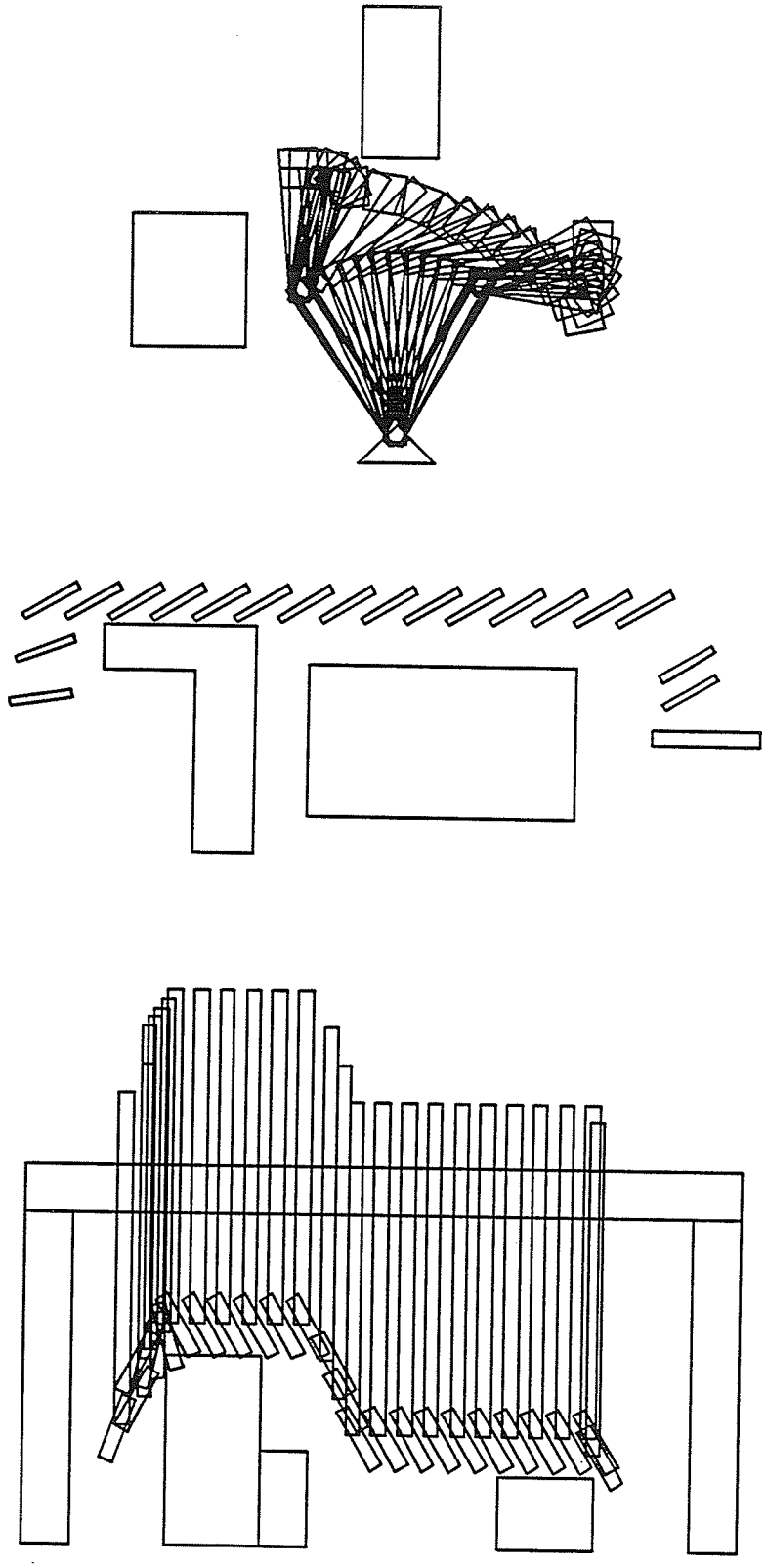


Figure 5.7 Examples of robot paths

collision it is possible to either modify the path or try to find a new path from the F-space representation.

Experiments were performed where all minimum length F-space sequences from a mixed C-space representation, in terms of number of F-space regions, were considered. The path selected was the shortest safe robot path of all the sequences. The final paths were safe but at the cost of approximately half a minute of processing time on the system. This processing time would increase for more precise representations of F-space.

5.3.2. Heuristic Path Planning

Experimental results for using the collision detector in a generate and test find-path approach are shown in Figure 5.8, where a PUMA 560 robot is executing a path that moves over a single obstacle with four joint-interpolated motions. Again, paths were checked with the proposed collision detector, with the object and the links of a PUMA 560 robot represented with a hierarchy of spheres. The rule used in generating the path shown was to raise the payload until it no longer interferes with the obstacle and attempt to move directly to the destination with joint-interpolated motion until another collision is detected. With the system, paths in three-dimensional Cartesian space require approximately one minute to generate.

5.3.3. Find-path from F-space representation

Results for generating and using an F-space description for three links of a three-dimensional robot, specifically a PUMA 560 robot, were also obtained. The added complexity of the third dimension is indicated by the much longer computation time required for determining F-space. For the first three links of a PUMA 560 robot with a single object in its environment, generation of the F-space description required approximately 10 minutes. For the same problem with four objects in the environment the computation time increased to approximately one hour. This increase in computing time is partially attributed to the fact that the "garbage collector" was necessary for the second test. The "garbage collector" is a LISP program which reclaims previously allocated memory, and slows down program execution time considerably. Also, an improvement may be possible if the four objects are grouped together and represented by a single sphere. This should reduce computation time appreciably because many of the checks will be made with only the single enclosing

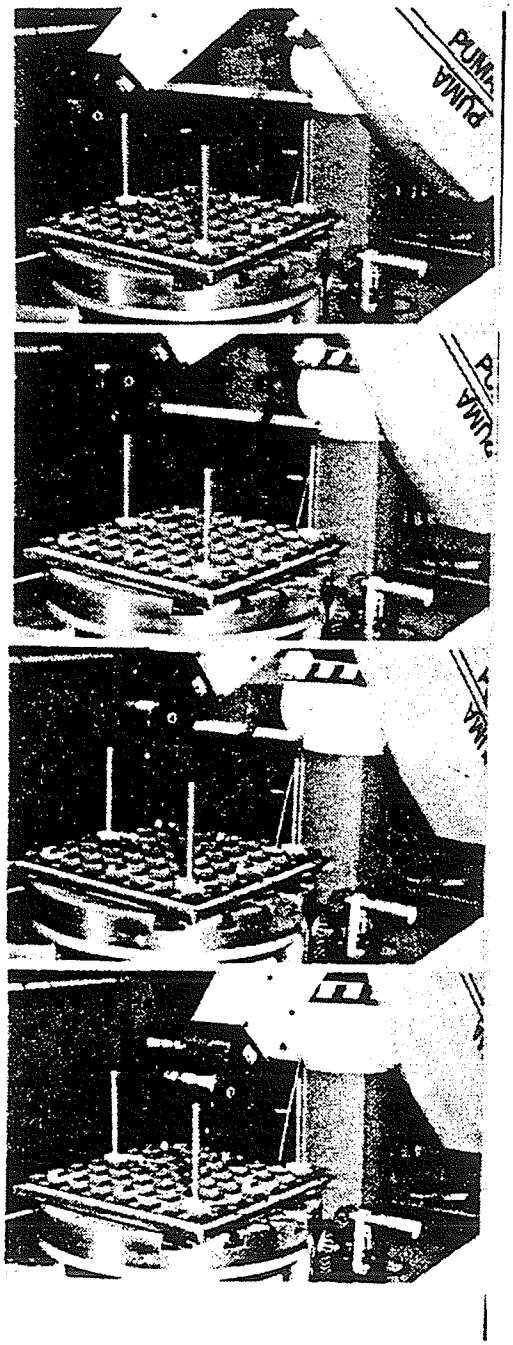


Figure 5.8 PUMA executing safe path determined by generate and test

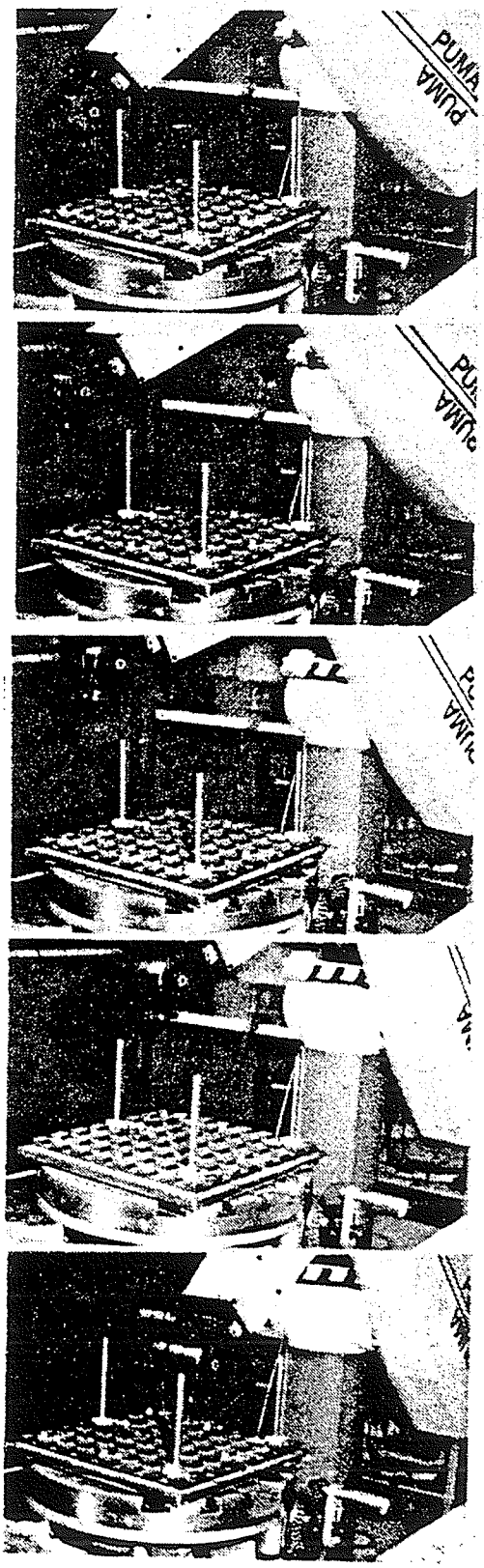


Figure 5.9 PUMA executing safe path determined from F-space representation

sphere rather than four. It should also be noted that the algorithms were not coded optimally and computation time could be improved by improving the software or translating the entire system into a language which is fast. Figure 5.9 shows photographs of a PUMA 560 robot executing a path generated by searching F-space. The initial and final configuration of the PUMA in Figures 5.8 and 5.9 were the same. In this example, it is worth noting that the path generated from F-space is slightly more conservative than that produced by the generate and test approach.

5.4. Extensions

In developing the robot simulation environment, experimenting with object representations for collision detection, and experimenting with find-path algorithms, many potential areas for improvement were identified. The following paragraphs identify some ideas and extensions which would possibly improve the system.

It may be useful to store each collision free path as it is found. This would result in a collection of safe robot configurations and safe paths between them, in other words, a second simpler representation of F-space. This partial representation of F-space could act as a "freeway network" upon which future path planning would be based. If the environment changes, the representation no longer guarantees safe paths and would have to be verified. Such a capability could be described as "learning" because the system would initially use a large amount of time to find collision free paths but would eventually use much less time to find new paths. This feature could be added to any type of find-path approach.

Methods of finding the sequence of F-space regions that contains an "optimal" robot trajectory should be investigated. Optimal could include criteria such as minimum time, minimum energy, maximum safety, and minimum computation time.

To improve the trajectory collision detector, it should be possible to dynamically determine different discretization intervals of optimal size for each robot link, for any trajectory. This will eliminate a large number of unnecessary collision checks. Also, it would be useful to include a trajectory model of the particular robot because the actual motion of a robot and the assumed path can be very different for some robot types.

A disadvantage of an approach that uses free space representation is that the representation is payload and world state dependent and that obtaining an F-space representation is the major computational burden of the approach. If the world state is static, then the approach is useful because the F-space representation can be determined offline. A useful concept which requires further investigation is an ability to dynamically vary the precision of a F-space representation in small regions of interest. With this type of approach one could generate a conservative F-space description using worst case models of possible payloads and very simple representations of objects for non critical regions of C-space. In critical regions of C-space, precise object representations could be utilized to generate a precise F-space representation. Thus, the computational burden of determining a F-space representation could be distributed over time because the robot can further investigate its environment only when necessary.

Finally, it would be advantageous to determine how to modify a F-space representation when small changes occur in the environment, such as adding or removing a single object from a robot workspace.

CHAPTER 6

Summary and Conclusions

A flexible and general robot simulator was developed for purposes of developing advanced offline programming tools such as collision detection and automatic path planning. This simulation environment is open to future change and experimentation by being based on an data structure which can model many types of robots.

An object geometric representation, with a hierarchy of spheres, was proposed and implemented for purposes of efficient collision detection between objects. The associated collision detector was found to have two useful properties. Firstly, that computation time for collision detection between two objects is dependent on the proximity of objects. Secondly, that the collision detector has the property of concentrating its object refinement to regions of the object most likely to be in collision.

From experiments performed with three two-dimensional path planning examples, and from experiments with a PUMA 560 robot, use of the proposed collision detector in gross motion planning problems was found to be very effective. This comparison is based on results presently being reported by other researchers.

LIST OF REFERENCES

[1] Jon L. Bentely and Thomas A. Ottman, *Algorithms for reporting and counting geometric intersections,* IEEE Trans. on Comp., vol. 28, pp. 643-647 Sep. 1979.

[2] Michael Brady, John M. Hollerbach, Timothy L. Johnson, Tomas Lozano-Perez, and Matthew T. Mason, *Robot Motion: Planning and Control* , MIT Press Cam. Mass. 1982.

[3] S. Bonner and K. Shin, *A comparative study of robot languages* ,Computer, Dec. 1982, pp 82-96.

[4] Rodney A. Brooks, *Solving the find-path problem by good representation of free space,* IEEE Trans. Syst. Man Cybern., vol. SMC-13, no. 3, pp. 190-197, March/April 1983.

[5] Rodney A. Brooks, *Planning Collision Free Motions for Pick-and-Place Operations,* The International J. of Robotics Research, vol. 2, no. 4, pp. 19-44, Jan. 1983.

[6] Rodney A. Brooks and Tomas Lozano-Perez, 1983 (Aug.8-12,Karlseruhe, West Germany). *A subdivision algorithm in configuration space for findpath with rotation,* Proc. 8th Int. Joint Conf. Artificial Intell. Los Altos, Calif.:Kaufman,pp.799-806.

[7] Francis Chin and Cao An Wang, *Optimal Algorithms for the Intersection and the Minimum Distance Problems Between Planar Polygons,* IEEE Trans. on Comp., vol. 32, no. 12, Dec. 1983.

[8] Philippe Coiffet, *Modelling and Control* , Hermes Publishing (France] 1981.

[9] L. Gouzenes, *Generation of Collision-free trajectories for mobile and manipulator robots,* IFAC Artificial Intelligence, Leningrad, USSR 1983.

[10] L. Gouzenes, *Strategies for Solving Collision-free Trajectories Problems for Mobile and Manipulator Robots,* The International Journal of Robotics Research, vol. 3, no. 4, Winter 1984.

[11] Tomas Lozano-Perez, *An Algorithm for Planning Collision-free paths Amongst Polyhedral Obstacles,* Communications of the ACM (22) vol. 10, pp. 560-570, Oct. 1979.

[12] Tomas Lozano-Perez, *Automatic planning of manipulator transfer Movements,* IEEE Trans. Systems, Man, Cybernetics SMC-11, 10 (1981),681-698.

[13] Tomas Lozano-perez, *Spatial Planning: A Confuguration Space Approach,* IEEE Trans. on Comp., vol. 32, no. 2, pp. 108-120, Feb. 1983.

[14] J.K. Myers and G.J. Agin, *A Supervisory Collision Avoidence System For Robot Controllers,* Robotics research and advanced applications. Presented at the winter annual meeting of the American SME., pp. 225-232, Nov. 14-19, 1982.

[14] Joseph O'Rourke, Chi-Bin Chien,et al, *A New Linear Algorithm for Intersecting Convex Polygons,* Comp. graphics and image Proc., vol. 19, pp. 384-391, Oct. 1981.

[15] W.T. Park, *State-space representation for coordination of multiple manipulators,* Proc. 14th IIR, pp. 397-405.

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

[17] A. Requicha, *Representations for Rigid Solids: Theory, Methods, and Systems,* Computing Surveys, vol. 12, pp. 437-464, Dec. 1980.

[18] Jacob T. Schwartz and Micha Sharir, *On the "piano movers" problem I. The special case of a rigid polygonal body moving amidst polygonal barriers.,* Commun. Pure Appl. Math, 36:345-398, Jan. 1983.

[19] Jacob T. Schwartz and Micha Sharir, *On the "piano movers" problem 2. General techniques for computing topological properties of real algebraic manifolds.,* vol. 4, no. 3, Adv. Appl. Math., Sept. 1983.

[20] Robert Sedgewick, *Algorithms,* Addision-Wesley Publishing Company, Inc., Reading, MA, 1984.

[21] Charles E. Thorpe, *Path Relaxation: Path Planning for a Mobile Robot,* pp. 318-321.

[22] S. Udupa, *Collision Detection and Avoidance in Computer Controlled Manipulators,* Proc. 5th Int. Joint Conf. Art. Intell., Cambridge, Massachusetts, 1977, pp. 737-748.

[23] Richard S. Wallace, *Three Findpath Problems,* pp. 326-329.

[24] Patrick Henry Winston and Bertold Klaus Paul Horn, *LISP,* Addision-Wesley Publishing Company, Inc., Reading, MA, 1981.

APPENDIX A

Overview

This appendix provides an overview of the PPLAN software written for robot path planning and collision avoidance. The intent of this appendix is to give the reader a basic understanding of each major part of PPLAN without a lot of detail. For more information on any particular aspect of PPLAN, the reader is directed to Appendix B the PPLAN manual. This appendix is organized as follows. First a description of the world modelling method and utilities is given. Second, a description of the interference detection capabilities is presented. Next, the path planning algorithms are described and, finally, a description of the user interface is included. Before proceeding, a brief explanation of why LISP was chosen as the development language is given.

A.1 Why LISP ?

PPLAN is written in ZetaLISP on an LMI LAMBDA LISP Machine. LISP was chosen as the development language because of its powerful and productive programming environment and its flexibility in allowing program changes. If increased execution speed becomes necessary to incorporate this system in an application, then another language may be superior. The emphasis of this work was to try new ideas quickly and test their feasibility, and for this reason LISP is considered a superior language by many. The remainder of this appendix contains a brief description of the details and capabilities of PPLAN.

A.2 Modelling

A.2.1 Defining a World

This section describes how a particular robot and its environment can be modelled with the PPLAN program package. A robot and objects in its environment is modelled by a collection of objects or a list of objects called "world". The elements of the list are objects used to represent the robot links, swept volumes of robot links and obstacles in the robot's environment, for example, the following.

world => (base shoulder arm ... post table block ...)

A.2.2 Defining Objects

Each object has a data structure associated with it. The data structure is implemented with property lists. A property list in LISP is a collection of key-words and their values. The properties of objects presently used in PPLAN are as follows.

Object's property list

1 rep - A representation for purposes of graphic displays.

2 loc - location transformation relative to coordinate frame of supporting object.

3 dof - type of degree of freedom and range

4 bsphere - hierarchical representation of object with bounding spheres for interference detection between two objects.

5 fanout - list of all objects immediately supported by self.

6 checkl - list of all objects which should be checked for possible geometric interference.

7 swept-obj - name of object which has a representation which bounds the volume swept out by all supported objects.

8 sizef - scaler value giving rough indication of object size

A.2.3 Generating geometric representation

A function called find-bsphere is used to generate a hierarchy of bounding spheres for an object's geometric representation. This function returns a hierarchy of spheres representation for a box with a trapezoidal cross-section. A version of this function exists for both two and three dimensional objects. For purposes of explanation, consider a simple rectangle (Figure A.1) which we will call toy box. We describe a rectangle by its length, width and coordinates of its center. For the rectangle illustrated we would call the function as indicated and assign the returned value to the property bsphere of the name of the box.

```
(setq xc 4.0)
(setq yc 0.5)
(setq dx 8.0)
(setq dy 1.0)
(putprop 'toy-box (find-bsphere '(xc yc dx dy)) 'bsphere)
```

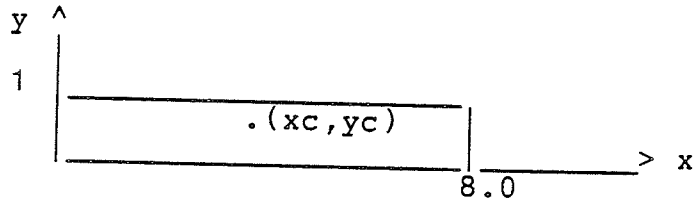


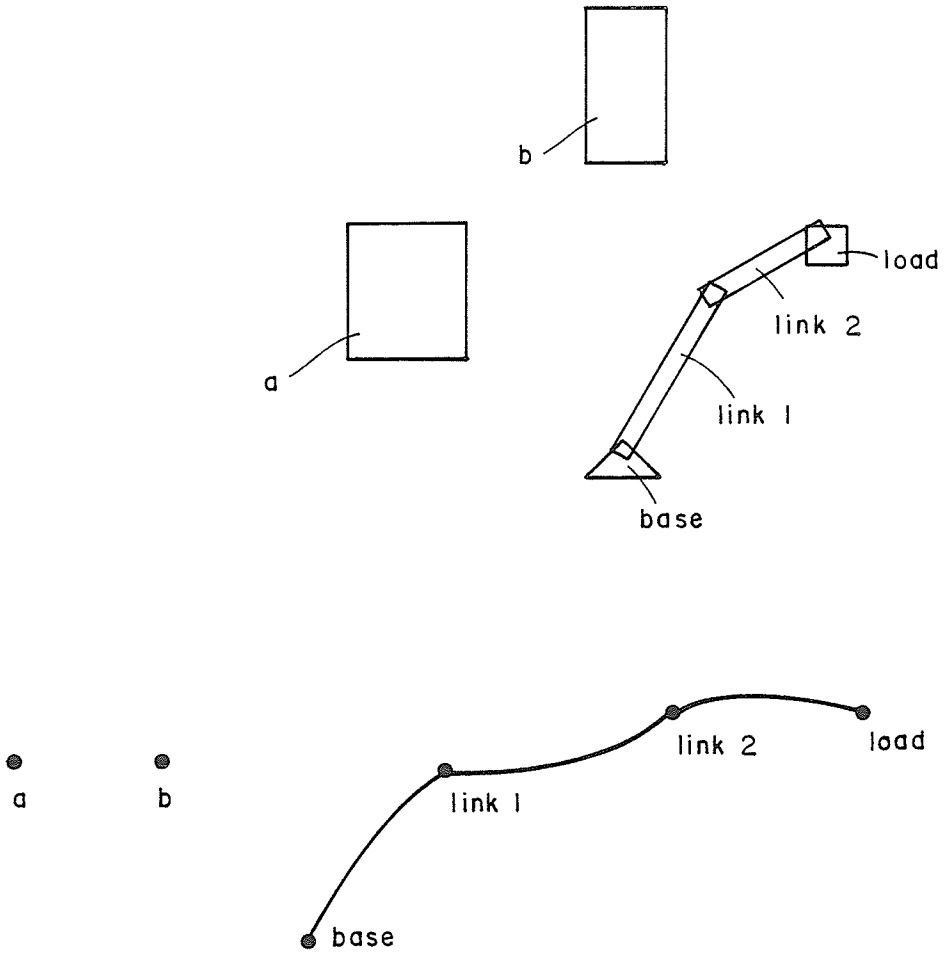
Figure A.1 generating bsphere representations

This function is also provided for generating bsphere representations for three dimensional objects. In addition it is possible to specify shrinking or enlarging factors which allow the algorithm to generate bsphere representations for three dimensional boxes with trapezoidal cross sections.

A.2.4 Defining Robots

The object data structure described above can be used to model any tree structured, two or three dimensional robot whose degrees of freedom can be rotational or translational. Hence robots can be mobile, fixed or have multiple arms. Open linked robots are modelled with objects by using the support and

fanout properties. For example a simple two dimensional robot with a base and two rotary joints is represented as follows. The support of the base is w which is the symbol indicating that an object is directly supported by the world. The value of support for link1 is base and the value of fanout for link1 is (link2). The value of support for link2 is link1 and the value of fanout for link2 is nil. The form for the fanout property is a list. This allows for the possibility of multiple arm robots. As an example consider the robot illustrated in Figure A.2 and the associated data structures.



world - (base link1 link2 load a b)

Figure A.2 A robot and its data structure

Robot link and obstacle properties

```
A
(BSPHERE (((2.0 4.0) 4.47213595) (((2.0 6.0) 2.828427127))
                                 (((2.0 2.0) 2.828427127)))
 SUPPORT W
 LOC (0.0 16.0 0.0)
 REP ((0.0 0.0) (4.0 0.0) (4.0 8.0) (0.0 8.0) (0.0 0.0))
 CHECKL NIL)

B
(BSPHERE (((3.0 3.5) 4.609772228))
 SUPPORT W
 LOC (-12.0 6.0 0.0)
 REP ((0.0 0.0) (6.0 0.0) (6.0 7.0) (0.0 7.0) (0.0 0.0))
 CHECKL NIL)

BASE
(BSPHERE (((2.0 0.6666666665) 2.108185105))
 FANOUT (LINK1)
 SUPPORT W
 LOC (0.0 0.0 0.0)
 REP ((0.0 0.0) (4.0 0.0) (2.0 2.0) (0.0 0.0))
 CHECKL NIL)

LINK1
(BSPHERE
 (((0.0 4.5) 5.02493781)
  (((0.0 7.0) 2.549509754)
   (((0.0 8.25) 1.346291201) (((0.0 8.875) 0.8003905294))
                             (((0.0 7.625) 0.8003905294)))
   (((0.0 5.75) 1.346291201) (((0.0 6.375) 0.8003905294))
                             (((0.0 5.125) 0.8003905294))))
  (((0.0 2.0) 2.549509754)
   (((0.0 3.25) 1.346291201) (((0.0 3.875) 0.8003905294))
                             (((0.0 2.625) 0.8003905294)))
   (((0.0 0.75) 1.346291201) (((0.0 1.375) 0.8003905294))
                             (((0.0 0.125) 0.8003905294)))))
 SWEPT-OBJ LINK1-SWEPT
 FANOUT  (LINK2)
 SUPPORT BASE
 LOC (2.0 1.5 57.14285707)
 DOF (TH (-60.0 60.0))
 sizef 10.0
 REP ((-0.5 -0.5) (0.5 -0.5) (0.5 9.5) (-0.5 9.5) (-0.5 -0.5))
 CHECKL (A B))

LINK1-SWEPT
(BSPHERE (((0.0 9.0) 10.0))
 SUPPORT LINK1
 LOC (0.0 0.0 0.0)
 REP ()
 CHECKL (A B))
```

```
LINK2
(BSPHERE
 (((0.0 3.0) 3.535533905)
  (((0.0 4.75) 1.820027472) (((0.0 5.625) 1.007782218))
                            (((0.0 3.875) 1.007782218)))
  (((0.0 1.25) 1.820027472) (((0.0 2.125) 1.007782218))
                            (((0.0 0.375) 1.007782218))))
 SWEPT-OBJ LINK2-SWEPT
 FANOUT (LOAD)
 SUPPORT LINK1
 LOC (0.0 9.0 76.0)
 DOF (TH (-80.0 80.0))
 sizef 7.0
 REP ((-0.5 -0.5) (0.5 -0.5) (0.5 6.5) (-0.5 6.5) (-0.5 -0.5))
 CHECKL (A B))

LINK2-SWEPT
(BSPHERE (((0.0 6.0) 3.0))
 FANOUT ()
 SUPPORT LINK2
 LOC (0.0 0.0 0.0)
 REP ()
 CHECKL (A B))

LOAD
(BSPHERE
      (((0.5 0.5) 1.414213563))
 SWEPT-REP NIL
 SUPPORT LINK2
 LOC (0.0 6.0 110.76923066)
 DOF (TH (-120.0 120.0))
 sizef 3.0
 REP ((-0.5 -0.5) (1.5 -0.5) (1.5 1.5) (-0.5 1.5) (-0.5 -0.5))
 CHECKL (A B BASE))
NIL
```

A.2.4.1 Kinematics

Forward kinematics for a robot is handled naturally by the method of representing each link. To obtain the world coordinates of a point on the end effecter of a robot chain, we simply find the coordinates of the point relative to each supporting link of the robot until the world coordinates are found. Normally each robot link has a single degree of freedom associated with it. If a single object requires more than one degree of freedom then virtual links, each with one degree of freedom, can be defined. For purposes of kinematic computation speed, it may be of interest to incorporate a specific transformation function for the forward kinematics of a robot, rather than working through a chain of links each time the world coordinates of an object on the end effecter of the robot is required. A forward kinematic solution for the

PUMA 560 robot which generates the position and orientation of the end effecter relative to the robot coordinate frame is implemented into the PPLAN environment.

For specific applications it may be important to incorporate the inverse kinematic solution for a robot. For applications with a PUMA-560 robot, its inverse kinematic solution was implemented in LISP and included into the PPLAN programming environment. It should be noted that the solutions for the forward and inverse kinematic solutions for the PUMA 560 were generated by J. Lauzon for the National Research Council. The kinematic solutions were originally written in Fortran. Translation of the kinematic solution into LISP was done by the author.

A.3 Geometric Interference detection

Functions are available which perform interference detection between any two objects with a bsphere representation. PPLAN has a number of higher level geometric interference detection routines which use the object to object collision detector. An interference detector for an object and a list of objects, and an interference detector for a robot and objects in checkl properties of robot links, are some of the high level interference detection routines available.

Trajectories of a robot are checked by a series of static interference checks along a trajectory. Two types of robot trajectories can be checked. Joint-interpolated trajectories are approximated by straight lines in the joint space. Trajectories with straight line motion of a robot end effecter can also be checked for collisions, but an inverse kinematic solution, which are robot specific, is required.

A.4 Path planning

A.4.1 Generating a description of free space

Free space is described by the set of all robot configurations which are not colliding with other objects. A representation of F-space for any robot described by PPLAN can be generated. A F-space representation is a collection of free space regions defined by ranges of the robot's degrees of freedom. These regions are represented by atoms which are automatically generated as the free space regions are found. Each F-space region has three properties. A

property list for a free space region is as follows.

1 def - list of joint value ranges which describe a hyperparallelpiped in the configuration space of the robot. def => ((j_{1min} j_{1max}) (j_{2min} j_{2max}) ... (j_{nmin} j_{nmax}))

2 neighbor - list of F-space regions adjacent to self neighbor (f_{34} f_{55} f_{98} .. f_{213})

3 visitp - boolean value used for find path algorithms

The cells are stored in a tree structure whose name is "fspace". To generate a description of free space the user must evaluate the function fspace-decomp and assign the returned value to the variable "fspace".

(setq fpsace (fspace-decomp root-link-of-robot))

A.4.2 Generating robot trajectories from free space

Often path planning is based on heuristics. Hence to build a path planner for a specific robot one must determine the heuristics or algorithms applicable and build the path planner using the programs given here as a basic framework.

Another method of path planning is to find a representation of the robot's free space. This representation should be built so that a path can be found if it exists and report that a path does not exist if one is not found.

A.5 Mathematical Library

A library of useful mathematical functions such as transforming three dimensional points, converting degrees to radians, distances between n dimensional points, and other common procedures was developed.

A.6 User Interface

A.6.1 load/store/create worlds

A world can be created interactively or through any text editor. Normally, a skeleton data structure for a robot and its environment is created as a data file with a text editor. This skeleton robot environment is then loaded into the PPLAN environment and the geometric properties are defined using some of the tools mentioned in the modelling section. Finally, the robot and its environment is saved on a new data file. Much more complete and powerful modelling tools are conceivable but this begins to move into the area of

CAD which is rapidly becoming a well established technology. Rather than developing such capabilities into the PPLAN system it may be simpler to establish links to an appropriate CAD environment.

A.6.2 Window Interface

After all PPLAN files are loaded into the PPLAN environment of the LISP machine, the user is immediately placed in a window interface to the PPLAN system. From this window all functions of the PPLAN system are available. In addition, communication and interaction with a PUMA-560 robot is available for PUMA 560 applications.

The PPLAN window interface is divided into four panes labelled command pane, status pane, graphics pane and interaction pane. The command pane contains a list of actions which the user can choose with a graphic input device, i.e. a "mouse". The status pane is unused at this time, but it is available to display status information about the robot and the program. The graphics pane is to display animation of two dimensional path planning examples. The interaction pane is used to display messages and query the user for information when necessary. An example of a typical PPLAN window is shown in Figure B.1. A brief summary of mouse selectable commands in the PPLAN system is described in the PPLAN Manual in Appendix B.

APPENDIX B

PPLAN MANUAL

This manual includes a brief description of all major functions of the PPLAN robot path planning software, and is organized as follows. The first section explains functions used for modelling objects, robots and robot motion. The second section relates to functions used for collision avoidance. The third section deals with functions for path planning. The fourth section describes functions used to interface PPLAN to the user and PPLAN to a PUMA 560 robot. Finally mathematical functions developed for PPLAN are described.

B.1 MODELLING

The source code for these functions is contained in

SYS$USER1:[SAWATZKY.LMI.PPUMA]MODEL.L;1
LAMA:SAWATZKY.PPUMA;MODEL.L;1

(input-obj) : Interactively build a world description with user. Presently, the following object properties are the only ones used by PPLAN

Object's property list

1 rep - For two dimensional objects this is used to represent the vertices of a polygon which describes the object for purposes of graphic displays. Unused for three dimensional objects, however, it is available for a representation suitable for three dimensional graphics.

rep => ((x1 y1) (x2 y2).... (xn yn)) for two dimensional

objects

2 loc - list position of an object relative to its support's coordinate frame.

(dx dy th) for two dimensional objects
(dx dy dz ax ay az) for three dimensional objects

3 dof - degree of freedom indicated by
(type (min max))
where type is one of {dx dy dz ax ay az th}
and min and max are real numbers.

4 bsphere - hierarchical representation of object with bounding spheres
(bs s1 s2)
bs is a bounding sphere defined by ((x y z) r), where x,y and z are real numbers which define the center of a sphere and r is a real number which defines the sphere radius.
s1 and s2 are the same form as bsphere and represent two sub-objects whose union represent the original object.

5 fanout - list of all objects immediately supported by self.

6 checkl - list of all objects which should be checked for possible geometric interference.

7 swept-obj - name of object which has a representation which bounds the volume swept out by all supported objects.

8 sizef - real number giving a rough indication of object size

(obj-fanout ol) : Determines all objects supported by each object in the object list ol. The fanout list are assigned to the fanout property. This function requires that only object supports are defined. Alternatively the fanout properties may be manually defined.

(find-bsphere x) : Returns a binary tree representation of the rectangular solid defined by x. x is a list that contains information regarding position and size of the object.

x => (xc yc zc lx ly lz)

xc, yc, and zc are real numbers defining the center of a box relative to its origin.

lx, ly, and lz are real numbers defining the lengths of the box in the x, y, and z dimensions.

It is also possible to define shrinking factors for any box dimension. These are global variables called *lxy, *lyz, *lxz, *lyx, *lzy, and *lzx. They define the rate of shrinking for an object dimension in a particular direction. By default they are all equal to 1.0, i.e. no shrinking. This allows boxes with trapezoidal cross sections us to be represented.

world : A global variable which is a list of all objects in the PPLAN environment, including the manipulator and obstacles.

B.1.1 Robot motion functions

The following functions are used to move a robot model and to determine where a robot model is. The source for these functions is contained in

LAMA:SAWATZKY;PPUMA;MOTION.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]MOTION.L

*doftype : A global variable which is an ordered list of the types of degrees of freedom allowed for objects. The order is the same as for location lists for objects.

dl : A global variable which is a list of objects in the world which have degrees of freedom. The list alternates between objects and their type of degree of freedom.

locations : A global variable which is a list of robot location names

*r : A global variable which is a list of link size factors used to give a rough indication of the relative size of an object.

(dof-list ol) : Returns a list of all objects from ol that have a degree of freedom associated with them and the type of degree of freedom.

ol => (o1 o2 o3 ... on) ;a list of objects

returned value => (oi type oi type ...)
where type is x, y, z, ax ,ay or az

(drive-nth-joint dl n pos) : Set the nth degree of freedom in dl to pos, where dl is a list of degrees of freedom, n is an integer, and pos is real value.

(drive-joint obj pos) : Set the degree of freedom of an object to a value, where obj is the object and pos is the desired value.

(joint-limitp x range) : Returns t if x is not in range. x is a number and range is a list of two numbers (min max).

(move x) : Sets the location of joints or degrees of freedom of the robot to the transformation defined by x.

(here x) : Defines x as the present location of the robot.

(where) : Returns a list of all joint values of the robot and the real world coordinates of the last joint.

(where-joint dl) : Returns a list of all joint values defined by the degree of freedom list dl.

This file contains functions that determine real world coordinates of points on objects.The following functions can be found in.

LAMA:SAWATZKY.PPUMA;KINEM.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]KINEM.L

(find-world-coor-pt x obj) : Returns the world coordinates of a point x which is defined relative to object obj.

x => (x y) or (x y z)
obj => object name

(find-world-coor x rep) : Returns the world coordinates of a representation of x, where x is an object name and rep is the representation to be transformed.

x => object name
rep => ((x y z) ...)

(transform-pt x transf) : Returns point x transformed by transf

x => (x y) or (x y z)
transf => (dx dy th) or (dx dy dz ax ay az)

(transform rep loc) : Returns the list of vectors transformed by loc. Rep is a list of points and loc is a three dimensional transformation.

rep => ((x y z) ...)
loc => (dx dy dz ax ay az)

This collection of functions perform the forward kinematic solution for a PUMA 560 robot. These programs are a translation of programs written by J. Lauzon a former employee of the National Research Council of Canada. The translation into LISP was done by G. Sawatzky. The following functions are found in.

LAMA:SAWATZKY.PPUMA;dirk.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]dirk.L;1

(dirk-6r j) : Returns the xyzoat representation of the joint space vector j. j is a list of joint values from a 6-r robot (j1 j2 j3 j4 j5 j6) given in degrees.

*a1 *a2 *a3 *a4 *a5 *a6

*d1 *d2 *d3 *d4 *d5 *d6

are global variables which define PUMA 560 link parameters

(t6-to-xyzoat t6) : Return location in xyzoat format. t6 is a location t6 matrix and is expected to be in a list of list of columns.

This collection of functions perform the inverse kinematic solution for a PUMA 560 robot. These programs are a translation of programs written by J. Lauzon for the National Research Council of Canada. The translation into LISP was done by G. Sawatzky. The following functions are found in the following files.

LAMA:SAWATZKY.PPUMA;INVK.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]INVK.L;1

(invk-6r x ci) : Returns the joint angles of a three dimensional 6r (6 rotations,open link) robot. x is a list that contains the position and orientation of the end effecter (x y z o a t). ci is a list that contains the configuration indicators of the robot (arm elbow wrist). Link parameters for the robot are stored in the following global variables.

*a1 *a2 *a3 *a4 *a5 *a6

*d1 *d2 *d3 *d4 *d5 *d6

(compute-t6 x) : Returns the t6 location transformation of the robot flange. x is the list of three dimensional location and orientation (x y z o a t) obtained from the PUMA 560. t6 is a 4 by 4 matrix and it is returned as a list of columns ((c1) (c2) (c3) (c4)).

(find-config j) : Returns a list of configuration flags that correspond to the PUMA 560. j is a list of the joint values in degrees (j1 j2 j3 j4 j5 j6). configuration flags are returned as (arm elbow wrist).

B.3 Interference Detection functions

This file contains all of the geometric interference functions used in the PPLAN package

LAMA:SAWATZKY.PPUMA;INTERF.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]INTERF.L;1

(interferep ol) : Checks each object in the object list ol, against each other object for interference.

(interferep-aux x ol) : Checks object x against all objects in the object list ol for interference. Returns a list of all pairs of objects which interfere geometrically. Otherwise, nil is returned.

(interferep-aux1 x ol) : Checks object x against all objects in the object list ol for interference. Returns a list of the first pairs of objects which interfere geometrically. Otherwise, nil is returned.

(interferep-obj x y) : Checks object x against object y for interference.

(interferep-bsphere x y xrep yrep) : Return t if the bounding spheres of x and y do not intersect.

(interferep-sphere c1 r1 c2 r2) : Returns t if spheres defined by centers c1,c2 and radii r1 ,r2 intersect.

(interferep-robot link) : Returns a list of all pairs of objects which interfere geometrically. Objects checked are all the objects in the checkl property

of links supported by link.

(interferep-robot1 link) : Returns a list of the first pair of objects which interfere geometrically. Objects checked are all the objects in the checkl property of links supported by link.

(safep-move x) : Returns t if robot's trajectory from its present configuration to its goal location is collision free. Otherwise, the last safe robot location is returned. x is the goal location. Trajectory is assumed to be a straight-line in joint space. For most instances this should be a suitable approximation to joint-interpolated motion.

(safep-moves x) : Returns t if robot's trajectory from its present configuration to its goal location is collision free. Otherwise, the last safe robot location is returned. x is the goal location in the xyzoat form. Trajectory is assumed to be a straight-line in cartesian space.

(safep-path p) : Returns t if the path p is safe. Otherwise, the joint vector at which the first collision is detected is returned.

p => path to be executed in the form of a list of move commands.

(interferep-line-polygon v1 v2 yrep) : Returns t if a line segment intersects with a two dimensional polygon (unused in PPLAN).

(interferep-line-line v1 v2 v3 v4) : Return t if two line segments defined by endpoints v1,v2 and v3 ,v4 intersect (unused in PPLAN).

(intersectp-line-pt x1 x2 x3) : Returns t if a point intersects a line segment where x1 and x2 are the endpoints of the line segment (unused in PPLAN).

B.3 PATH PLANNING

This section relates to functions used for path planning. The first part deal with building and using a description of Free Space of a robot and its environment and can be found in the following file.

LAMA:SAWATZKY.PPUMA;FDECOMP.LISP;1
SYS$USER1:[SAWATZKY.LMI.PPUMA]FDECOMP.L;1

fspace : A global variable used for the tree of the free space cells for a robot

(fspace-decomp x) : Returns a tree-structured list which represents the free space of an open link robot whose root link is x.

(find-dsteps x) : Determines the discretization interval for each degree of freedom of a robot that has body x as its root.

(gen-cell cell y) : Creates cell and defines its size property.

(find-adj x) : For each cell in x its adjacent cells are found. x is a tree structured representation of a space. If a cell g1 is adjacent to gi then gi is added to g1's neighbor list and g1 is added to gi's neighbor list.

(seg-test x1 x2 y1 y2) : Returns a code indicating the relative position of two line segments defined by x1 x2 and y1 y2.

(add-neighbor x y) : Add x to y and y to x neighbor lists.

This file contains some path finding functions which are based on a representation of Fspace stored in fspace. These functions can be found in the following files.

LAMA:SAWATZKY.PPUMA;FINDP.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]FINDP.L;1

(find-path x) : Returns a path from the present robot configuration to the robot configuration defined by x, where x is a joint vector. The path returned is in the form of a list of move commands

returned path => (move j1 move j2 ... move jn) or nil if none exists

(breadth start f) : Returns a path of Fspace regions from start to f using a breath-first search.

start => name of the Fspace region containing the initial robot configuration

f => name of a Fspace region containing the final robot configuration

(cell-init x) : Initializes visitp properties of all Fspace regions in x to nil

x => tree representation of Fspace.

(path-translate js jf p) : Returns a series of joint interpolated moves which would move the robot through p, a path of Fspace regions.

js => initial robot configuration

jf => final robot configuration

(find-cell j x) : Returns the Fspace region which contains the robot configuration defined by j.

j => vector of joint values for a robot configuration

x => tree representation of Fspace

This file contains some path finding functions which are based on a representation of mixed Cspace called fspace

LAMA:SAWATZKY.PPUMA;FINDPMS.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]FINDPMS.L;1

(find-path x) : Returns a path from the present robot configuration to the robot configuration defined by x, where x is a joint vector. The path returned is in the form of a list of move commands. This version of find-path selects the shortest collision free path from all paths of minimum path length in terms of number of Fspace regions.

returned path => (move j1 move j2 ... move jn) or nil if none exists

(path-length s p) : Finds the Euclidian distance of a series of joint-interpolated motions expressed as a path.

s => initial robot configuration expressed as a list of joint values

p => path to be executed in the form of a list of move commands.

(breadth start f) : Returns a list of paths of Fspace regions from start to f using a breath-first search.

start => name of the Fspace region containing the initial robot configuration

f => name of a Fspace region containing the final robot configuration

returned value => list of lists of paths expressed as a list of Fspace regions.

This file contains path finding functions which are based on simple hueristics derived from knowledge of the world It also contains functions relating to finding paths from a Fspace representation and can be found in the following.

LAMA:SAWATZKY.PPUMA;FINDPh.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]FINDPh.L;1

(findpath x) : Returns a safe path for the robot that moves the robot to the configuration defined by x. x is the initial robot configuration defined as a list of joint values. nil is returned if a path does not exist or could not be found.

(change-z x dz) : Return a legal robot configuration which has a higher z coordinate for the load position. Load orientation remains the same. x is a robot configuration in joint vector form and dz is a z offset.

(change-r x fr) : Return a legal robot configuration which has a load position which is closer or farther from the base. Load orientation remains the same. x is a robot configuration in joint vector form and fr is a radius scale factor.

B.4 User and PUMA interface functions

The following functions are used for loading and saving world information. This file contains functions which carry out basic input output functions for PPLAN such as reading in a world or robot locations The functions are contained in.

LAMA:SAWATZKY.PPUMA;IO.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]IO.L;1

(read-world filen) : Reads objects and their property lists from an input file filen.

(read-locations filen) : Reads robot locations from an input file filen.

(print-locations filen) : Prints robot locations to a file called filen.

(print-plist ol filen) : Prints the objects and their property lists to the file filen. ol is a list of objects.

(print-path (x) : Print the number of intermediate points and the joint values of the points to a file called filen.

(print-cells x) : For each cell in x a cell is generated using gensym. x is a tree structured representation of a space.

(read-cells) : Reads the free space decomposition tree and the cells into the lisp environment.

The following functions are found in the following.

LAMA:SAWATZKY.PPUMA;PPLAN.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]PPLAN.L;1

(load-pplan) : Loads files which contain pplan functions.

This file contains user interface functions, most of which are not used very often. The following functions are found in the following.

LAMA:SAWATZKY.PPUMA;USERINT.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]USERINT.L;1

(help) : Prints information on some commands available to the user.

The the toplevel user interface for the PPLAN system is defined by the functions found in the following files.

LAMA:SAWATZKY.PPUMA;WINDOW.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]WINDOW.L;1

Functions to control serial communication with the PUMA 560 robot are found in the following files.

LAMA:SAWATZKY.PPUMA;PUMA_COM.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]PUMACOM.L;1

(puma-init) : Initialize serial port and assign to puma-stream

(puma-exit) : Close serial port and assign nil to puma-stream

(to-puma) : A virtual terminal emulation program for the puma 560. To exit program enter <quit>.

(puma-com command) : Sends command to puma-stream and returns the immediate response of the puma 560 in a character string. Command is given as a character string. Caution there may be future responses from the PUMA.

(where-puma) : Returns the xyzoat and joint angles as a list from the puma 560 An attempt is made to check for a transmission error and the user is notified.

(puma-point x name) : Defines the configuration defined by x to name in the puma.

(here-puma name) : Name is a location to be defined in the PUMA memory. After appending the string name to "here " the function returns either the xyzoat or joint angles as a list.

(puma-move name) : Name is a string of a location name already defined in the PUMA memory. name is appended to the end of a "do move" command which is then sent to the puma. This function returns t if the attempted move is received by the PUMA, otherwise, returns nil and prints a message to terminal display.

(puma-moves name) : Name is a string of a location name already defined the PUMA memory. name is appended to the end of a "do moves" command which is then sent to the puma. This function returns t if the attempted move is received by the PUMA, otherwise, returns nil and prints a message to terminal display.

(puma-ex name) : Name is a string of a program name already defined in the PUMA memory. name is appended to the end of a "ex " command which is then sent to the puma. This function returns t if the attempted program is received and executed by the PUMA, otherwise, returns nil and prints a message to terminal display.

B.5 Mathematical Functions

The following functions are used wherever needed and are found in the following files.

LAMA:SAWATZKY.PPUMA;MATH.LISP
SYS$USER1:[SAWATZKY.LMI.PPUMA]MATH.L;1

(threed-trans p transf) : Returns p transformed by transf.
p => three dimensional point (x y z)
transf => transformation (dx dy dz ax ay az).

(twod-trans x transf): Returns the x transformed by transf.
x => 2-dimensional point (x1 x2)
transf => two-d transformation (dx1 dx2 dth).

(rotate x1 x2 th): Returns x1 and x2 rotated by th where x is a 2-dimensional point (x1 x2) and th is an angle in radians.

(deg-t-rad x) : Returns x in radians, where x in degrees.

(rad-t-deg x) : Returns x in degrees, where x is radians.

(sign x) : Returns the sign function of x.

(mean vl) : Returns the geometric mean of vl where vl is a list of n dimensional points ((x1 x2 ... xn) ...).

(sum x) : Returns the sum of x where x is a list of n dimensional vectors ((x1 x2 ... xn) ...).

(add-vectors x y) : Returns the sum of x and y where x and y are two n-dimensional vectors. (x1 x2 ... xn) (y1 y2 ... yn).

(diff-vectors x y) : Returns the difference of x and y where x and y are two n-dimensional vectors. (x1 x2 ... xn) (y1 y2 ... yn).

(normalize x n) : Returns the normalized version of x where x is an n-dimensional vector and n is a scaler. (x1/n x2/n ... xn/n).

(mult-vector x c) : Returns the product of a vector and a scaler where x is an n-dimensional vector and c is a scaler. (x1*c x2*c... xn*c).

(dot-product x y) : Returns the dot product of x an y. x and y are two n-dimensional vectors.

(dot-product-aux x y) : Returns the vector which is a term by term product of x and y. x and y are two n-dimensional vectors.

(maxdist-pt-ptset x xl) : Returns the maximum distance between x and xl where x is an n-dimensional point (x1 x2 ... xn) and xl is a list of n dimensional points ((x1 x2 ... xn) ...).

(twod-det x11 x12 x21 x22) : Returns the determinant of a 2x2 matrix.

(find-twod-line p1 p2) : Returns the coefficients of a twod line (a b c) which passes through p1 and p2, where p1 and p2 are two twod points. p1 (x1 y1), p2 (x2 y2), line ax + by = c.

(solve l1 l2) : Returns the solution to the set of equations defined by l1 and l2.

(equal-pt x y) : Returns t if x and y are equal, where x and y are two n-dimensional points. (x1 x2 ... xn) (y1 2y ... yn)

(midpoint x y) : Returns the midpoint of x and y , where x and y are two n-dimensional points. (x1 x2 ... xn) (y1 y2 ... yn)

(dist x y) : Returns the euclidian distance between x and y, where x y are two n-dimensional points. (x1 x2 ... xn) (y1 y2 ... yn)

(max-ceiling x) : Returns the maximum value of a vector. maximum value is defined as the term with the largest ceiling function value.

(inches-mm x) : Returns a list of numbers converted from inches to millimeters

(asin1 x) : Returns arcsin of x, where x is radians.

(acos1 x) : Returns arccos of x, where x is radians.

(asin2 x) : Returns arcsin of x, where x is radians.

(acos2 x) : Returns arccos of x, where x is radians.

B.6 User Interface

B.6.1 load/store/create worlds

A world can be created interactively or through any text editor. Normally a skeleton data structure for a robot and its environment is created as a data file with a text editor. This skeleton robot environment is then loaded into the PPLAN environment and the geometric properties are defined using some of the tools mentioned in the modelling section. Finally the robot and its environment is saved to a new data file. Much more complete and powerful modelling tools are conceivable but this begins to move into the area of CAD which is rapidly becoming a well established technology. Rather than developing such capabilities into the PPLAN system it may be simpler to establish links to an appropriate CAD environment.

B.6.2 Running program

The following procedure will load the PPLAN software into a LMI LAMBDA LISP Machine. After logging onto the LAMBDA, enter the PPLAN environment by typing

(pkg-goto 'pplan)
If the PPLAN package is undefined enter
  (make-package 'pplan)
  (pkg-goto 'pplan)
Load the system by entering
(load "gord.ppuma;pplan")
(load-pplan)

After all files are loaded into the LISP environment of the package PPLAN, the user is in a window interface to the PPLAN system. From this window all functions of the PPLAN system are available. In addition, communication and interaction with a PUMA-560 robot is available for PUMA 560 applications.

The PPLAN window interface is divided into four panes labelled command pane, status pane, graphics pane and interaction pane. The command pane is a list of actions which the user can choose with the mouse. The status pane is unused at this time, but is available to display status information about the robot and the program. The graphics pane is used to display animation of two dimensional path planning examples. The interaction pane is used to display messages and query the user for information when necessary. An example of a typical PPLAN window is shown in Figure B.1.

A brief summary of mouse selectable commands in the PPLAN system is as follows.

"Plot-world" "Plot world on graphics pane."

"Plot-robot" "Plot robot bsphere representation on graphics pane."

"Plot-obj" "Plot object bsphere representation on graphics pane."

"move-object" "Move an object to a new location."
This allows the user to specify a new location for any object in the graphics display. Location syntax is indicated on the interaction pane. After the object is moved the user is asked if free space is to be recalculated. If free space is not recalculated then the find path command will yield incorrect results. The collision detectors will always yield correct results.

"get obj loc" "assign object location to present robot world location."
This queries the user for an object name. Then user is queried whether he wants to change the location of the object location transformation directly to the x y z of the present location of the robot end effecter. If the user replies yes then he is allowed to supply an offset for the z-direction, after which the program assigns the object location property to x y + z offset 0 0 0. Otherwise, the user is allowed to input any location transformation for the object.

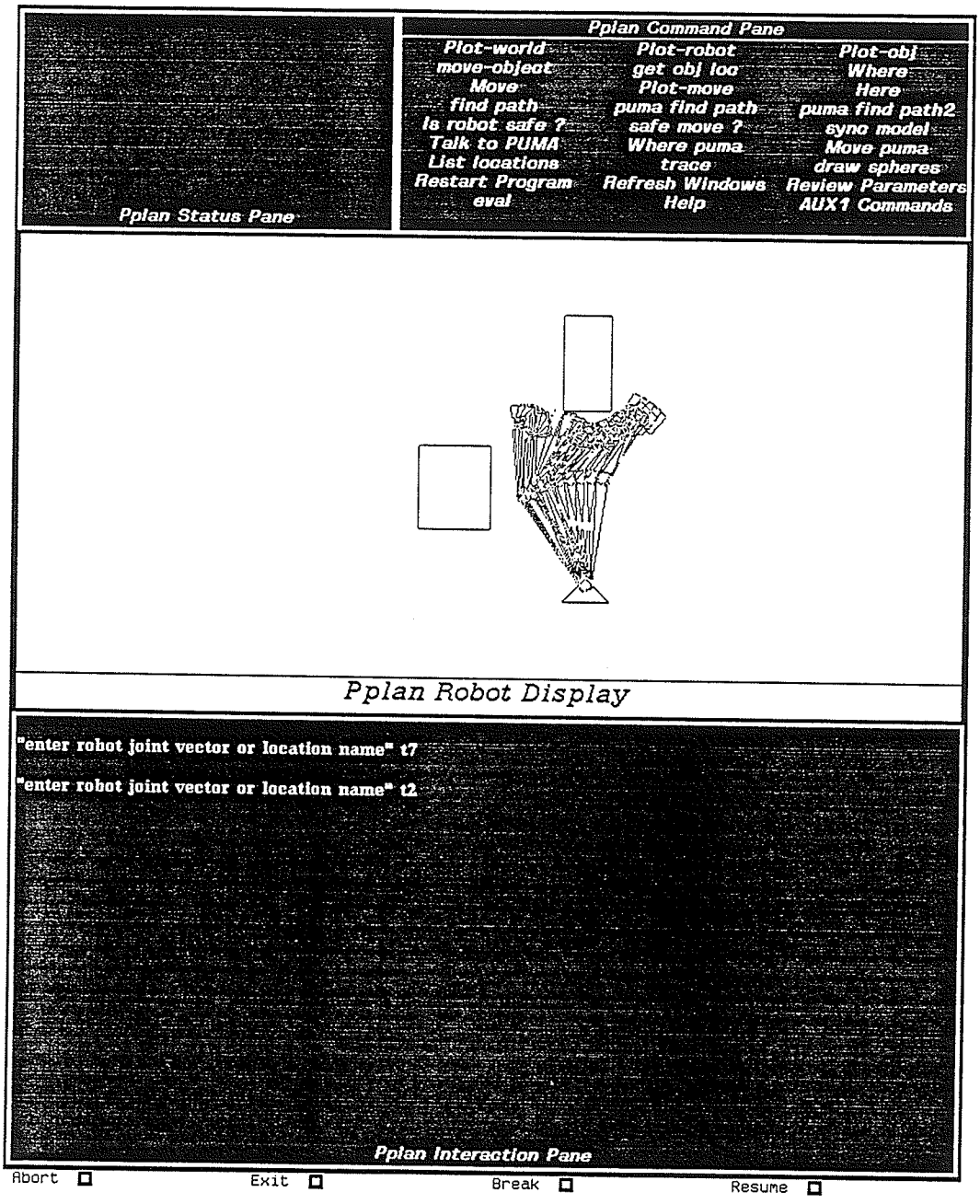


Figure B.1 Window interface for PPLAN system

"Where" "Returns joint angles of robot model."

"Move" "Erase robot, move robot model to new configuration and redraw."

"Plot-move" "display an animation of a robot trajectory."

This function displays an animation of straight-line joint motion for a two dimensional robot. The goal destination can be entered as a list of joint values or a robot location name.

"Here" "Add current robot model configuration to location list."

This queries the user for a location name which is then assigned the value which is a list of the present joint vectors of the robot. Finally the location name is added to the list of locations.

"find path" "find a path to destination and animate the motion."

This function attempts to find a path to a robot destination using a representation of robot free space. If the present or goal destination is not included in free space then the user is notified. If no path in the free space representation exists then the user is also notified.

"puma find path" "Find a path to new location via free space rep if one exists.

"puma find path2" "Find a path to new location by heuristics."

"Is robot safe ?" "report on the state of the robot's geometric interference."

This returns a yes or no response indicating whether or not the robot is geometrically interfering with any objects in the world. This is based on the geometric and kinematic models of the robot and its environment.

"safe move ?" "Checks if a move is safe."

Determine if a straight-line joint motion to a destination is collision free. If the path is found to be clear of collisions then the robot is redrawn at the destination. Otherwise, the robot is redrawn at the first point of collision.

"sync model" "Move robot model to current robot configuration."

"Talk to PUMA" "Virtual terminal communication to the PUMA 560 "

Allows the user to communicate directly to the PUMA robot with a terminal emulation program. Please see VAL user's guide for

description of VAL the operating system of the PUMA robot.

"Where puma" "Returns joint angles of robot."

"Move puma" "Move robot to new configuration."

Query the user for a robot location and move robot to location by defining the location in VAL, using a variable name of #temp, after which the puma is commanded to move in joint-interpolated motion to that location.

"List locations" "List robot locations in the interaction pane"

This returns the list of presently defined robot configurations. The user is queried whether they would like to see all locations. If the user replies yes, then all known robot joint vectors are listed. Otherwise, the user is asked to supply a specific location name and it is returned.

"trace" "Plot-overlapping graphics."

A switch which controls whether the graphics writes over itself or not.

"draw spheres" "Plot spheres used in interference detection"

A switch which decides whether spheres used for interference detection are sent to the graphics display.

"Restart Program" "Abandon everything and start PPLAN from scratch"

Ensures that the selected package is pplan, clears all windows and redraws the robot.

"Refresh Windows" "Refresh all the windows in this display."

Clears all windows and redraws the robot in the graphics pane.

"Review Parameters" "Review parameters, and maybe make modifications"

"eval" "eval a lisp expression"

"Help" "Interactive Help with Pplan"

"AUX1 Commands" "Extra commands, mostly for reading and writing files"

"list world files" "List data files containing object data"

"read new world" "Read in a new robot world into PPLAN package"

"save world" "Save present world state into a file"

"list location files" "List data files containing location data"

"read new locations" "Read in a new robot locations into PPLAN package"

"save locations" "Save present robot locations into a file"

"list fspace files" "List data files containing free space representations"

"read free space" "Read a free space representation from a file"

"save free space" "Save present free space representation into a file"

"open plot file" "Open a file for storing all future vector drawing in the graphics pane. This file can be sent to a qms laser printer.

"close plot file" "Close plot file.

"initialize puma io" "Establish a serial communication channel to PUMA"

"close puma io" "Close serial communication channel to PUMA"

B.6.3 Storage details of PPLAN

At the time of this writing the entire PPLAN system source resides in a single directory.

lama:gord.ppuma;

A brief description of the files is as follows

dirk.lisp : Direct kinematics for PUMA 560.

fdecomp.lisp : F-space construction functions.

findp.lisp : Find-path algorithms for Fspace representations.

findpms.lisp : Find-path algorithms for mixed Cspace representations.

findph.lisp : Heuristic find-path algorithms for PUMA 560.

io.lisp : File input/output functions.

interf.lisp : Interference detection algorithms.

invk.lisp : Inverse kinematic functions for PUMA 560.

kinem.lisp : Kinematics functions.

graphics.lisp : Graphics functions.

math.lisp : General mathematical functions.

motion.lisp : Robot motion functions.

model.lisp : Robot modelling functions.

pplan.lisp : PPLAN loading software.

puma_com.lisp : PUMA 560 serial communication software.

user_int.lisp : User interface functions.

window.lisp : Window interface software.

All robot and obstacle data information resides in a data files under the directory of

lama:gord.pplan;

All robot and obstacle information reside in data files ending in .dat, robot locations reside in files ending in .loc and free space representations reside in files ending in .fsp. Presently there are four two dimensional robot examples and one three dimensional example, namely a PUMA 560. These data files are stored in the following.

horn.dat : Two dimensional robot with three rotational degrees of freedom.

gantry.dat : Two dimensional robot with two translational and one rotational degree of freedom.

mobile.dat : Two dimensional robot with two translational and one rotational degree of freedom.

postf_2d.dat : Two dimensional robot with two translational and one rotational degree of freedom.

puma.dat : Three dimensional robot with six rotational degrees of freedom