FINAL REPORT

PATH PLANNING FOR ROBOTIC TRUSS ASSEMBLY

Arthur C. Sanderson
Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, NY 12180

Submitted to

R. W. Will
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

March 21, 1992 to September 22, 1993

Research Grant No. NAG-1-1413

PATH PLANNING FOR ROBOTIC TRUSS ASSEMBLY

CONTENTS

PART I: Flexible Potential Field Path Planner

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A new Potential Fields approach to the robotic path planning problem is proposed and implemented. Our approach, which is based on one originally proposed by Munger [1] [2], computes an incremental joint vector based upon attraction to a goal and repulsion from obstacles. By repetitively adding and computing these "steps", it is hoped (but not guaranteed) that the robot will reach its goal. A attractive force exerted by the goal is found by solving for the the minimum norm solution to the linear jacobian equation. A repulsive force between obstacles and the robot's links is used to avoid collisions. Its magnitude is inversely proportional to the distance. Together, these forces make the goal the global minimum potential point, but local minima can stop the robot from ever reaching that point.

Our approach improves on a basic, potential field paradigm developed by Munger by using an *active*, *adaptive* field — what we will call a "flexible" potential field. Active fields are stronger when objects move towards one another and weaker when they move apart. An adaptive field's strength is individually tailored to be just strong enough to avoid any collision.

In addition to the local planner, a global planning algorithm helps the planner to avoid local field minima by providing subgoals. These subgoals are based on the obstacles which caused the local planner to fail. A best-first search algorithm A* is used for graph search.

CHAPTER 1
INTRODUCTION

1.1 Goal

This paper addresses the problem of reliable and efficient planning of a collision-free path for a single chain multi-link robot. Our proposed algorithm plans free space moves, i.e., we do not plan paths that incorporate contact motions.

One application of this work is to build truss structures which might be used as space platforms, energy collectors, radiators, in space applications. An example of such a platform is shown in Figure 1.1 [3].

We have tested our algorithm's path planning for two different robot configurations. At the Center for Intelligent Robotic Systems for Space Exploration (CIRSSE) the planner provides a general capability for free space path planning. It is incorporated in a testbed which explores the integration of autonomous robotic algorithms for space operations. At NASA's Langley Space Center, roboticists are also working on the construction of space structures, but the emphasis is on robotic *assistance* to a *human's* task. Here, the planner can relieve a busy human of a very time consuming task.

The goal of our work may be stated as follows. Given:

- Robot's kinematic data,
    1. Modified D-H Parameters,
    2. Joint ranges,
    3. Link models (planes and cylinders),
- Environment's obstacles,
- Joint angle vector for the Start position of the robot,

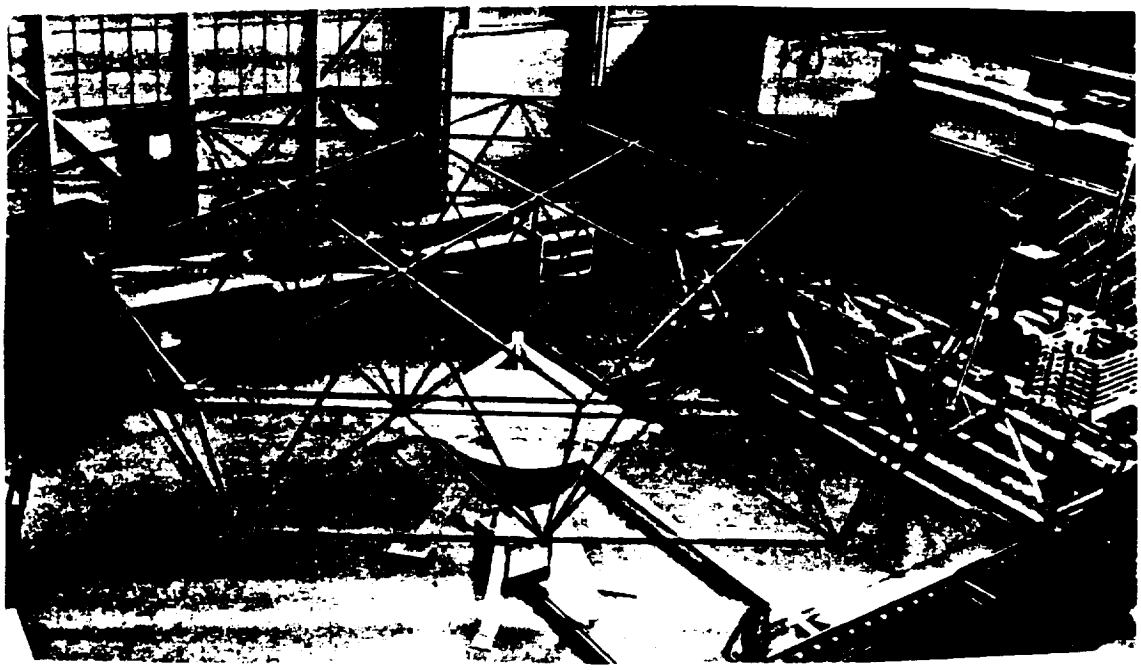


Figure 1.1: Langley Structural Assembly Laboratory

- Position and orientation of the goal's end effector,
- Approach and departure offsets,

the program should produce a sequence of joint angles (called a path) which will be kinematically correct, collision free, and accurate.

Our algorithm is based on a combination of previously proposed planning methods. It is relatively fast and its plans are smooth. Unfortunately, the algorithm is not guaranteed to find a solution, even if one exists, nor is its solution guaranteed to be optimal (with respect to time, distance, or speed). The body of this work strives to increase the planner's flexibility by addressing problems encountered by the previous work. As a result, for relatively uncluttered spaces like those in the two applications listed, acceptable solutions are found.

1.2 Organization of the Text

Chapter 2 reviews other paradigms of path planning. Then it summarizes Rolf Munger's thesis project, from whose work this paper is directly descended. Details

of his work can be found in CIRSSE Report #91, and we suggest that his work be kept at hand while reading this report. Chapter 3 discusses our improvements to Munger's local path planner. Chapter 4 reports our modifications to his global path planner. Chapter 5 discusses the algorithm's modular programming and input files. Chapter 6 describes the CIRSSE testbed and the implementation of the path-planner within that testbed. Chapter 7 describes the NASA Langley testbed, its goals, and the planner's solutions. Finally, we conclude our discussion in Chapter 8 with observations on the results obtained by the planner and suggestions for future improvements.

In the Appendixes, we list our program's files and develop a model for planar objects.

1.3 Nomenclature

Here are some definitions which are used in this report.

DOF Degrees Of Freedom. For a single chain robot: the number of independently moving joints.

redundant If a robot's DOF is greater than the size of the space it is expected to work in, then it is redundant. Essentially it is more flexible because it has the freedom to chose more than one configuration per pose.

pose The combination of the EE's position and orientation in cartesian space.

EE End effector, tool frame, gripper. The part of the robot which carries the tool or payload.

Throughout the text, scalars are printed in *italics*, constants in *ITALICS*, vectors in **bold**, and matricies in **BOLD**. *Occassionally*, key words will be emphasized by putting them in italics, these should be distinguishable from scalars by their

context. In discussing the software development, we will indicate UNIX commands and file names by the **typewriter** font.

CHAPTER 2
SURVEY OF SOLUTIONS TO THE PROBLEM

2.1 Previous Solutions

In the early days of robotics, people planned the paths; robots just followed them. Programmers would enter a sequence of joint values or cartesian points. Then the robot would move, in order, to each knot point. Later, a "teach" method was devised. While the operator drove the robot to the goal, he would save key positions along the way.

These methods have drawbacks. If the goal, the starting point, or any obstacles change after the knot points are saved, then the path would have to be re-entered. These methods also take time and expertise.

Clearly, none of these approaches are desirable for our problem. We want an autonomous, computer generated solution. Presently, most proposed solutions of this path planning problem fall into one of two categories; global planners and local planners.

2.1.1 Global Methods

Global methods consider the entire environment (or a large part of it). They have an advantage over local solutions in that, they will usually find a solution, if one exists. This comes, however, at the expense of computational complexity. In fact, as the degrees of freedom of the robot increase, these methods may quickly become intractable.

The global methods can be categorized as follows:

1. Search through Graphs of the Environment,

   (a) Cell Decomposition Graphs.

(b) Visibility Graphs.

2. Divide, then Conquer the Environment.

3. Apply Calculus of Variations of Optimization.

These techniques are all linked in one key aspect: they require that the environment be mapped onto the robot's joint space. For example: to a 6-DOF robot, what was a point in cartesian 3-space "balloons" into a three dimensional object in the robot's six dimensional joint space. Meanwhile the robot itself has shrunk to a point. The problem is dramatically reduced from planning a path for a 6 dimensional robot in 3 dimensional space to planning a path for a point in six dimensional space, albeit with larger obstacles, see [5] [6] [7] [8]. Unfortunately, the environment's transformation requires inverse kinematics routines, which are notoriously computation-intensive [9] [10] [11].

2.1.1.1 Graph Search Methods

Cell Decomposition methods divide the world into two types of space: free and occupied. The free space cells are put into a graph, and the adjacent cells are connected. Then a search algorithm is applied which finds the shortest path from the "start" cell to the "goal" cell through connected cells. The specific search algorithm used will depend on the type of optimality desired [12] [13] [14].

An alternate way of building the graph is to choose a sufficiently large set of subgoals based on geometrical information. This method requires an algorithm which determines visibility between subgoals. If visibility is established then the subgoals are connected.

2.1.1.2 Divide and Conquer Method

Divide and Conquer methods try to minimize the number of transformations of obstacles to joint space. First, a line is drawn in joint space between the start and the goal. This is our starting path. Search along that line for obstacle collisions. If one is found, then search in the hyperplane normal to the line for a point which is not in occupied space. When such a point is found, draw a line to it from the last safe point found on the original path. Repeat the search along this new line. Continue by connecting the new point to the goal. By adding back recursion, this method will always find a feasible path. It also has the advantage of only transforming points along the path into joint space, see Weaver [4].

2.1.1.3 Calculus of Variations

Finally, there is a more mathematical approach which relies on the calculus of variations to minimize a cost function which usually involves distance to obstacles and path length. These methods are currently intractable with robots of high order [20].

2.1. Local Methods

Local solutions try to solve the "global" problem by repetitively finding incremental changes of the current position which bring the robot closer to the goal. Thus they concentrate only on what is "near" the current state, and ignore the larger picture. These methods are computationally fast, but they may not find a solution if the increments are poorly chosen. In addition, their solutions may not be optimal.

The simplest local method is called the hypothesize and test method. The algorithm generates a joint increment, "step", towards the goal, and if the step is feasible, repeats. If not feasible, if a collision occurs, then it chooses a heuristic step

(perhaps random) and continues. This has not led to great success.

A better method is called the potential field approach. While the goal exerts an attractive force on the robot, obstacles repulse the robot. Each step is taken along the gradient of this potential field. Since there is only one attractor, the goal, it will have the globally minimum potential. Unfortunately, this does not rule out the existence of local minima, which have null gradients and thus null steps [21].

There are many variations of the potential field method. The vortex field method proposed by DeMedio and Oriolo [15] attempts to reduce the problem of local minima by adding a cross product to the repulsion. A method to escape local minima by brownian motion has been reported by Barraquand [16] to have success with many difficult problems. Still others, propose minima-less potential fields based on superquadric potentials, Khosla and Volpe [17] [18], and on star-shaped obstacles, Rimon and Koditschek [19].

2.2 Munger's Hybrid Global and Local Path Planner

As mentioned earlier, this project extends the method proposed by Rolf Munger [1]. This overview will cover the concepts necessary to understand our project's modifications. We will also note some strengths and weaknesses of his approach, but save a complete discussion for Chapters 3 and 4 when we describe our modifications.

To summarize: Munger combines a global graph search over geometrically chosen subgoals and a local potential field approach to determine the visibility between those subgoals. For speed, he uses a single representation for all objects in the environment, the robot's links, the struts, and the world's obstacles.

His work can be divided into four areas:

- Modeling of the World
- Potential Field Path Planning (local)

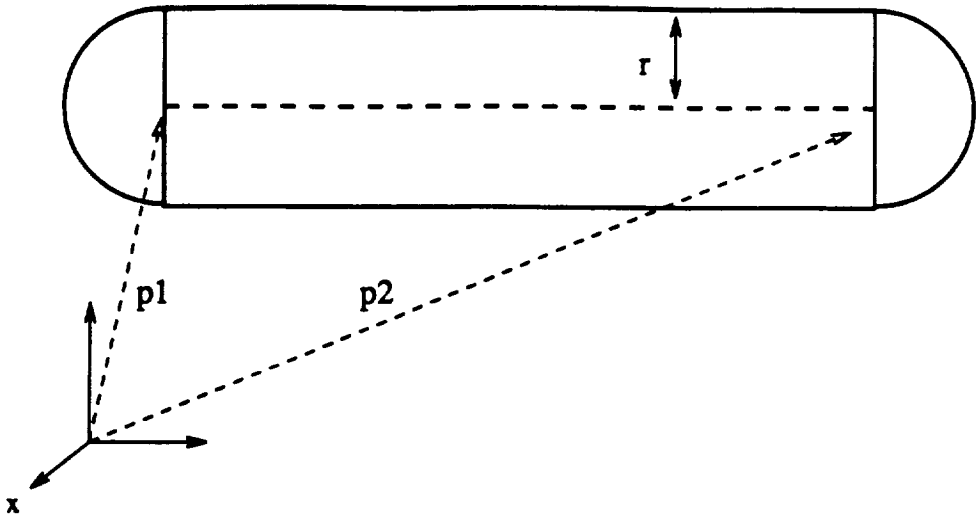


Figure 2.1: Swept Sphere Model for Links, Struts, and Obstacles

- Graph Search Path Planning (global)
- Software Implementation

2.2.1 Swept Sphere Model

All path planners need a model of the physical world to check for collisions. Potential field path planners not only need to determine distances between models in order to calculate potentials, but also need to know the direction from one model to another in order to calculate the direction of repulsion.

Since the repulsions between n moving parts and themselves, as well as between themselves and m fixed parts, need to be calculated at each step for n joints (i.e. $O(n^3) + O(n^2 m)$), the calculation needs to be *fast*. Therefore, Munger chose a very simple model called a swept sphere. Figure 2.1 shows the area swept out by a disk along a line segment.

Note that one weakness of having a single simple model is that complicated obstacles cannot be modeled accurately. Since he needs to guarantee a collision-free

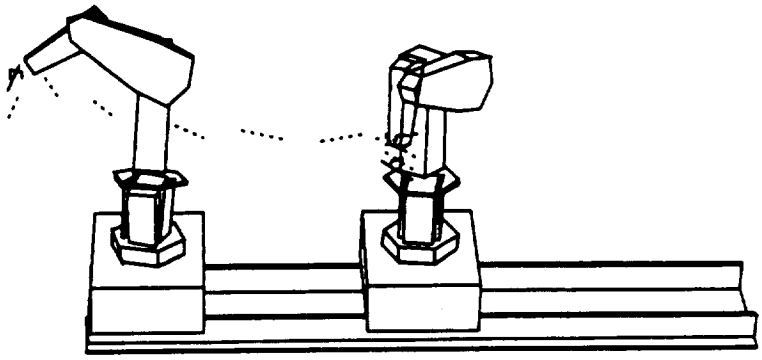


Figure 2.2: Example of Incremental Steps Forming a Path

path, the swept spheres must be made conservatively larger than the actual objects. Therefore, on occasion, collisions will be reported which would not actually occur, and feasible paths will not be found.

In Appendix A, we describe a plane model which helps to simulate more types of world obstacles. Tornero and Hamlin [28] discuss a computationally fast modeling method based on spherical objects.

2.2.2 Local Path Planning with Potential Fields

As previously mentioned, the local planner calculates a joint increment which avoids obstacles and moves towards the goal. This increment is added to the current joint position vector. Then we repeat the process. Thus a "path" is a list of joint increments which if followed sequentially by the robot, moves it from the start to the goal, see Figure 2.2. The dotted lines in the figure trace the path of the payload strut for each step.

Munger's potential field method sets the joint vector increment, $\mathbf{dq}$, sometimes

called the "step", as follows:

$$\mathbf{dq} = \mathbf{dq_{att}} + \mathbf{dq_{rep}} + \mathbf{dq_{range}} \tag{2.1}$$

where

$\mathbf{dq_{att}}$ is the attractive force exerted by the goal. It is found by solving the system of linear equations:

$$\mathbf{J}\ \mathbf{dq_{att}} = \mathbf{dx} \tag{2.2}$$

where $\mathbf{J}$ is the manipulator jacobian and $\mathbf{dx}$ is made by stacking the cartesian direction vector and rotation axis vector which r es the current iteration's gripper pose to the goal's pose. Since for redundant robots (> 6-DOF) this linear equation is underdetermined, there are, in general, many solutions. We choose a solution which minimizes the cost function

$$cost(\mathbf{x}) = \mathbf{x^T Q x}$$

$\mathbf{Q}$ is a positive definite diagonal weighting matrix.

This standard optimality problem (see ref) is solved by introducing a lagrangian vector λ. The attractive force solution is

$$\mathbf{dq_{att}} = \mathbf{Q^{-1} J^T} \lambda \tag{2.3}$$

where λ is found from

$$\mathbf{J Q^{-1} J^T} \lambda = \mathbf{dx} \tag{2.4}$$

Since $\mathbf{J Q^{-1} J^T}$ is square, this linear equation can be solved by gaussian elimination and back substitution. Instead of gaussian elimination, however, Munger uses Householder Transformations, which, while slower, have better numerical properties [22].

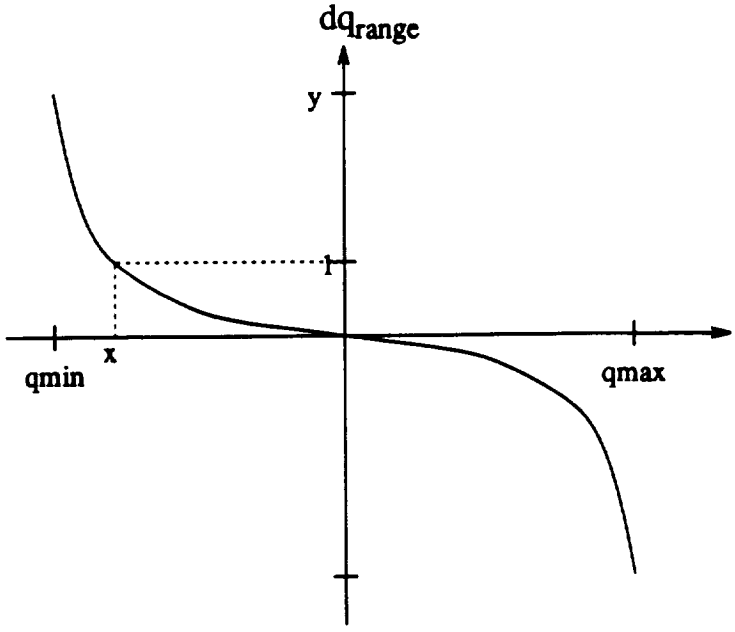


Figure 2.3: Joint Limit Repulsion

$\mathbf{dq_{rep}}$ is the collision repulsion vector. Since a joint affects all the links later in the chain, the repulsion felt by each joint is equal to the sum of the repulsion contributions of each later link. The repulsion contribution of each link is the sum of the repulsions, $\delta\mathbf{q}$, between that link and every obstacle in the environment. The repulsion between a link and a obstacle is

$$\delta\mathbf{q} = C\frac{\mathbf{r}\cdot\mathbf{s}}{\mathbf{d}^2} \tag{2.5}$$

where C is a constant, $\mathbf{r}$ is the unit vector from the link to the obstacle, $\mathbf{s}$ is the unit direction that the joint moves the link, and d is the distance between the link and the obstacle. Note that this is an inverse square law much like gravity or electro-magnetism.

$\mathbf{dq_{range}}$ tries to keep the joints within the physical limitations of a particular robot by increasing rapidly when the current joint position nears its limit. The repulsion is defined as shown in Figure 2.3. Through trial and error, x and y were set to be $.8q_{min}$ and 10, respectively.

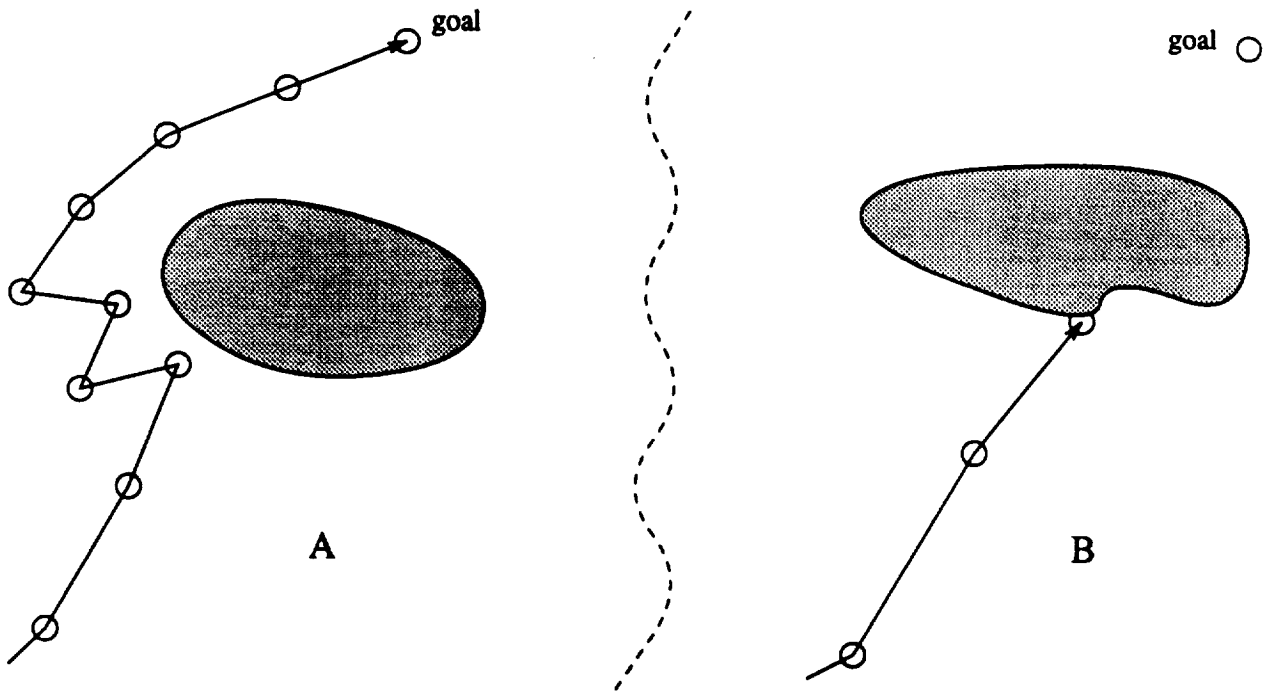


Figure 2.4: Bad Step Sizes (A. oscillation B. collision)

Because the translational and rotational motions of the robot do not necessarily finish at the same time, Munger also provides a method to reduce vectors to the null space of the manipulator jacobian's translational or rotational part (top or bottom half, respectively). If **dq** is in the null space of **J** then it does not affect the pose of the gripper **dx**, but if the robot is redundant, it can still move some links of the robot away from obstacles. This is called "self-motion". Working with the null space of the top or bottom of **J** allows the goal position to be held while the robot finishes rotating or for the goal orientation to be held while the robot finishes translating, respectively.

There are three major weaknesses with Munger's solution. First, there are many "magic" numbers which need to be balanced with one another by trial and error. Second, the iterative nature of the plan produces discrete steps which, if too

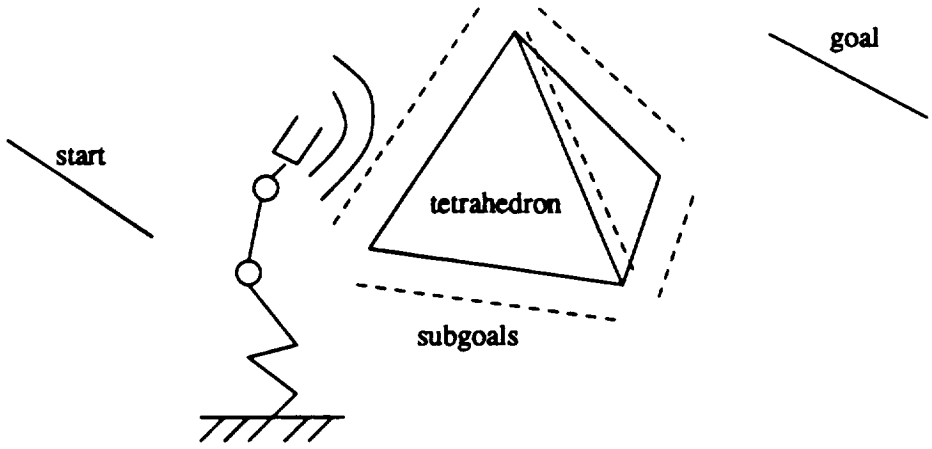


Figure 2.5: Global Subgoals Help Local Planner

large, can cause oscillations, collisions, or joint overruns (see Fig 2.4). Finally, there may be places where $\|\mathbf{dq}\|$ approaches zero, trapping the robot in a local minimum.

Taken together, these problems make the algorithm inflexible, prone to failure, and slow. The majority of this project goes towards remedying these ailments.

2.2.3 Graph Search Around Obstacle Corners

Munger uses a global algorithm to assist his potential field path planner.. He employs the graph search method. First, a list is created which includes the starting location and the goal. Then since he deals with tetrahedral trusses, subgoals are put at the corners of tetrahedra. These subgoals are likely to position the robot to avoid the tetrahedra.

Next, all the nodes in the list are connected and each connection is given a weight which is the lower bound of the cost to traverse that connection, i.e., the distance (plus an angular rotation factor). Now, any search algorithm can be applied

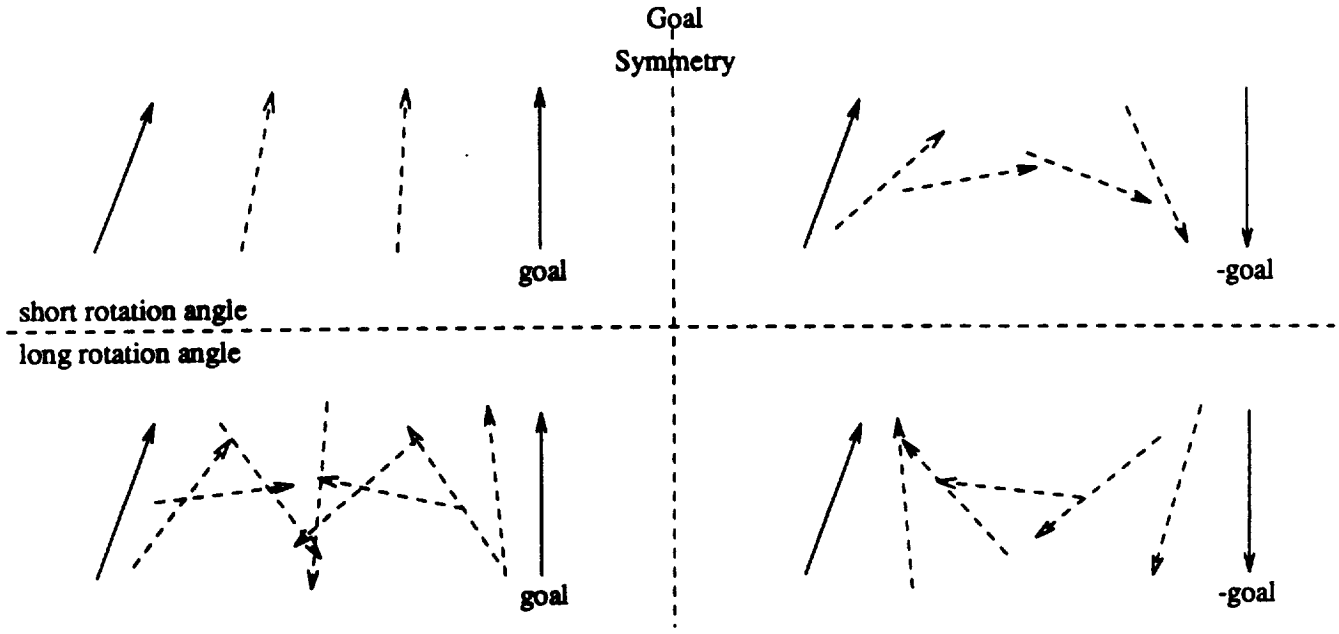


Figure 2.6: Four Ways to Rotate from Start to Goal

to find a path through the connections from start to goal. The A* algorithm was chosen for its speed in finding the lowest weighted path [23] [24].

The A* algorithm is applied, and a trial list of connections is generated. Clearly, since all nodes are connected, the goal and the start are connected, and this trivial connection will always be the first solution tried.

Each connection in the list must be tested, in order, for visibility by the local path planner. If the local planner cannot find a path for the connection, then that connection is labeled "invisible", and A* will no longer include it in subsequent solutions.

Before a connection is considered invisible, however, the local planner is given four opportunities to find a feasible path for it. This includes inverting the goal's orientation of symmetric payloads and rotating the payload the long way around

the axis of rotation (see Fig 2.6).

Applying global search in this way greatly expands the ability of the local path planner, but the global search is only as good as the choice of subgoals. If the subgoals have bad orientation or approach directions or are near too many obstacles, they may be invisible. If there are too many invisible subgoals, the algorithm will be very, very slow.

2.2.4 Software Implementation

The structure of the software was left essentially unchanged except for the interface with CIRSSE and will be explained fully in the chapter on software, Chapter 5, and the chapter on CIRSSE, Chapter 6.

CHAPTER 3
LOCAL PATH PLANNER

This chapter is the start of the original work done for this project.

3.1 Pseudo-Potential Fields

The simplicity of Equation 2.1 is deceptive. In nature, forces like gravity are applied continuously in time, but our pseudo-potential fields are applied discretely, in steps. As a result, oscillations, collisions, and joint-range overruns can occur from one step to another because the steps are not well suited to the repulsion field. In addition, the repulsive field treats objects as point "masses" located at their points of closest approach, instead of as distributed masses over their entire volume. This can cause "rocking" when one point on the object is pushed away only to bring another point on the same object too close to the obstacle; resulting in a back and forth motion like a see-saw.

We have made four improvements to Munger's path planner: more robust path finding, shorter computation time, more accurate goal pose acquisition, and smoother paths.

3.2 Robust Path Planning

3.2.1 Resolving Singularities

To obtain $\mathbf{dq_{att}}$, Munger used the jacobian to transform a cartesian attraction vector to a joint space vector. In solving the optimality problem that arises, (Section 2.2.2), he failed to account for the robot's singular positions, i.e., where the jacobian looses full rank; where there exists a vector in cartesian space which no joint space vector can effect. Mathematically: if $rank(\mathbf{J}) < 6$ then $rank(\mathbf{JQJ^T}) < 6$

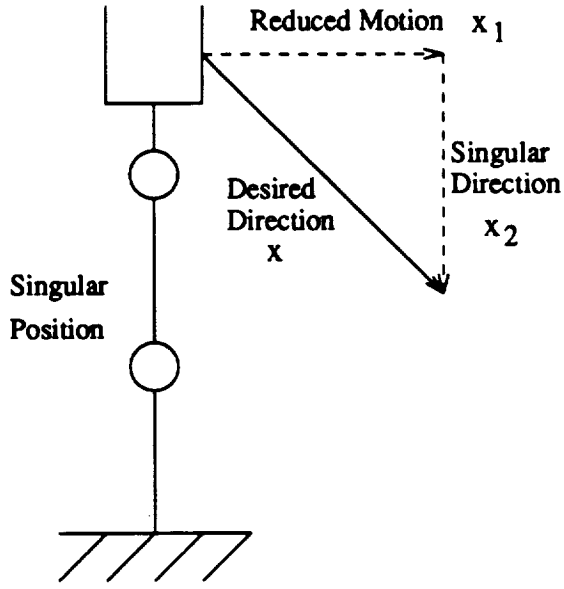


Figure 3.1: Singular Position Limits Motion

which implies that $\mathbf{JQJ^T}$ is singular. If $\mathbf{JQJ^T}$ is singular, we cannot solve Equation 2.4 for λ. If we cannot solve for λ, then we cannot solve Equation 2.3 for $\mathbf{dq_{att}}$. Thus, the planner does not move towards the goal; it fails.

Our solution is to always make the jacobian full rank. First, we check the jacobian's rank. If it is degenerate, we delete rows of the jacobian until it attains full rank. Since the new jacobian $\mathbf{J_{new}}$ is fat (more columns than rows) and $\mathbf{Q}$ is also full rank (positive definite), $\mathbf{J_{new}QJ_{new}^T}$ will be full rank. Finally, we can solve for $\mathbf{dq_{att}}$ as before.

$\mathbf{dq_{att}}$ *algorithm:*

1. Using Munger's algorithm, solve for $\mathbf{dq_{att}}$
2. If his algorithm returns without error then quit.
3. Else, apply gaussian elimination on Equation 2.2 and obtain:

$$\mathbf{J'\,dq} \;=\; \mathbf{dx'} \tag{3.1}$$

where

$$\mathbf{J'} \;=\; \begin{pmatrix} j'_{1,1} & j'_{1,2} & \cdots & & j'_{1,DOF} \\ 0 & j'_{2,2} & \ddots & & \vdots \\ \vdots & \ddots & j'_{n,n} & \cdots & j'_{n,DOF} \\ 0 & 0 & 0 & \ldots & 0 \end{pmatrix}$$

$$\mathbf{dx'} \;=\; \begin{pmatrix} \mathbf{dx_1} \\ \mathbf{dx_2} \end{pmatrix}$$

Let n be the rank of the jacobian, $\mathbf{dx_1}$ be $n \times 1$, and $\mathbf{dx_2}$ be $(6-n) \times 1$.

4. Drop $\mathbf{dx_2}$ and the null rows of $\mathbf{J'}$ to obtain a new formula,

$$\mathbf{J''\,dq = dx_1} \tag{3.2}$$

5. Since $\mathbf{J''}$ is full rank, $\mathbf{J''QJ''^T}$ is also full rank. Solve for $\mathbf{dq}$ by using Munger's minimum norm linear equation algorithm.

—end of algorithm—

In step #3, we use a standard gaussian elimination with scaled partial pivoting algorithm which can be found in many texts on computational mathematics [26].

Note: if $\mathbf{dx_2}$ is non-zero, then it represents an unrealizable component of the desired direction $\mathbf{dx}$ (see Figure 3.1).

By making the jacobian full-rank, the planner no longer fails whenever the robot is at or near a singular configuration. This greatly improves performance since redundant robots have many singularities throughout their workspace.

The planner will still fail, however, when the desired direction and the singular direction are precisely the same. In such a case, $\mathbf{dx_1}$ will be a null vector, and the algorithm will return a null vector for $\mathbf{dq_{att}}$. Not only is it very unlikely for the vectors to coincide exactly, but even if they do, the other components of $\mathbf{dq}$, $\mathbf{dq_{rep}}$ and $\mathbf{dq_{range}}$, will probably push the robot away from this situation. We have not encountered this problem in our experiences with the planner, but for less flexible robots (example: those with 6-DOF) this problem may occur more often. A possible solution would be to choose a small random $\mathbf{dq_{att}}$ whenever the $\mathbf{dq_{att}}$ *algorithm* fails.

This rank reduction step is fairly complicated, $O(n^3)$, where $n = DOF$, but since the DOF of most manipulators is fairly small, the time consumed is not noticeable on a per iteration basis.

3.2.2 Joint-Range Excursion

Any real robot's joints have constrainted ranges of motion. If the planner's planned path does not stay within these limits then its path is not considered feasible.

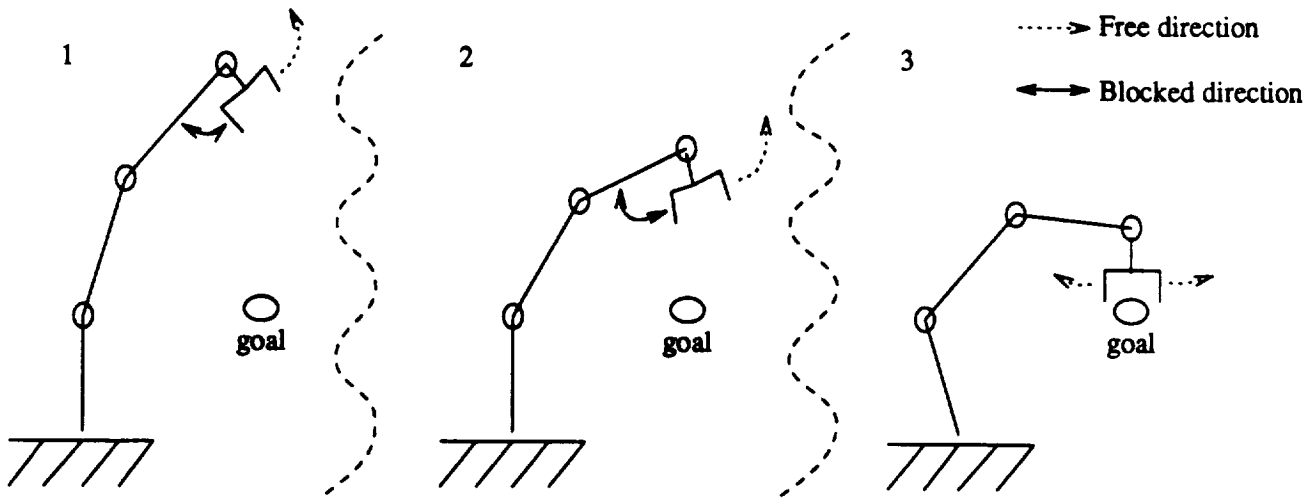


Figure 3.2: Joint #3 is Frozen

Munger's solution proposed an inverse square law repulsive force near joint limits. There are two problems. First, if the repulsion constant is too small, then the joint can go out of range. Second, if the repulsion is too large, then the "flexibility" of the robot's joints will be lowered, and more local minima will be created. Moreover, there is no single setting which will be "just right" for all cases.

We solved this problem by combining a small joint limit repulsion with an algorithm for "freezing" any joint which exceeds its limit.

Joint Range Freezing Algorithm:

1. Calculate **dq** from Equation 2.1.
2. Add **dq** to the *current* joint vector.
3. If this *new* joint vector is in the robot's range, then quit.
4. Else, if the i^{th} joint is out of range, then zero the i^{th} column of the jacobian, the i^{th} column of $\mathbf{dq_{rep}}$ and the i^{th} column of $\mathbf{dq_{range}}$.

5. Re-solve $\mathbf{dq_{att}}$ using the $\mathbf{dq_{att}}$ *algorithm.*

6. Calculate $\mathbf{dq}$ from Equation 2.1 using the new $\mathbf{dq_{att}}$, $\mathbf{dq_{rep}}$, and $\mathbf{dq_{range}}$.

7. Goto Step #2

—end of algorithm—

re-solve Equation 2.2 for $\mathbf{dq_{att}}$, add $\mathbf{dq_{rep}}$ and $\mathbf{dq_{range}}$ (saved from the first iteration), and repeat. By zeroing a column of the jacobian, Step #4, the joint corresponding to that column has no effect on the end effector's pose. Then, by virtue of finding the minimum cost solution, we can guarantee that the i^{th} element of $\mathbf{dq_{att}}$ will be zero. (Note: the i^{th} elements of $\mathbf{dq_{rep}}$ and $\mathbf{dq_{range}}$ must also be zeroed).

This algorithm completely eliminates the problem of joint overruns while maintaining good flexibility. Figure 3.2 shows a typical case where joint overruns might occur. The wrist, joint #3, is strongly attracted to the goal because it has such a strong effect on the EE pose. As a result it has reached its limit and must be frozen. The goal-ward rotation, however, is continued by the other joints, which take on more of the burden of rotating. Finally, the wrist is freed to move again when the goal no longer attracts it against its limit.

3.2.3 Active, Adaptive, Flexible Potential Fields

This section proposes fundamental changes to the repulsion field calculation in order to minimize the number of failures due to local minima.

Munger proposed a "static" repulsion field. It relies on the constant, C, in Equation 2.5, to scale the repulsion field of obstacles with respect to the attraction field of the goal. This "universal constant" does not adjust to varying attractive field strength nor to large clusters of obstacles. An "adaptive" repulsion would make itself only as large as was necessary to stop a collision.

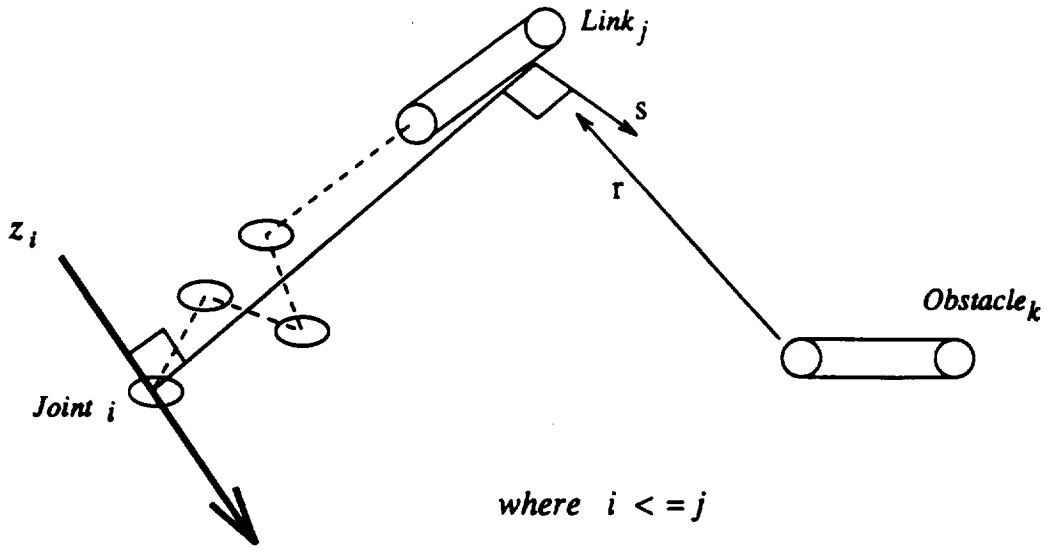


Figure 3.3: Active and Passive Obstacle Repulsion

His proposal is also "passive". The repulsion between objects is the same whether they are moving towards or away from one another. Hence, objects which are already moving apart avoid each other unnecessarily. An "active" scheme repulses more when objects are moving towards one another and less when they are moving apart.

Our active repulsion selects a strong adaptive repulsion if a link is moving towards an object, and it chooses a weak static repulsion if they are moving apart. How does it determine relative motion? Referring to Figure 3.3, **r** is the direction vector from the stationary obstacle to the moved link. **s** is the direction that $Link_j$ moves when $Joint_i$ rotates in the positive right hand sense about its axis $\mathbf{z_i}$. **r** and **s** are unit vectors. (While this figure shows a revolute joint, our proposed method works equally well with prismatic joints, with only minor modifications.)

Obstacle Repulsion Algorithm:

1. Let $j = DOF$; let $\mathbf{dq} = \mathbf{dq_{att}}$; let $\mathbf{dq_{rep}}$ equal the null vector.

2. Calculate $\mathbf{r}$ and the distance between $Link_j$ and $Obstacle_k$.

3. Let $i = 1$.

4. Calculate $\mathbf{s}$ for $Link_j$ and $Joint_i$.

5. If $sign(sign(\mathbf{r} \cdot \mathbf{s})sign(dq_i))$ is non-negative then the link is moving towards the obstacle; calculate dq_{rep_i} using passive repulsion, from Equation 2.5:

$$dq_{rep_i} \leftarrow dq_{rep_i} + C\frac{\mathbf{r} \cdot \mathbf{s}}{\mathrm{d}^2} \tag{3.3}$$

6. Else, the links are moving together; calculate dq_{rep_i} using adaptive repulsion, as follows:

$$dq_{rep_i} \leftarrow dq_{rep_i} + (THRESHOLD)^2\frac{\mathbf{r} \cdot \mathbf{s}\,|dq_i|}{d^2} \tag{3.4}$$

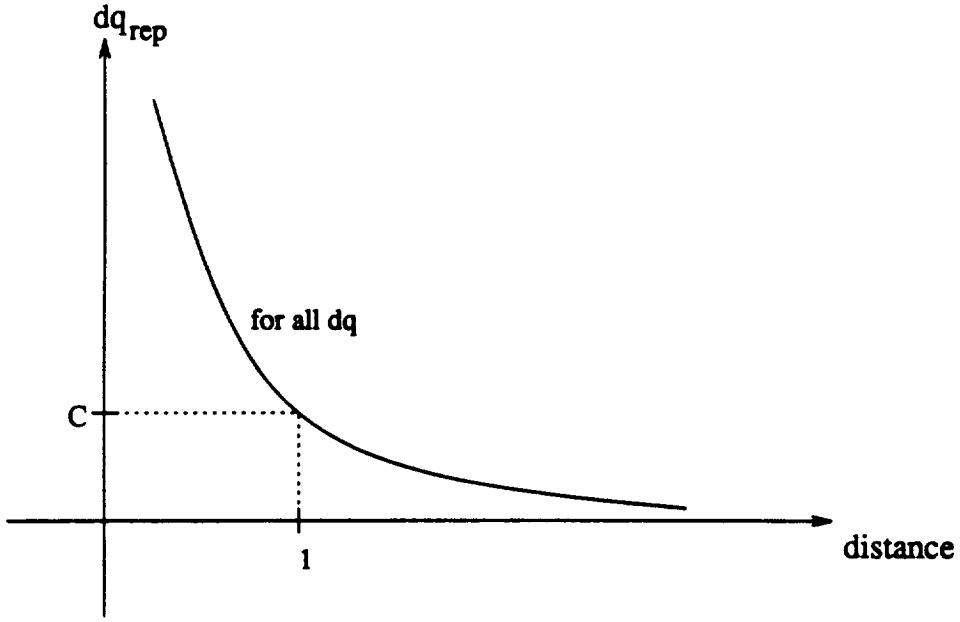
   With the addition of the dq_i term, the repulsion becomes normalized about $THRESHOLD$. If $d < THRESHOLD$ and the link is moving directly at the obstacle, i.e., $|\mathbf{r} \cdot \mathbf{s}| \approx 1$, then $|dq_{rep_i}|$ becomes greater than the attraction force, $|dq_i|$, and the links move apart. Compare this adaptive force with the static force in Figure 3.4.

7. Let $dq_i \leftarrow dq_i + dq_{rep_i}$. This keeps the joint increment current for future calculations of the adaptive force.

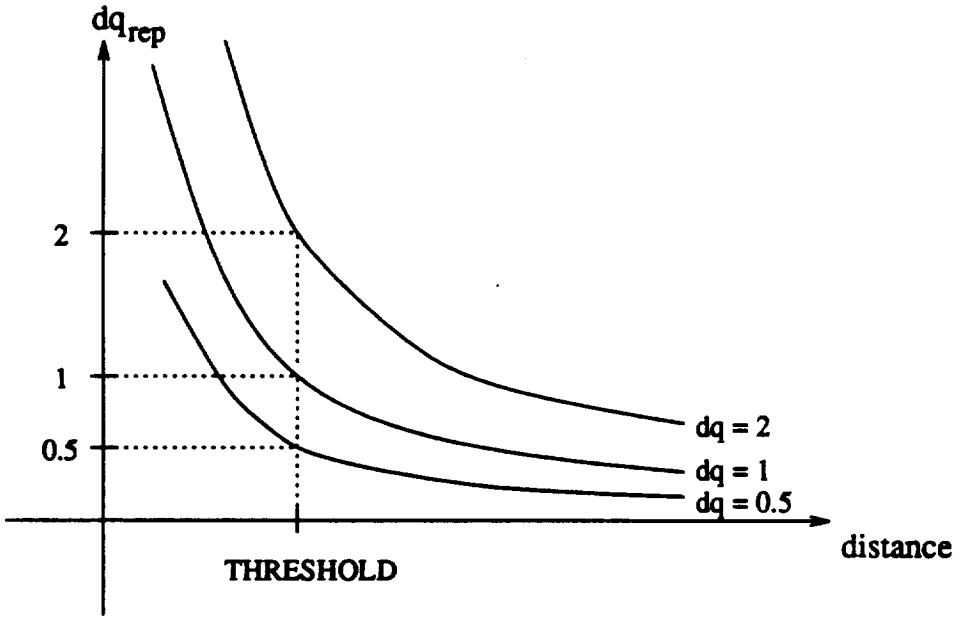
8. If $i \leq j$ then let $i \leftarrow i + 1$; goto Step #4.

9. Until every obstacle has been tested, increment k and goto Step #2.

10. Let $j \leftarrow j - 1$; goto Step #2.



Static Repulsion



Active Repulsion

Figure 3.4: Adaptive and Static Repulsion

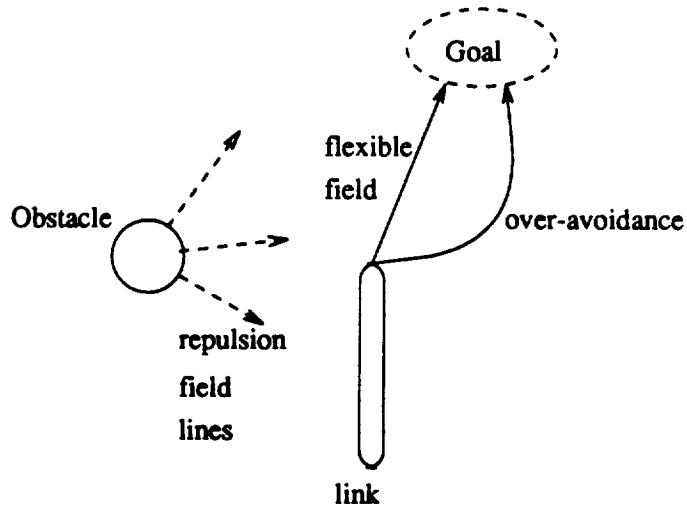


Figure 3.5: Flexible Fields do not Over-Avoid Obstacles

11. report $\mathbf{dq_{rep}}$.

—end of algorithm—

It is difficult to quantify the improvement due to flexible fields. Qualitatively, however, there are three visible improvements to the planner. Figure 3.5 shows a virtue of the active nature of our flexible field. Since the link is moving away from the obstacle, our flexible field *actively* selects the low, passive repulsive force, and the link moves directly towards the goal. If a passive field were used, the link would unnecessarily avoid the obstacle.

A second improvement is shown in Figure 3.6. Since the adaptive field repulses only as much as necessary to keep the link farther than the threshold distance, when the link is between two obstacles, it does not oscillate, rather the repulsions adapt to keep the link where it is. With the static field, the closer obstacle's repulsion is stronger, so the link moves towards the farther obstacle, then this repeats in an oscillatory manner.

The best improvement is also difficult to quantify; fewer collisions occur. A well chosen threshold distance can make collisions very rare. Still there are some situations where collisions will occur no matter how large the threshold is made.

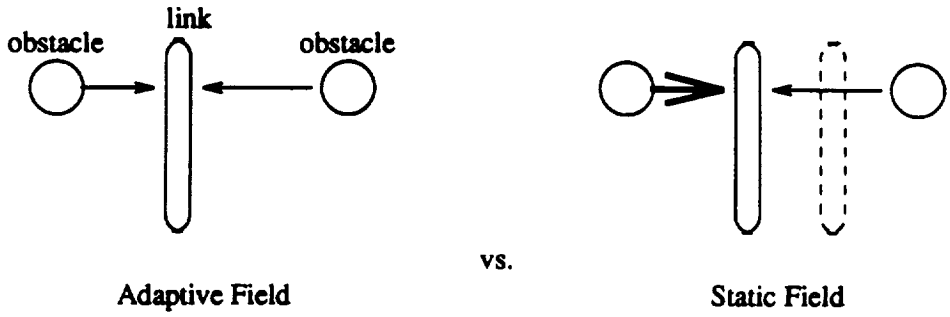


Figure 3.6: Adaptive Fields do not Oscillate

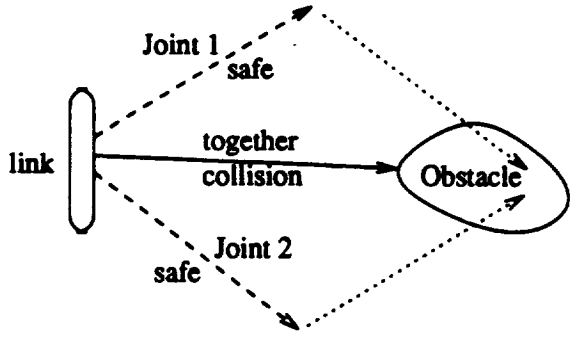


Figure 3.7: Superposition of Joint Effects Causes a Collision

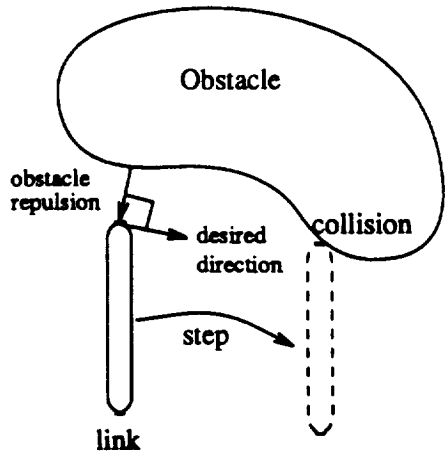


Figure 3.8: Grazing Collision

For example; since the repulsions are calculated for one joint at a time, two joint moves, each of which is safe alone, can cause a collision when combined, Figure 3.7. Figure 3.8 shows a situation where a lumped mass obstacle is insufficient to avoid a collision. Since the link's desired direction is normal to the obstacle's closest point, $\mathbf{r} \cdot \mathbf{s}$ is zero. The repulsion is zero; the link collides with the obstacle.

3.2.3.1 Clusters

When there are too many obstacles grouped closely together, their overlapping repulsive fields can become too large for the robot to approach. If this "cluster" is near the goal, then the planner may report a local minimum even though there is a direct, feasible path to the goal.

We solve this problem by reducing groups of obstacles to a single "cluster", having a single repulsive field. Figure 3.9 shows two obstacles being combined into and replaced by one cluster. The *Cluster Formation Algorithm*, which describes how the environment's obstacles are converted into clusters, is called in place of Step #2 of the *Obstacle Repulsion Algorithm*. Therefore, clusters are defined for each individual link and dissolve after each step is calculated. They are always

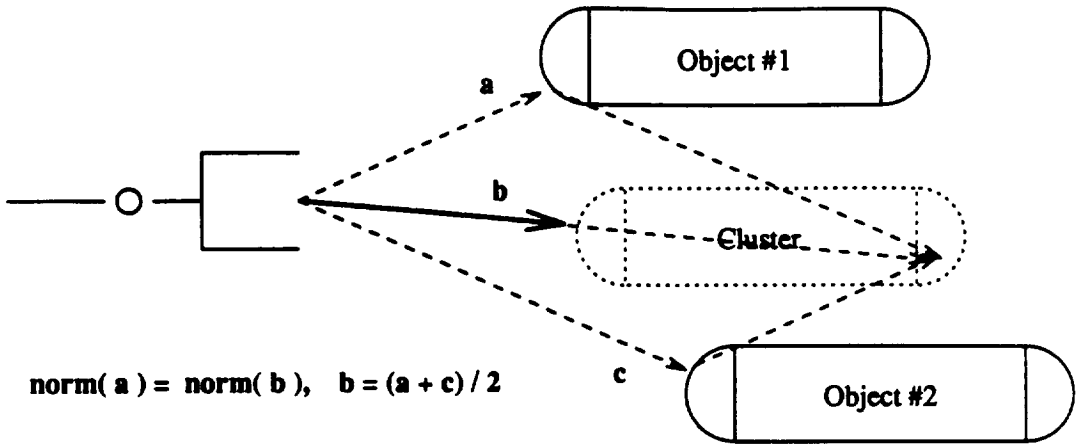


Figure 3.9: Cluster Formation from Objects

"up-to-date", and thus, do not significantly diminish the resolution of the obstacle models.

Cluster Formation Algorithm:

1. Choose any two objects, i and j, from the environment.

2. Find the vectors from the current link to objects i and j. Call these vectors **a** and **c**, respectively.

3. If

$$\frac{\mathbf{a} \cdot \mathbf{c}}{\|\mathbf{a}\|\|\mathbf{c}\|} > 1 - SMALL\ ANGLE. \tag{3.5}$$

then the pair (i, j) is a cluster; replace objects i and j with a new object i (same name) with the following characteristics:

(a) The *direction* of the new object is **b**.

$$\mathbf{b} = \frac{(size_i)\,\mathbf{a} + (size_j)\,\mathbf{c}}{size_i + size_j} \tag{3.6}$$

where *size* has been defined in previous iterations as follows:

(b) Initially, the *size* of all objects is 1. New objects follow the formula: $size_i \leftarrow size_i + size_j$.

For example: if a cluster formed from two objects (as in Figure 3.9) combines with another individual object, then the resulting cluster (object) has a size of three.

(c) Let $\|\mathbf{b}\| = \min(\|\mathbf{a}\|, \|\mathbf{c}\|)$. The new object's *distance* is as close as the nearer of the two original objects.

4. Repeat steps #2 and #3 for all remaining object pairs i and j.

5. If any clusters were found then repeat steps #2–4 (in order to group clusters together into bigger clusters).

6. Report remaining objects' (clusters') direction vectors and distances to the *Obstacle Repulsion Algorithm.*

—end of algorithm—

The advantage of using the *size* weighting system is that it eliminates the pathological case where a long string of obstacles is reduced to a single, unrepresentative cluster. The weighting factor essentially provides a "center of mass" for the cluster.

Clustering greatly improves the planner's performance in crowded environments. Munger's planner failed whenever too many obstacles were near the goal. His planner avoided the obstacles so well that the robot never reached the goal itself. Clustering completely eliminates this problem. A cluster of obstacles can be approached as closely as a single obstacle: to within $THRESHOLD$ meters.

A problem does arise, however, when too many obstacles are loosely called clusters. Figure 3.10 illustrates what may happen if $SMALL\ ANGLE$ is made too large. Obstacles *a*, *b*, and *c* form the cluster *abc* which does not adequately repulse the link from obstacle *a*. Thus a collision with *a* occurs on the next iteration.

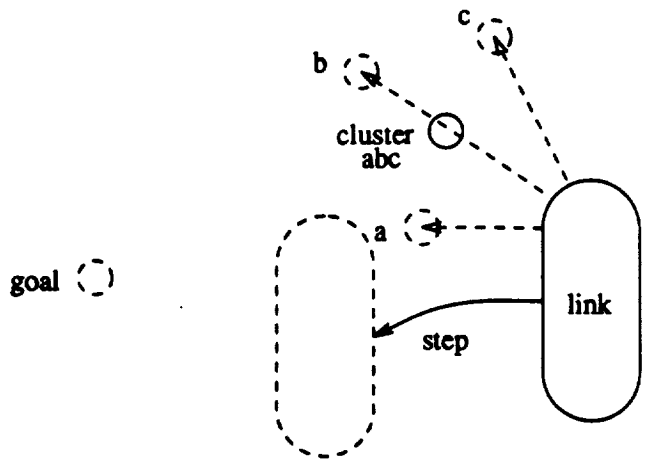


Figure 3.10: Cluster Causes a Collision

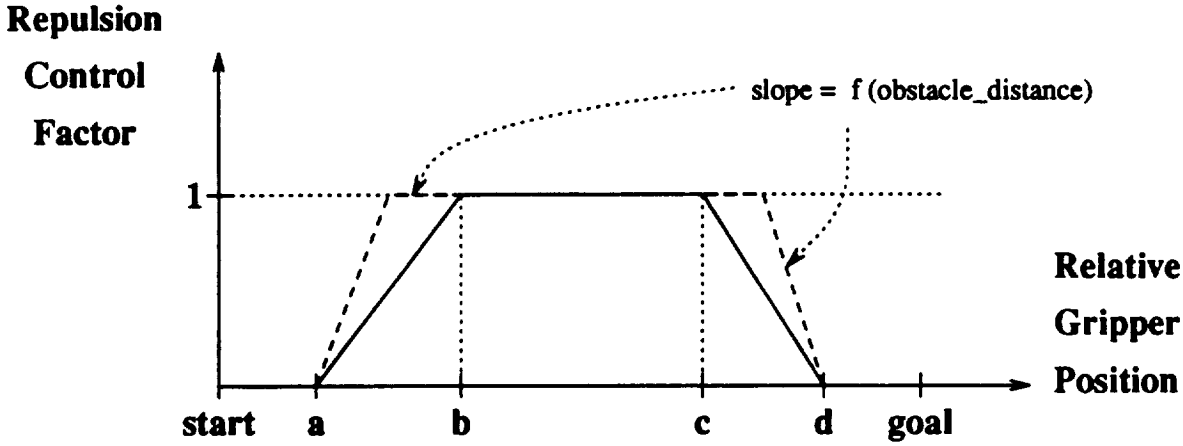


Figure 3.11: Master Control Over Repulsion Strength

The cluster's temporary nature, which makes it accurate, also slows down the planner. Calculating clusters is complicated. If n is the number of links and m is the number of obstacles, then the average complexity is $O(nm^2)$. But the worst case complexity (due to truly staggering bad luck) is $O(nm^4)$. Clearly, we must keep the number of obstacles m low.

3.2.3.2 Overall Repulsion Control Factor

If a goal is within the adaptive distance threshold of any obstacle, then the end effector cannot reach that goal since the adaptive repulsion is larger than the goal attraction. This problem was solved by modifying the joint increment equation, Equation 2.1, with an overall repulsion control parameter, ρ. The new joint increment equation is

$$\mathbf{dq} = \mathbf{dq_{att}} + \rho\,(\mathbf{dq_{rep}} + \mathbf{dq_{range}}) \tag{3.7}$$

ρ is defined as shown in Figure 3.11. It is zero whenever the EE pose is near the goal or the start so that obstacles near the goal and start can be approached. It has a value of one (full repulsion) during the middle of the path for maximum obstacle avoidance. Finally, there are two transition periods which turn on and off the repulsion fields. The slopes of the transitions are inversely proportional to the minimum distance between any obstacle and the robot.

This factor may allow some collisions near the endpoints of the path, but this small price gains us the ability to reach goals which are also obstacles; for example, when we wish to pick up a strut.

3.2.4 Variable Step Size

The norm of the joint increment **dq** determines the size of the step. Munger's step size is set arbitrarily to a constant value. If the constant is too large, Figure 2.4, oscillations or collisions result. If the constant is too small, then the algorithm will be slowed by unnecessary iteration computations.

A simple solution is to reduce the maximum step size when the robot is near obstacles.

Step Size Algorithm:

1. Calculate **dq** step.

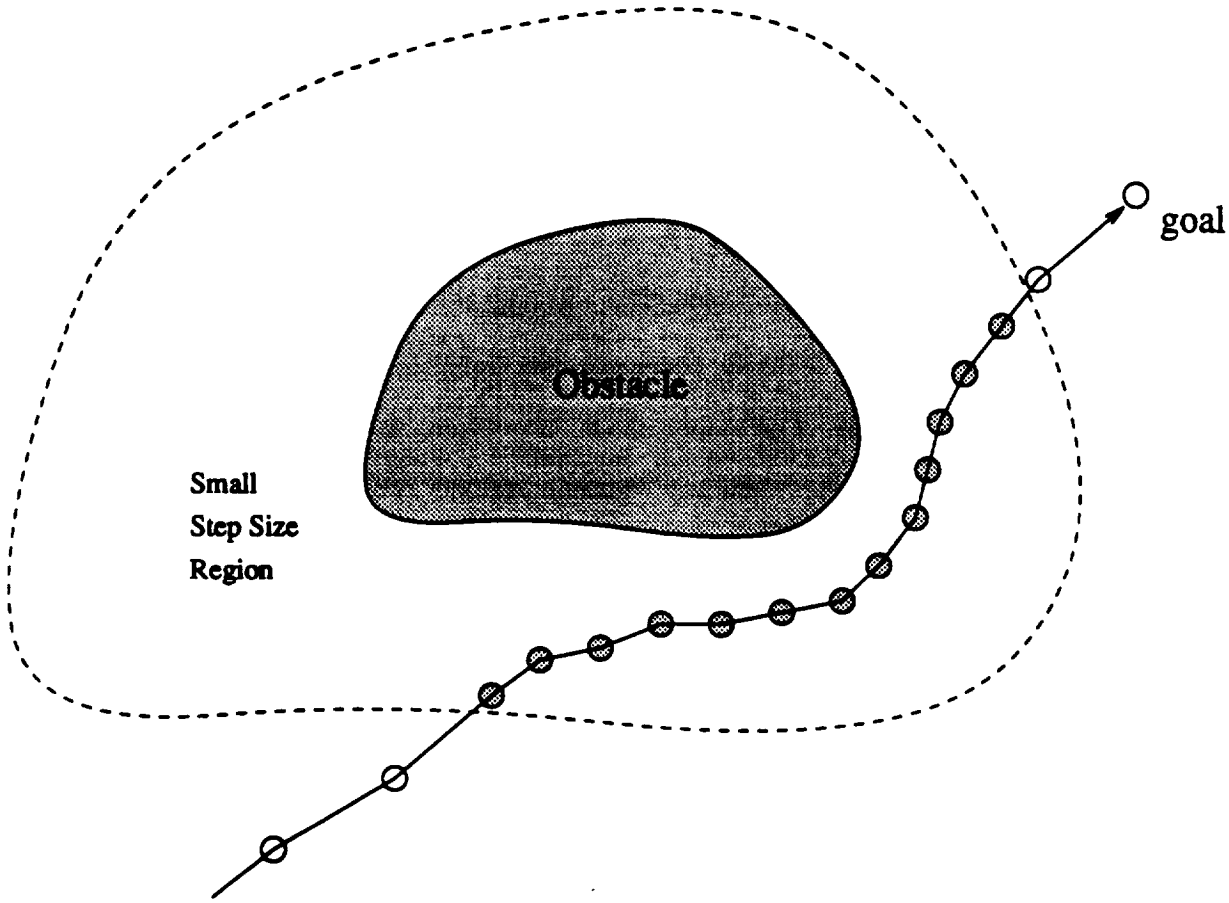


Figure 3.12: Smaller Step Size Near Obstacles

2. Calculate the maximum step size, as a function of the repulsion control ρ:

$$max\ step\ size = (MAX\ SIZE - MIN\ SIZE) + \rho\,(MIN\ SIZE) \quad (3.8)$$

3. If $\|dq\| > max\ step\ size$ then

$$\mathbf{dq} \leftarrow \frac{\mathbf{dq}}{\|\mathbf{dq}\|} max\ step\ size. \quad (3.9)$$

4. If any robot's link is closer than $MINIMUM$ meters then

$$\mathbf{dq} \leftarrow \frac{\mathbf{dq}}{\mathbf{FACTOR}} \quad (3.10)$$

where $FACTOR$ is currently equal to 2.

—end of algorithm—

Thus when ρ is zero, the step size is at a minimum, and when $\rho = 1$, the step size is at its maximum. The logic is that when there is full repulsion, we can be confident that moving fast will not cause a collision, but when there is less repulsion, we should be more cautious and move slower.

Figure 3.12 shows the typical solution to the problem. The reduced step size gives the repulsion a chance to act on the path, making it smoother. Not all cases are handled quite so well; if the threshold is too small, a *max step size* step can jump over the $MINIMUM$ distance region and cause a collision.

3.3 Faster Path Planning

Because the local path planner is used to determine visibility between global subgoals, it is imperative for the local planner to be fast.

3.3.1 Standard Stopping Criteria

First, we implemented two standard stopping criteria for iterative processes. The simplest counts the number of steps taken and if greater than *max count*, stops.

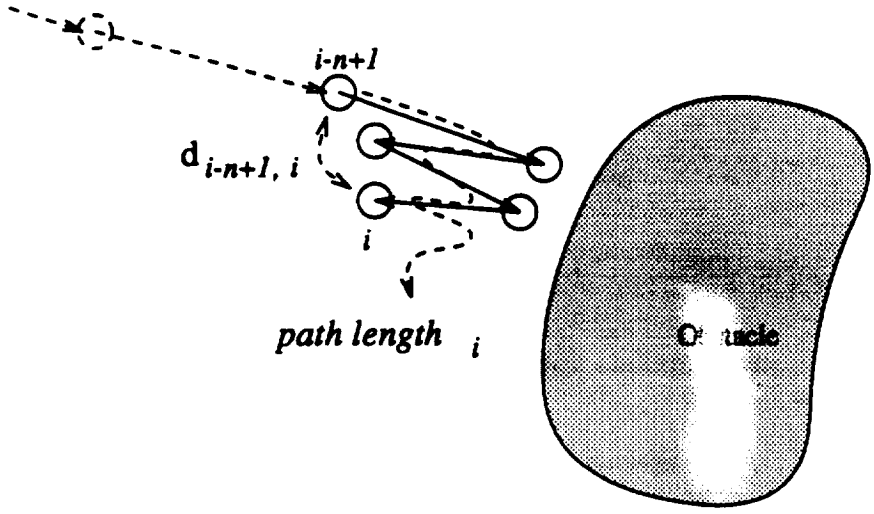


Figure 3.13: Detection of Oscillations (n = 5)

This is a brute force cure for endless loops. The second criteria tests for movement. If the step size $\|\mathbf{dq}\|$ drops below some minimum threshold, then the robot is not moving, and we stop. Since, through self-motion, redundant robots can have large $\|\mathbf{dq}\|$ steps without any movement at the gripper, we also test the change in gripper pose. If it becomes too small, we stop.

3.3.2 Oscillation Detection

Another condition which should cause the local planner to quit is a path with oscillations at the gripper pose. These paths are undesirable because they are hard for the robot's motor controllers to handle.

We define oscillations according to the path of the end effector (Figure 3.13). First, we choose a range of n steps to be tested. Then we call the *Oscillation Detection Algorithm* after each iteration of the local planner. If oscillations are detected then we report a local planner failure.

Oscillation Detection Algorithm:

1. Store the current end effector's pose as $pose_i$.
2. Calculate and store the distance from $pose_{i-1}$ to $pose_i$ as $d_{i-1,i}$.
3. Calculate $d_{i-n+1,i}$,the straight line distance from $pose_{i-n+1}$ to $pose_i$.
4. Calculate the path length traversed by the end effector in the last n steps,

$$path\ length_i = \sum_{j=1}^{n-1} d_{i-j,i-j+1} \tag{3.11}$$

5. If $\frac{d_{i-n+1,i}}{path\ length_i} < \frac{1}{n}$, then the end effector has not moved significantly within the last n steps; report oscillations.
6. Else, report no oscillations.

—end of algorithm—

3.3.3 Ignore Obstacles

The more obstacles there are in the environment, the slower the planner. Speed can be improved by ignoring some obstacles. Before calculating the repulsion between any two objects, and even before putting an object into a cluster, the distance between the objects is compared to a threshold, and if that distance is larger than the threshold, then the pair of objects is considered too far apart to significantly effect one another, and they are dropped from further consideration.

This simple method greatly improves the speed of the algorithm in crowded environments. Clusters, Section 3.2.3.1, would be too time consuming without ignoring distant obstacles.

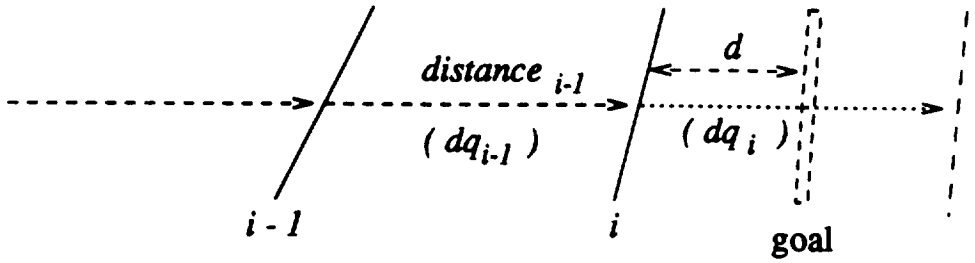


Figure 3.14: Reduce Step Size to Avoid Overshoot

3.4 Accurate Goal Pose Acquisition

3.4.1 Step Size Conditioning

Obtaining both positional and rotational accuracy with an iterative algorithm is a convergence problem. Consider Figure 3.14 which shows a strut payload moving towards its goal. If the step size, dq_i, is too large, the goal will be overshot (as indicated by the dotted line).

To remedy this, we scale down the step sizes. Within the local planner's iterative loop: after **dq** is calculated but before it is added to the current joint vector (between steps #1 and #2 in *Joint Range Freezing Algorithm*), we call the *Step Size Conditioning Algorithm.* But first, let us define a "distance measure" by adding the cartesian distances traveled by each end of the strut and dividing by two. This distance measure measures both cartesian distance and rotation, and thus, it is a measure of pose. Using this new measure, let us define $distance_{i-1}$, in Figure 3.14, to be the distance traveled by step $i-1$, and d to be the distance from step i to the goal.

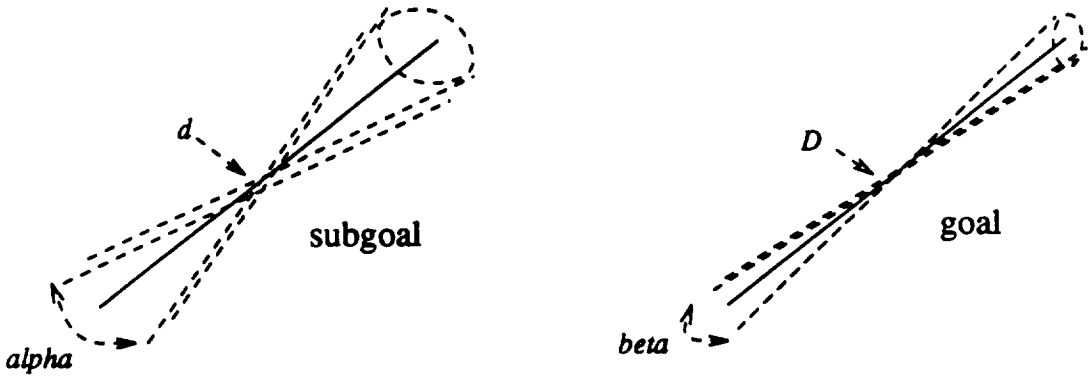


Figure 3.15: Range of Acceptable Locations of a Strut in the Gripper

Step Size Conditioning Algorithm:

1. Calculate d and $distance_{i-1}$.

2. If $d < distance_{i-1}$, then condition the i^{th} step according to the ratio, $d/distance_{i-1}$. Let

$$\mathbf{dq_i} = \mathbf{dq_{i-1}} \frac{d}{distance_{i-1}}. \tag{3.12}$$

—end of algorithm—

Note that we assume that linearly reducing the step size will linearly reduce the distance measure. Near the goal, this is a good assumption because we are dealing with very small joint increments.

With this improvement, we have been able to obtain fine accuracy (to within mm) at the expense of more iterations. While the convergence properties of our conditioning algorithm cannot be guaranteed, it does appear to be linear in nature, i.e., when the position is required to be ten times more accurate, the algorithm requires ten times as many iterations to achieve that accuracy.

3.4.2 Goals and Subgoals

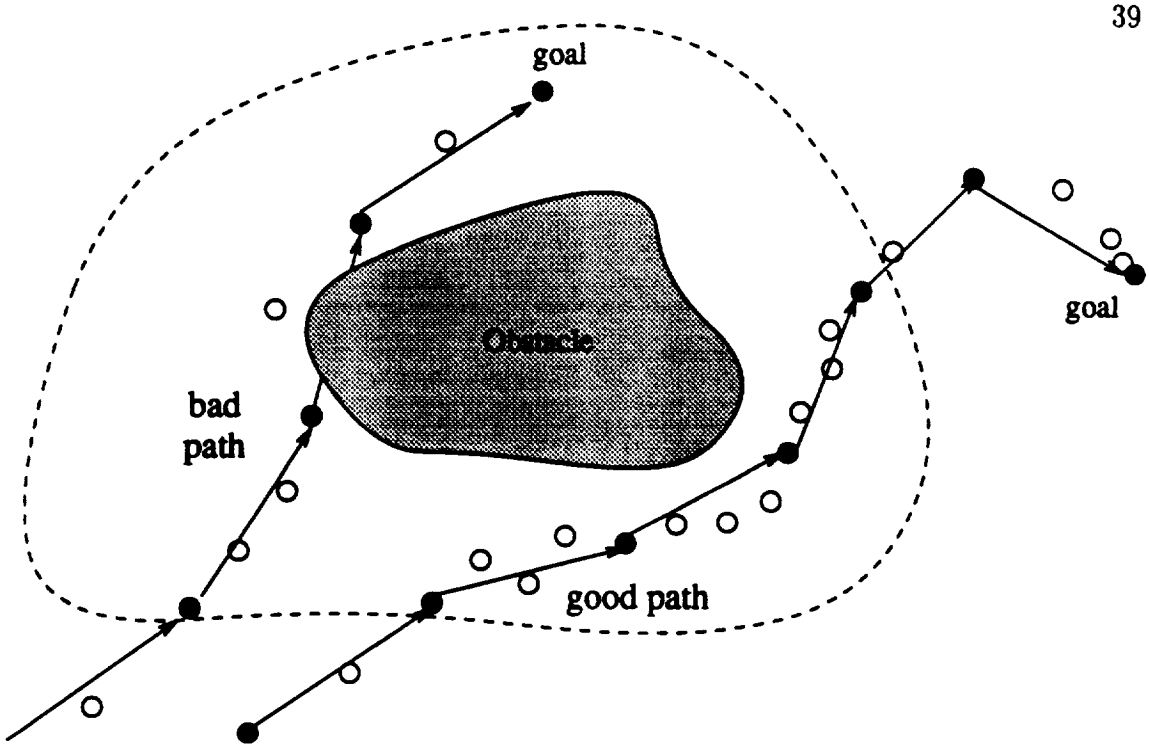


Figure 3.16: Choosing the Correct Report Size can be Critical

In the interest of speed, it would be advantageous to discriminate between our "real" goal, and other "subgoals". A subgoal such as those produced by the global planner from geometric information does not need to be attained with the same level of accuracy as the real goal, where we may need to do part insertions, visual inspections, or other detailed work. Thus, we adjust the stopping criteria for a successful plan accordingly (in Figure 3.15 $alpha > beta$ and $d > D$).

3.5 Smoothing

Our step size conditioning may leave the output path with very closely spaced steps and small oscillations. Trajectory generators and motor controllers do not respond well to these types of paths. We take a particularly simple one; we accumulate steps until the sum of their norms exceeds a threshold. Then we report that as one step. We also vary the size of the threshold in proportion to the minimum robot-obstacle distance to minimize the possibility of collision.

As can be seen in Figure 3.16, while this can smooth the path, it can also lead to undetected collisions. Therefore, the threshold must be chosen carefully (small).

One better method to smooth paths is called string-tightening. This method averages the changes between steps (see Weaver [4]).

CHAPTER 4
GLOBAL PATH PLANNER

4.1 Sending Different Options to the Local Planner

The local planner is given four chances to establish visibility between intermediate goals. We have added two control options for the local path planner. The first option sends two labels: one describes the goal as either the real goal, or as an intermediate goal, and the other describes the starting location as the original starting location, or as an intermediate one. We have already described how the local planner will change the success criterion according to the type of goal in Section 3.4.2. It will also only plan lift-off and approach offsets if the start and the goal, respectively, are real.

The second option controls the magnitude of the lift off and goal approach offsets.

4.2 More Subgoals

The global planner is only as good as the choice of intermediate goals, or subgoals. If there are too many subgoals, the planner will be slow, in fact, $O(m^2)$ where m is the number of subgoals. But if there are too few subgoals, then the planner may not find any solutions. Munger placed subgoals at the corners of tetrahedra, and we have added more subgoals near the edges of triangles (see Figure 4.1).

Through experience, however, we have found that these geometric subgoals have some undesirable properties. For every strut in the environment there is one subgoal: too many when building large truss structures. Another problem is that the subgoals are placed without regard to either the final goal or the starting position. Often these subgoals are invisible.

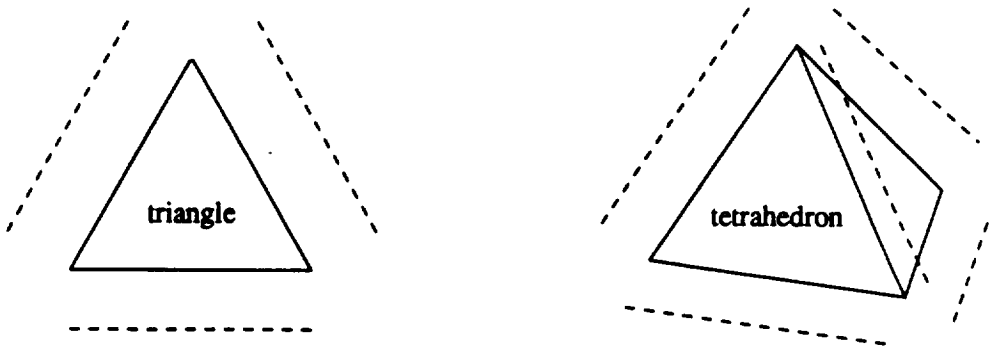


Figure 4.1: Two Geometric Objects and Their Subgoals

We have devised one solution for both problems. After every failure of the local path planner to establish visibility between nodes, we create two subgoals around the obstacle which was closest to the robot when the local algorithm failed. This obstacle probably caused the failure, so it needs to be avoided. The two subgoals are placed offset from the obstacle (in this case a strut, see Figure 4.2) along the normal to the plane defined by the obstacle's axis and the line connecting the obstacle to the center of the goal. See Chapter 8 for examples of this type of subgoal's success.

While this type of subgoal produces two subgoals per obstacle, it only does so for those obstacles from which we have encountered interference. The worst case is worse ($O(4m^2)$), but the average case will have fewer subgoals (m will be significantly reduced).

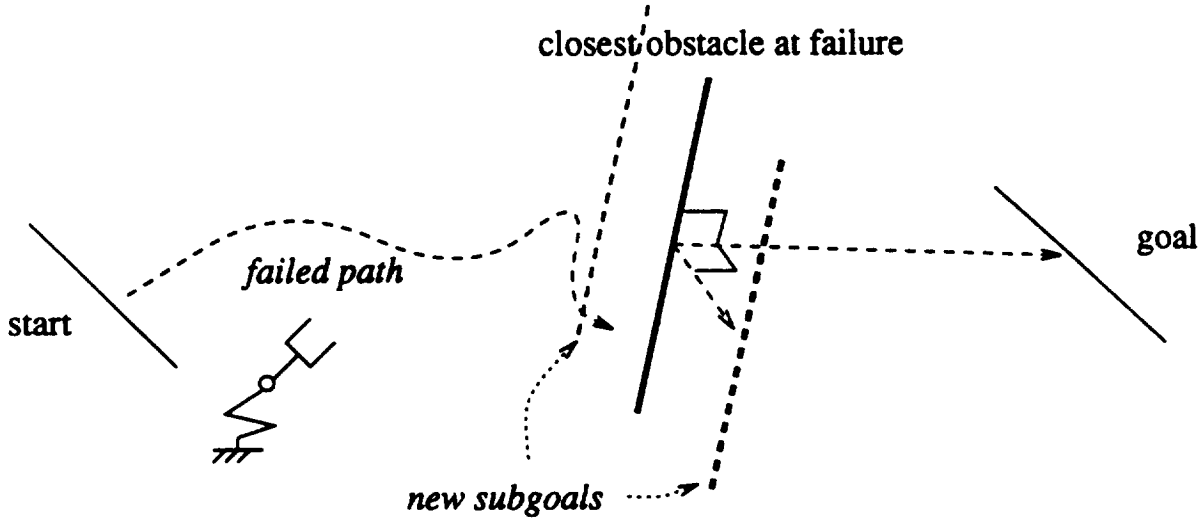


Figure 4.2: Subgoals Around Blocking Obstacles

CHAPTER 5
SOFTWARE IMPLEMENTATION

The algorithms described in the previous chapters have been implemented in C running under UNIX on CIRSSE's Sun 3/60, Sun 4/350, and SPARC workstations. The algorithm can run on its own or as a client within a distributed application in the CIRSSE Testbed Operating System (CTOS, see Chapter 6). In either case, the algorithm is the same except for a few modifications in the interfacing procedures. The algorithm is portable to other computer systems with the exception of the CTOS interface and the graphics interface. Since the basic structure of the software has not changed, this chapter repeats many of Munger's descriptions.[1]

5.1 Programming Concepts

We divide the code into 19 modules for organization, compiler friendliness, debugging, and reprogramming. Each module consists of a C code file (`filename.c`) and a header file (`filename.h`). This division is a way to hoard information "privately" within a module while sharing "public" information with other modules. The code files contain the private constants, type definitions, and variables, as well as the public and private procedures. The header files contain public constants, type definitions, variables, and procedure declarations. Within each header file is a complete description of its public procedures, sufficient to enable a user to use those procedures.

5.1.1 Module Hierarchy

If module A uses public information in module B, then module A is said to be "higher" in the program hierarchy. There are two degrees to which module A

[1]Sections marked with a * are either paraphrased or taken directly from Munger [1], pp. 44–66.

can use module B. If A's code file #includes B's header file, then A has access to all of B's public features. For example, A can call public procedures defined in B. However, if a procedure in A returns a value whose type is defined in B, then A's header file must also #include B's header file so that B's type definitions are available to modules which wish to use A. This causes a problem, however, if a module which includes A's header file also includes B's header file: B's header file will be included twice. To avoid this, we surround every header file's code with a unique labeling structure that ensures that every header file is included only once.

```
#ifndef UNIQUE_MODULE_LABEL
#define UNIQUE_MODULE_LABEL

   module's constants, typedefs, declarations ....

#endif
```

Figure 5.1 shows the path planner's complete module hierarchy. Solid lines represent normal links and dashed lines represent an inclusion of header file in header file. The arrows point from the included file to the including file.

5.1.2 Variables

In order to keep modules as separate as possible, program-wide global variables are banned. Although global variables are used extensively within modules, they are not shared with other modules (i.e. they are `static`). Thus data transfer between the modules is carried out solely in the procedure arguments. There are, regrettably, two exceptions to this rule: `gsmTid` and `ppTid` are task identification numbers used by CTOS to facilitate message passing and are needed in many path planner modules (wherever CTOS is used) so these were given global scope.

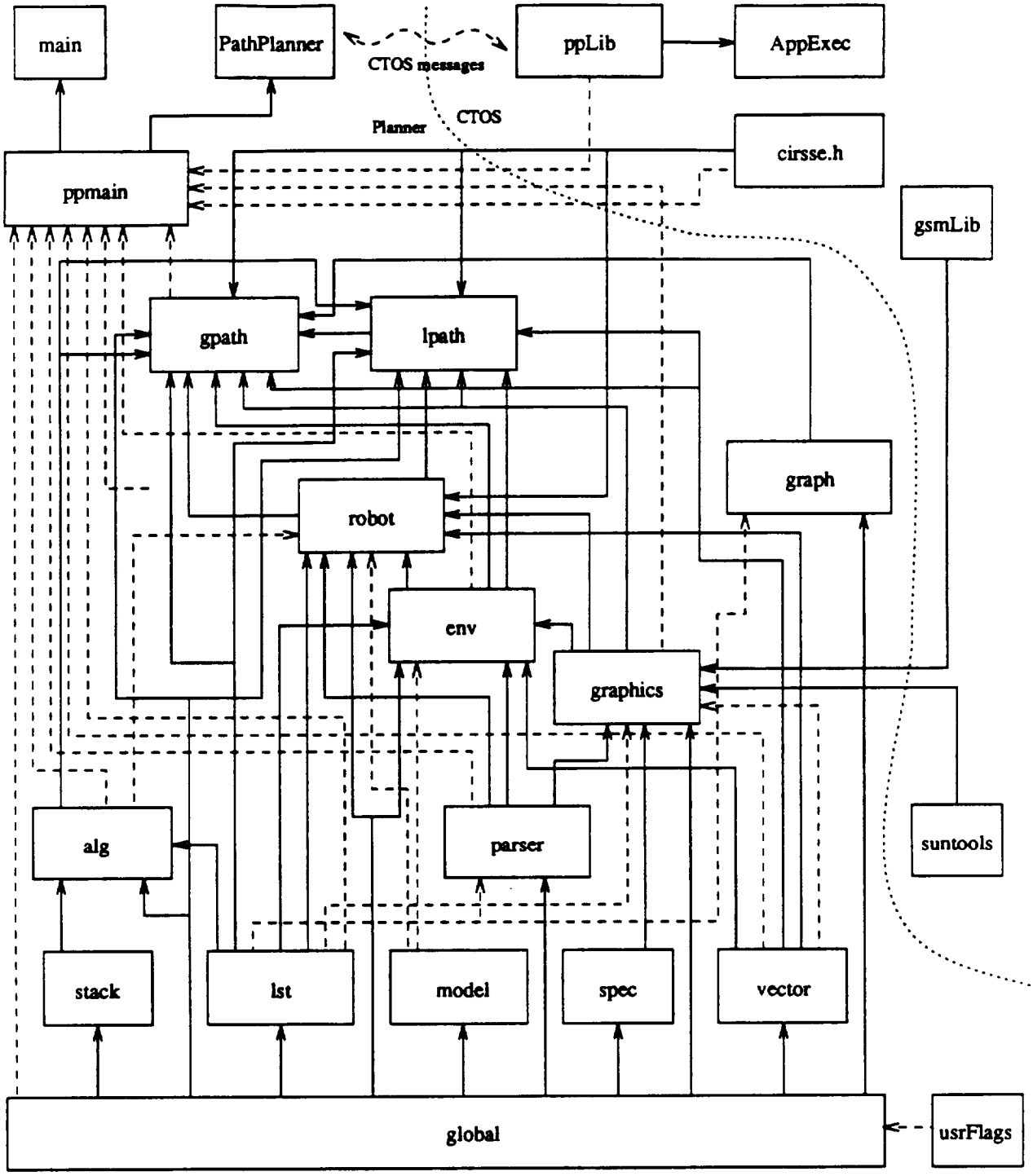


Figure 5.1: Module Hierarchy

5.1.3 Data types*

In order to keep strict control over public data types, user modules are allowed only limited access to other modules' public data types. Users are not permitted to directly access data items within another module's public data type. Controlled access, however, is provided in the form of public procedures in the supplier's module. For example, a typical data type may be defined in the header file:

```
typedef struct complex
{
  double real;
  double imag;
} COMPLEX
```

This could represent a data type for a module which implements operations on complex numbers. As a convention, all instances of data types printed in all uppercase letters are allocated in heap memory. These data types all have procedures to create new instances and procedures to kill old instances:

```
COMPLEX *New_Complex ()
void Kill_Complex (COMPLEX *x)
```

Other procedures are provided as needed, such as (for our example) Set, Add, Sub, Display, etc.. Since other modules are not allowed to access the structure directly, statements like the following, while perfectly legal in C, are not allowed:

```
c->imag = 2.0;
x = c->real;
```

A procedure that adds two complex numbers $4 + 5i$ and $2 - 8i$ would look something like:

```
#include ''complex.h''

void Example ()
{
  COMPLEX *a, *b, *c;

  a = New_complex ();
  b = New_complex ();
  c = New_complex ();
  Set_Complex (a, 4.0, 5.0);      /* set a to 4+5i */
  Set_Complex (b, 2.0, -8.0);     /* set b to 2-8i */
  Add_Complex (a, b, c);          /* add a and b and put the result in c */
  Display_Complex (c);            /* print the result */
  Kill_Complex (a);               /* remove instances from heap memory */
  Kill_Complex (b);
  Kill_Complex (c);
}
```

It may seem that this requires alot of extra effort. However, the idiom is easily mastered. A little extra sweat now is rewarded with greater speed, and later, when a data type needs to be changed, only the owning module needs to be updated.

5.1.4 CIRSSE's Make

All the program's modules can be compiled using a CIRSSE version of the UNIX utility **make** called **cmkmf**. By having the dependency tree information and checking the time/date stamp on every file, **cmkmf** can decide which files need to be recompiled and linked. The information on dependencies resides in a file called **Imakefile** which reflects the hierarchy shown in Figure 5.1. See CIRSSE Report #128 and Tech Memo #16 for details on the use of **cmkmf** (see Appendix A for an example Imakefile).

5.1.5 Compiler Flags

Within "usrFlags.h" are four **#define**'d flags which allow us to use the same code for both UNIX and CTOS demonstrations. By bracketing system dependent

code in `#ifdef FLAG ... #endif` pairs, code can be hidden and unhidden by undefining or defining, respectively, the `FLAG`.

The flags and their effect, if defined:

CTOS_ACTIVE Indicates that we are running as part of a CTOS application. Activates the proper display devices.

PREVIEWER Only defined if CTOS_ACTIVE is defined. Indicates that we are running as part of a larger demonstration which handles graphics on its own. We are only responsible for the bare minimum: the output path and any error messages.

DIAGNOSTICS Displays a full array of state information with every step.

MAN_IN_LOOP If not in PREVIEWER mode, query the user for acceptability of paths found; if not acceptable, plan another path.

5.2 The Modules

Here, we describe each module of the program starting from the bottom to the top of the hierarchy. The same descriptions can be found in the modules' header files.

5.2.1 The "usrFlags" module

This header file is included into the global module's header file (so it is included in all the modules). It contains four flags which specify compile time options (see Section 5.1.5).

5.2.2 The "global" module

This module is included by every other module of the program. It includes the two standard header files `stdio.h` and `math.h`, defines the boolean data type

and provides standard procedures for displaying error conditions on the screen. It also contains a customized version of the **atan2** function.

5.2.3 The "spec" module*

This module provides information about the machine the program is running on, including the availability of a graphics screen and whether or not the screen has color capability.

5.2.4 The "lst" module*

The list module provides a way of putting any type of data into a sequential list. A list consists of a main list data structure (LIST) and a number of list elements (LIST_EL) representing the data elements. These list elements are dynamically allocated, so no information about the list's length is needed at compile time. This is the main advantage of using this module over using a simple array.

The LIST_EL data type contains a pointer "next" pointing to the next LIST_EL in the list and a pointer "data" that points to the listed data element, thus a simple forward chained list is implemented. However, this chaining mechanism is totally hidden in the module, so the fact that the user's data types must be stored in a list has no impact on their internal structure.

Lists are built by adding elements to either the beginning or the end of the list. The most common way of reading a list is by sequential access using the procedures "Get_First" and "Get_Next".

This module also provides random access, but since this procedure must go through the chain of list elements, the access is slow for long lists. To improve random access performance, the module allows the creation of an index array. This regular C array (in a contiguous block of memory) contains pointers to the data elements, thus array-like list access becomes possible. However it must be noted

that every change in the list caused by adding or deleting an element automatically destroys the index, so indexed list access is only possible if the list is not changed after the index is created.

5.2.5 The "stack" module*

The stack module provides a way to organize any type of data in a stack (LIFO - buffer). Every data entry is represented by an instance of STACK_EL. This data structure holds a pointer to the user's data and a pointer to then next instance. Access to the stack is accomplished by the procedures "Push" and "Pop". The procedure "Read_Top" reads the latest entry on the stack without removing it.

5.2.6 The "vector" module*

The vector module provides three data types:

- column vector with 3 elements
- 3x3 matrix
- 4x4 homogeneous matrix with 4th row omitted (assumed [0 0 0 1])

The elements of a vector are doubles, the columns of the 3x3 matrix are vectors and the homogeneous matrix is comprised of a matrix for the first three columns and a vector for the 4th column. Unfortunately, all three data types are used as normal variable declarations; no instances of these types are allocated in heap memory. This causes much unnecessary passing of structures.

The module also provides a set of useful operations on vectors and matrices. The distance computations for line segments and planar segments are implemented here (see Appendix E).

5.2.7 The "alg" module*

This module provides a set of operations on m by n matrices (linear algebra). The basic data structure is a variable (VAR) which may be a matrix, a vector or a scalar. A variable automatically adapts its size to a matrix that is assigned to it, so the user doesn't need to know the dimensions of the result of an operation ahead of time.

There are some element oriented functions that require the specification of row and column values (typically parameters r and c). As a convention, the first row or column is number 0, so the element in the top left corner has row and column indices (0, 0). Names of functions returning a value of type VAR begin with a capital V (example: Vadd).

All functions returning a BOOLEAN return TRUE if they complete successfully and FALSE if a problem is encountered.

All assignments must be made using the procedure 'Put (expr, v)' which puts the result of expression 'expr' into variable 'v'. For example, the assignment $X = A + B * C$ is programmed as follows:

```
Put (Vadd (A, Vmult (B, C)), X);      /* correct. */
```

Note that direct pointer assignment will not work. The following statement is wrong:

```
X = Vadd (A, Vmult (B, C));           /* wrong ! */
```

The following example program will assign values to A, B and C, will evaluate the expression $A + BC$, assign the result to X and print it on the screen.

$$A = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \qquad B = \begin{bmatrix} 20 \\ 13 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \tag{5.1}$$

```
#include "alg.h"
main ()
```

```
{
  VAR *A, *B, *C, *X;

  Init_Var ();                          initialize module
  A = New_Var ();                       make the variables
  B = New_Var ();
  C = New_Var ();
  D = New_Var ();
  Put (Vuser (2, 1, 1.0,                makes A the 2x1 variable [1.0]
                    3.0), A);                                    [3.0]
  Put (Vuser (2, 2, 2.0, 0.0,           makes B the 2x2 matrix [2.0  0.0]
                    1.0, 3.0), B);                             [1.0  3.0]
  Put (Vuser (2, 1, 0.0,                makes C the 2x1 variable [ 0.0]
                   -1.0), C);                                    [-1.0]
  Put (Vadd (A, Vmult (B, C)), X);      evaluate A+B*C, put result in X
  Print_Var (X);                        print X on the screen
  Kill_Var (A);                         free space
  Kill_Var (B);
  Kill_Var (C);
  Kill_Var (X);
  Exit_Var ();                          exit module
}
```

5.2.8 The "graph" module*

The graph module provides a way to organize any kind of data in a directed or undirected graph. The data structure consists of a main structure (GRAPH) and the two structural elements G_NODE for the nodes (vertices) and G_EDGE for the edges of the graph.

The GRAPH data structure contains a list of the graph's nodes. Every node in turn has a list of adjacent edges. If the graph is directed, then the node's list contains only adjacent edges that are pointing away from that node. Every edge has two pointers to the two nodes it is connected to. These two pointers are called 'node1' and 'node2'. If the graph is directed, the edges are always pointing from 'node1' to 'node2'. Both the edges and the nodes have a pointer to a data structure in the user's module. In an example of a graph representing cities and connecting

roads the nodes would contain a pointer to CITY and the edges a pointer to ROAD. Only the user's data structures are used for communication between the modules so the internal structures G_NODE and G_EDGE are invisible for the user.

An edge in a graph always has an associated weight that represents the cost of traversing the edge. In directed graphs, the edges cannot be traversed in the wrong direction, it is however possible to define two edges between the same two nodes having opposite directions and different weight values. This weight value is not passed to the edge at the time the graph is being established, but the user must provide a weight function that returns the weight of any edge to the graph module. This way the graph module can query the weights whenever they are needed and no unnecessary weights are computed. If the computation of the weights is complicated, then this feature can save a considerable amount of computing time. Once the weight is computed, it is stored in the edge structure, so the computation is done only once per edge. This implies that an edge's weight cannot change during the lifetime of the graph.

The module offers procedures for building, changing and deleting graphs, and the graph search algorithm A*. There is also a set of utility functions for navigation in the graph structure, but these functions are not accessible from outside the graph module. They are intended as tools for development of new graph algorithms and are documented in graph.c.

5.2.9 The "parser" module*

The parser module provides a convenient way of reading information from an input text file. The text in the file must conform to the following syntax:

```
S          = {expression}
expression = keyword [par_list]
par_list   = '('{parameter ','} parameter ')'
keyword    = string
```

```
parameter  = string
string     = {char} char
char       = 'A'..'Z' | 'a'..'z' | '0'..'9' |
     '+' | '-' | '.' | '&' | '_'
```

In this syntax description, S is the start symbol, lowercase words are nonterminal symbols and characters in single quotes are terminal symbols. An expression in braces can be repeated any number of times (including zero times) and an expression in square brackets [ ] is optional. If there are a number of expressions separated by bars — then either expression is legal at this point.

Examples for legal commands are:

```
ADD (5, 6, 7, -11.5)

Exit_Program

save&quit (foo.c)
```

The parser module will first read a user specified source file, parse it according to above syntax, store the data in a list of expressions and return this list to the user. The order in the list corresponds to the order in which the expressions are encountered in the source file. If there are syntax errors, they will be printed on the screen. The module offers a variety of interface procedures that enables the user to read the data in a convenient manner. Expressions can be read from the list sequentially as it is normally done with lists. Lists can also be scanned for the next occurrence of an expression with a particular keyword. An expression is a data type (EXP) that also has some procedures associated to it. The user can read an expression's keyword string, the number of parameters in the expression and a particular parameter string given by its number in the parameter list. Finally there are utility procedures that convert a parameter string to a real or an integer number. This is necessary since all parameters are handled as arbitrary strings.

5.2.10 The "model" module

This module provides a geometric primitive which is useful for the modeling of solids. The primitive is described by two or three points p1, p2, and p3, a radius r, a direction vector dir, a task identification id, and a "type". If the type is STRUT_TYPE then the object is obtained by moving a sphere of radius r on a straight line from point p1 to point p2 (a cylinder with spherical caps). If the type is TRIAN_TYPE then the object is a triangular planar segment whose boundaries are defined by p1, p2, and p3. This plane segment has a volume equal to that swept out by a sphere moving through every point in the plane segment. The "dir" element describes a subgoal's approach vector. "id" is used for graphics object identification when running with CTOS applications.

Procedures are provided to read and change the model's parameters and to compute the minimum distance between two swept sphere models using procedures in the vector module.

5.2.11 The "graphics" module

This module currently serves two purposes. It was originally programmed using SUNcore graphics for use on SUN machines. This capability has been maintained, even though it is outdated and being phased out. More recently, an X Windows graphics viewer has been written by Nicewarner [30] and is incorporated in our graphics module.

"SUNcore allows line and character drawing in three dimensional space. Colors are used if the monitor allows and if black and white mode is not explicitly selected. After initialization, a three dimensional coordinate system is displayed. There are procedures to create segments — an entity that holds a number of primitives — and others to create lines and characters at arbitrary locations in three dimensional space."

"Other procedures allow the user to insert primitives into a segment and delete them from segments. Yet another procedure allows the user to rotate the current picture around the vertical and the horizontal axis of the screen by moving the mouse horizontally or vertically, respectively. This mode ends in the current orientation when the middle button is pressed. The reason for using segments is the segment concept of SUNcore. SUNcore segments do not provide any way of deleting single primitives stored in them, so the whole segment must be deleted and reconstructed in order to delete one primitive. This module automatically deletes and reconstructs the SUNcore segments as needed. Unfortunately, this process is visible on the screen, especially on slow machines. The segment concept allows the user to split the picture into parts, avoiding the reconstruction of the whole picture when a single primitive is deleted."*

If X Graphics are selected, then we rely on the graphical viewer written by Nicewarner [30]. If we are running on UNIX as a stand-alone program, then the output is in the form of a file readable by the viewer interface. If we are running as part of a CTOS application, then we send the Geometric State Manager messages directly. We can create, delete, and move struts and links. These struts and links must be defined in .cgm files.

5.2.12 The "env" module*

"env" stands for environment, so this module holds all data about items that belong neither to the robot nor to its payload. At initialization, the module reads the locations of the struts and planes from the parsed source file and informs the graphics module.

Procedures are provided to get models of struts, planes, and intermediate steps currently in the environment and to add and remove them.

Procedures are also provided for extracting tetrahedra and triangles and placing intermediate steps around them. Whenever a strut is added or removed, all intermediate steps are deleted and then re-established based on the new environment.

5.2.13 The "robot" module*

This module models a single chain robot with an arbitrary number of links. The description of the robot's kinematics, model geometry, joint ranges and so forth are stored in a file `robot.def`, so that the files `robot.c` and `robot.h` are applicable to any single chain robot without change. The robot's kinematics are described using modified Denavit Hartenberg parameters (see Craig [27]).

This module maintains a set of transformation matrices that represent the transformation from each link to world coordinates. Derived from the modified Denavit Hartenberg parameters [27] and the current joint vector, they are updated each time the robot changes its joint vector. The module also maintains a swept sphere model of each link. These models are not automatically updated when the joint vector changes, since this process is time consuming and not always necessary.

The module provides three procedures to alter the robot's state: the robot's joint vector can be set, a part can be added to, or removed from, the gripper.

Various readout procedures supply information about the current position of the link models, the type of a particular link (revolute or prismatic), the origin and the axis of the joints, the current value and range of each joint, which joints are out of range, and whether the robot is carrying a payload or not.

5.2.14 The "lpath" module

The path planning algorithm using potential fields is implemented in this module. The user must specify which arm to plan for, the rotation-type, the departure

and approach offsets, the initial joint vector, and the desired goal pose. The module will return a list of joint vectors that describe a path leading there. If this is not possible, it returns a FALSE.

5.2.15 The "gpath" module

The global path planning algorithm using graph search is implemented in this module. It establishes a list of joint vectors describing a path that leads from the current joint vector to a goal position defined in cartesian space. It may call the local path planner several times on the whole task or on part of it. It may also query the local planner for information about failures in order to set up subgoals. The global planner may be called on repeatedly, and it will return a different path each time until it can find no more new paths, then it will return the first path found.

5.2.16 The "ppmain" module

This module provides the interface between the planning algorithm and higher level coordinators. The procedures contained within this module are invoked by either the main module or the PathPlanner module. There are procedures to initialize and shutdown the path planner. There are procedures to read instructions from a file and to output paths to a file. A procedure which parses and executes the input file is also provided.

5.2.17 The "main" module

This module is used exclusively when running the planner in its UNIX standalone mode. `CTOS_ACTIVE` must not be defined in "usrFlags.h". This module parses and executes an input file.

The available input file commands are listed in Section 5.3, below.

5.2.18 The "PathPlanner" module

This module is the event handler for CTOS messages coming from the higher level coordinator, currently the dispatcher petri-net. It handles all the standard CTOS messages and also plans paths, re-plans paths, does inverse kinematics, and can run input files. The messages' definitions are in `ppLib.h` and their calling procedures are in `ppLib.c`. This procedure is the communications highway between the path planner and the application coordinator.

Refer to Chapter 6 for a discussion of CTOS and message passing.

5.3 The Input File

The input file is an auxiliary input source for testing the program. It can also be used to set up the environment when embedded in a larger program.

There are two types of commands found in the input file: static commands and sequential commands. The static commands are read at initialization time and their order does not matter. The planner executes the sequential commands, in order, of course.

We describe a strut in space in two ways. First, we can describe it by its endpoints. This requires six parameters, three for each cartesian space endpoint. For the remainder of this section, the parameters (`x1, y1, z1, x2, y2, z2`) will denote the two cartesian endpoint vectors of a strut with respect to the world coordinate system.

A strut within a tetrahedral structure has a simpler description. By defining the position and orientation of a tetrahedral structure, all struts within that structure can be described by a tetrahedron number and a strut number within that tetrahedron according to the numbering system shown in Figure 5.2. **p** is the position of the structure, and **r1** and **r2** are the orthogonal orientation vectors of

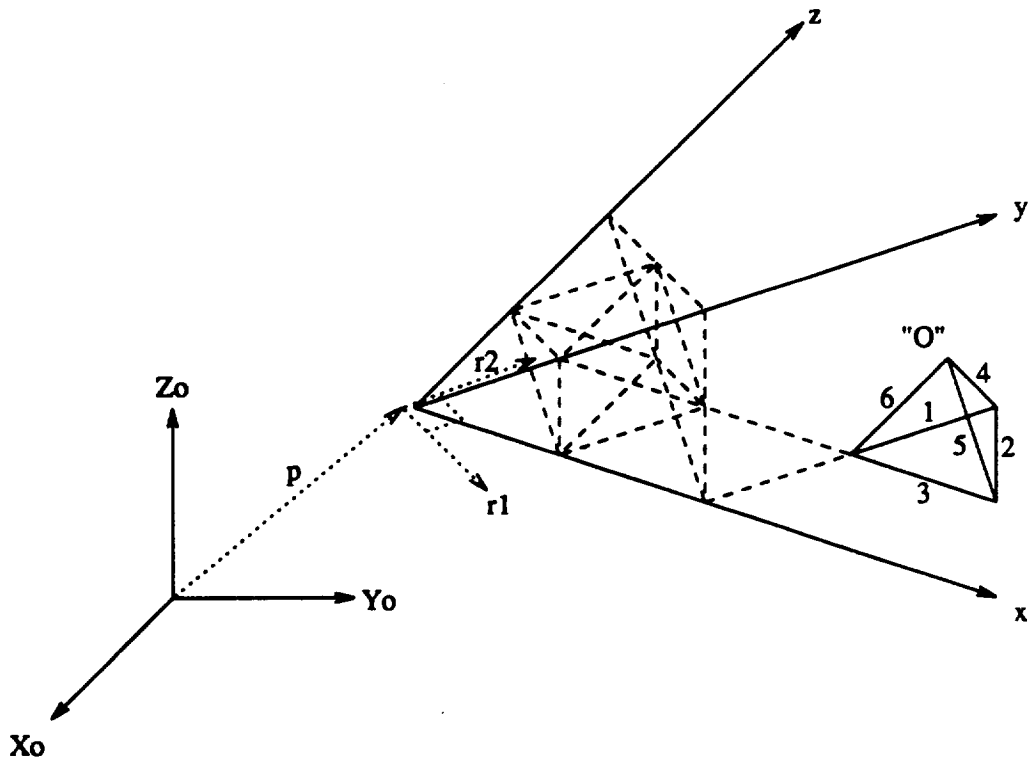


Figure 5.2: Tetrahedral Structure Numbering Conventions

the structure with respect to the world coordinate frame (**Xo, Yo, Zo**). The coordinate system (**x, y, z**) is used to specify a tetrahedron in the structure. The length of the system's unit vectors $\mathbf{e_x}, \mathbf{e_y}$ and $\mathbf{e_z}$ is equal to the length of a tetrahedron's edge (the sum of a strut length and two times the node connector length). Now we can describe any tetrahedron by an ordered triple of integers. For instance, (2, 1, 0) describes the "O" tetrahedron in Figure 5.2, ($\mathbf{O = p + 2e_x + 1e_y + 0e_z}$). The "O" tetrahedron also shows the numbering system for a particular strut within a tetrahedron. Thus we can describe struts with four integers, once the structure's position, orientation, and unit length have been specified. For the remainder of this section, the ordered quadruple (X, Y, Z, N) will denote such a strut.

The following are the static commands:

- **STRUTLENGTH** (l)

  l specifies the length of all struts in the environment.

- **NODELENGTH** (l)

  l specifies the length of all nodes. Together with STRUTLENGTH defines the unit length of the tetrahedron structure.

- **STRUCTURE_LOC (px, py, pz, r1x, r1y, r1z, r2x, r2y,r2z)**

  Defines the location of the tetrahedral structure with respect to world coordinates. The parameters are vectors **p, r1, and r2** from Figure 5.2.

- **SUBGOAL (x1, y1, z1, x2, y2, z2)**

  Defines an intermediate goal which the global planner puts in its Graph search. The endpoints will be automatically adapted to the strut's length.

- **STRUT (x1, y1, z1, x2, y2, z2)**
  STRUT (X, Y, Z, N)

A strut is added to the environment at the given location. The strut will automatically be adapted to the strut length such that the center stays fixed.

- **TETRA (X, Y, Z)**

  Six struts of a tetrahedron are added to the environment.

- **PLANE (x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4)**

  A plane object is created with the four vertices given. See AppendixE.

- **ROBOT (px, py, pz, r1x, r1y, r1z, r2x,r2y, r2z)**

  Orients the robot's zero frame with respect to the world coordinate system. Vector **p** denotes the origin of the robot's zero frame, **r1** denotes the orientation direction of the robot's x axis, and **r2** denotes the orientation direction of the robot's y axis.

- **X_GRAPHICS, SUN_GRAPHICS**

  Selects either X Window or SUNcore graphics to be displayed.

- **ZOOM (z)**

  If SUN_GRAPHICS is active, then z is the magnification of the display.

- **B&W**

  If SUN_GRAPHICS is active, then the graphics are displayed in black and white.

- **DIAGNOSTICS**

  Activates debugging print statements for display during execution.

- **STEP**

  Pauses execution of planner after each step.

This second list shows the sequential commands which must be between the START and the QUIT commands.

- **START**

  Denotes the beginning of the command sequence.

- **MOVE (x1, y1, z1, x2, y2, z2, dx, dy, dz)**
  MOVE (X, Y, Z, N, dx, dy, dz)

  Plans a path that leads the payload strut (real or imaginary) to the indicated goal. d is the approach direction, and -d will be the next MOVE's departure direction. The very first MOVE has no departure direction.

- **GRASP (x1, y1, z1, x2, y2, z2)**
  GRASP (X, Y, Z, N,)

  The strut closest to the position specified is removed from the environment and put in the robot's gripper. The environment recomputes intermediate steps around tetrahedra and triangles.

- **JOINTS (theta1, theta2, ... , thetaDOF)**

  Sets the robot's joints to the joint vector specified. Units are in degrees for revolute joints and meters for prismatic joints.

- **UNGRASP (x1, y1, z1, x2, y2, z2)**
  UNGRASP (X, Y, Z, N)

  The payload is released and added to the environment at the specified position. The environment recomputes intermediate steps.

- **UNGRASP**

  Unlike the other two UNGRASP commands, the payload is released at the exact location the robot has brought it to.

- **ADD_STRUT (x1, y1, z1, x2, y2, z2)**
  ADD_STRUT (X, Y, Z, N)

  A strut is added to the environment at the indicated position.

- **REMOVE_STRUT (x1, y1, z1, x2, y2, z2)**
  REMOVE_STRUT (X, Y, Z, N)

  The strut closest to the indicated position is removed from the environment.

- **VIEW**

  If SUN_GRAPHICS is defined, then execution of the command sequence stops and the user can use the mouse to change the orientation of the display. Execution resumes when the user presses the middle mouse button.

- **LEFT_ARM, RIGHT_ARM**

  Selects which robot arm will be effected by all subsequent commands.

- **QUIT**

  Denotes the end of the command sequence.

Appendix B lists a typical input file. The input commands can be in either large or small case.

CHAPTER 6
CIRSSE TESTBED

6.1 Physical Plant

At the center of CIRSSE's testbed are two, nine DOF robotic manipulators, each consists of a PUMA, six DOF articulated arm, mounted on a three DOF cart. The two carts are mounted on a single linear track (see Figure 6.1). The manipulators are controlled by a host of single board computers mounted on a VME cage running VXWORKS, a real-time operating system. The VME cage is then connected by ethernet to several UNIX based Sun 3's and Sun Sparcstations, a Datacube vision system, the PUMA arm controllers, and the Aronson cart controllers. The manipulators are also aided by several sensors: a pair of stereoscopic cameras and a laser are mounted in the ceiling while the manipulator's wrist carries a force/torque sensor and another camera. A good introduction to the robotic testbed is Nicewarner's Tech Memo #22.

6.2 CIRSSE Planner Requirements

The current goal of the CIRSSE system's demonstration is to autonomously build the triangular base of a tetrahedron. This requires the planner to be able to plan paths from the rack which holds the struts to the table where the triangle will be assembled. The paths must avoid three obstacles: the table, the strut rack, and the struts in the strut rack.

In the future, a planner will be needed to coordinate the two arms so they can jointly build a tetrahedron. This is beyond the current capability of our planner.

6.3 Software Architecture

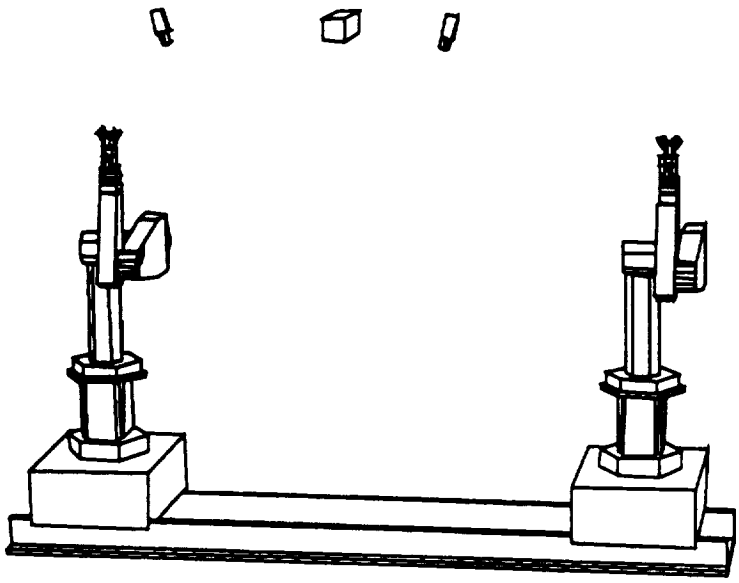


Figure 6.1: CIRSSE Testbed Robots

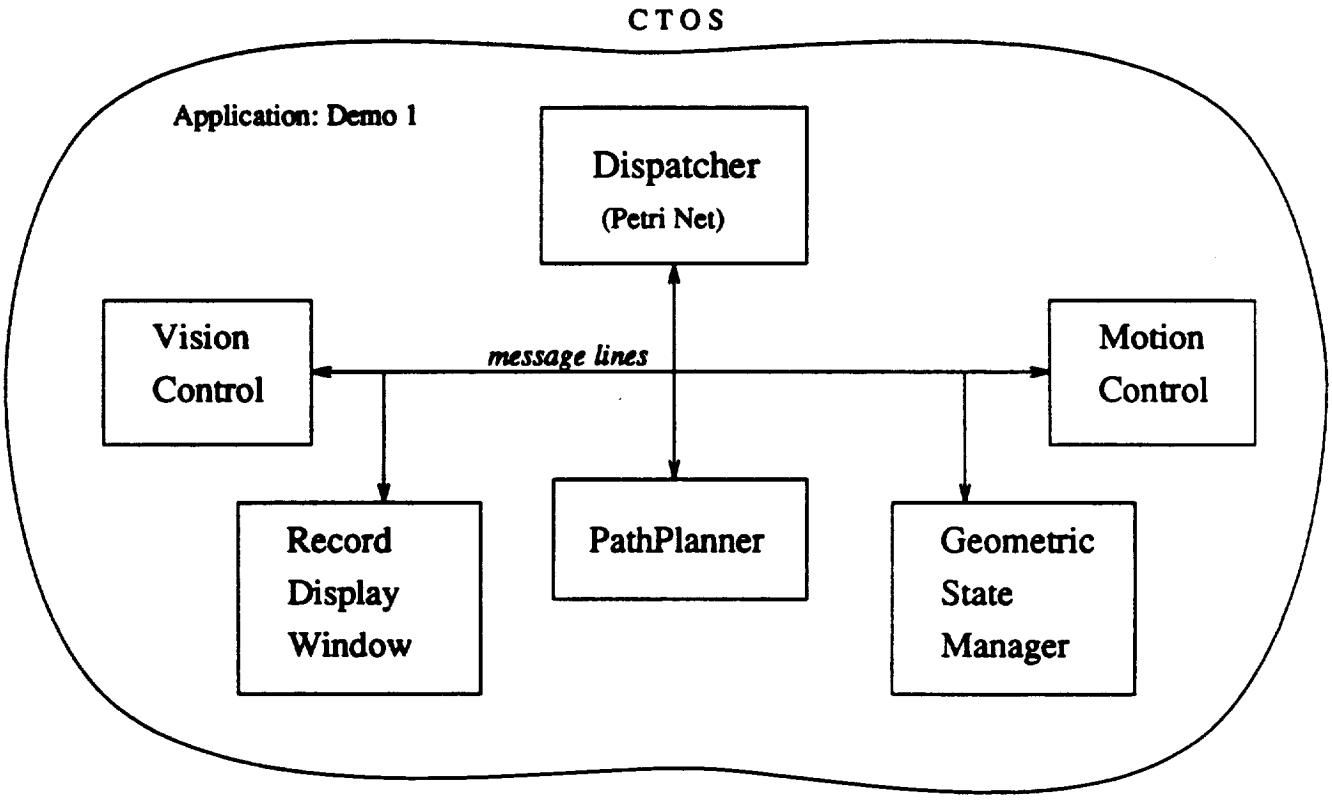


Figure 6.2: Demo 1 Application

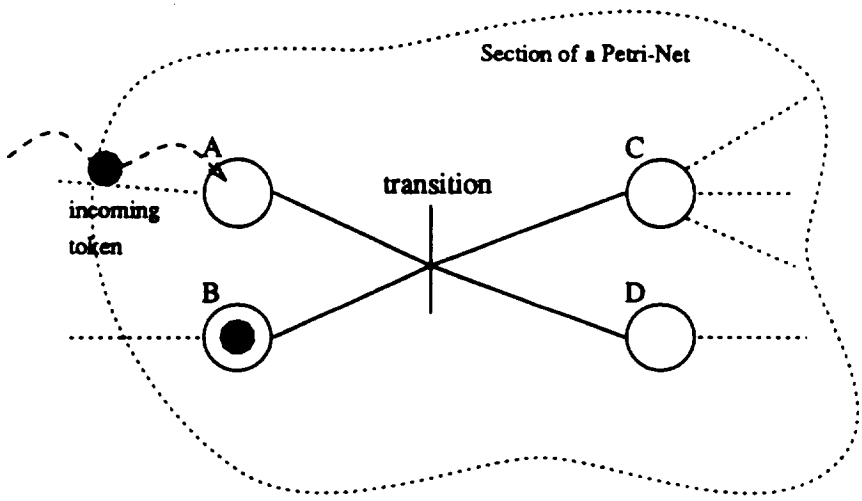


Figure 6.3: Nodes and Transitions

At CIRSSE, the pathplanner is just one of many "tasks" working together in an "application". The pathplanner algorithms were described in Chapter 5. This section will describe the planner's implementation within a CTOS application. CTOS (CIRSSE Testbed Operating System) was developed to handle multi-task applications. For a full description of CTOS, see Tech Memo #5, Tech Report #97, and Tech Report #128.

Each task is a separate program with its own string of execution. Tasks can run on different CPU's, or even, different computer systems. A group of tasks working on a job, called an application, communicate their work via "messages". CTOS messages contain two important pieces of data: a message identification number and a data block. CTOS makes it possible for complex jobs to be broken up into manageable pieces, which can be programmed and debugged separately by separate programmers.

Figure 6.2 shows how the path planner fits in with the other tasks at CIRSSE.

The dispatcher is implemented in the form of a petri-net, a token-based flow-chart which is designed to organize command flow. There are two basic units to a petri-net: nodes and transitions. Nodes hold data, while transitions are programs that use the data from the node. Referring to Figure 6.3, if the two nodes A and B are occupied by a token then the transition will fire and a token will be put in nodes C and D. Firing a transition is equivalent to executing a program, the output tokens are the products of the program.

The petri-net is a large, CTOS application executive. The it is like a director: when the script says that the robot needs to move, the dispatcher asks the planner for a path. After the planner returns a path, the dispatcher notifies the motion controller, and finally, the motion controller moves the robot.

All commands are all passed by CTOS messages. Therefore, each task must have access to the other tasks' message commands. By convention, the message identification numbers are defined in a `Lib.h` file, while the messages themselves are sent by a procedure in the corresponding `Lib.c` file. For example, when the dispatcher wants a path planned, it calls `ppPlanPath()` from the `ppLib.c` library. Then `ppPlanPath()` will send the CTOS message `MSG_PP_PLANPATH`, which is defined in `ppLib.h`, and other necessary data to `PathPlanner.c` by invoking the CTOS command `msgBuildSend()`. The path planner will return the path by invoking the command `msgReply()`, see details of these CTOS commands in Tech Report #128.

The path planner's defined messages, their calling procedures, and actions are:

`MSG_PP_INITIALIZE--ppInitialize()` initializes the planner; must be called before the planner is used for the first time.

`MSG_PP_SHUTDOWN--ppShutdown()` frees the data in lists, var's, and graphics.

`MSG_PP_STARTPATH--ppStartPath()` sets the planner's internal robot's joint vector.

`MSG_PP_PLANPATH--ppPlanPath()` executes path planning algorithm; returns a list

of joint angle knot points in a file.

`MSG_PP_REPLANPATH--ppReplanPath()` re-executes path planning algorithm with the same data as in the most recent ppPlanPath(); returns the next A* algorithm path.

`MSG_PP_ALTPLANPATH--ppAltPlanPath()` calls an alternate path planning strategy (see Lefebvre [29]).

`MSG_PP_INVKIN--ppPotInvKin` solves inverse kinematics problems by using the path planner; returns the goal's joint vector.

`MSG_PP_DEMO_EXEC--none` This message is sent by a stand-alone test program named AppExec; it is used to execute Munger's simulation input files.

`PathPlanner.c` is commonly called an event handler. When a message is sent to the path planner, CTOS calls a function `PathPlanner()` and passes it the message's identification number and a pointer to the message's data block. `PathPlanner()` then decodes the message, using a case statement, and calls the proper functions in ppmain.c (see Figure 5.1 and Section 5.2.16).

6.4 Compiling the PathPlanner Using CMKMF

As described in Chapter 5 the path planner can be compiled by typing `cmkmf PathPlanner` from within the directory which contains an Imakefile describing the planner's hierarchy (currently: `/home/tseng/CIRSSE/pathplanner/graphicstest`. That directory also includes the definition of CIRSSE's robot testbed in `robot.def`. Thus after the path planner is compiled it will only plan paths for CIRSSE's robot. In Chapter 7 when we want to plan paths for the NASA Langley robot, we will have to change the `robot.def` file to define the Langley robot.

6.5 Executing CTOS Applications

After the application is built using cmkmf, it is almost ready to be executed. First, a configuration file is needed to define the hardware involved in the application, the tasks which will run in the application, and the CPU each task will run on. Tech Memo #16 describes how to build the necessary configuration files. A sample configuration file is listed in Appendix C.

The hardware then needs to be prepped by setting up "application servers" for each machine involved in the application (**`app_win machine_name`**). Finally, the configuration file is bootstrapped by invoking **`app_bts configFileName`**, which gets the application running on all the chassis.

The path planner can be run in one of three modes, each of which has its own directory, Imakefile file, configuration file, and execution procedure.

1. CIRSSE Demo 1. Autonomous part extraction and insertion; full CIRSSE testbed demonstration. Directory: /home/tseng/CIRSSE/pathplan/ctos. Configuration file: demo1.cfg. Execution: **`app_bts demo1.cfg`**.

2. CTOS Planner Test. Execute auxiliary input file; planner demonstration. Directory: /home/tseng/CIRSSE/pathplan/graphicstest. Configuration file: pp_config. Execution: **`app_bts pp_config`**; application executive queries "input file?".

3. UNIX Planner Test. Stand-Alone planner demonstration using input file. Directory: /home/tseng/CIRSSE/pathplan/unix. Configuration file: none. Execution: **`pp inputFileName`**.

6.6 Demonstration #1 Paths

The path planner has been tested for all paths in the CIRSSE triangle building demonstration. Two example paths are shown here. Figure 6.4 shows the robot's

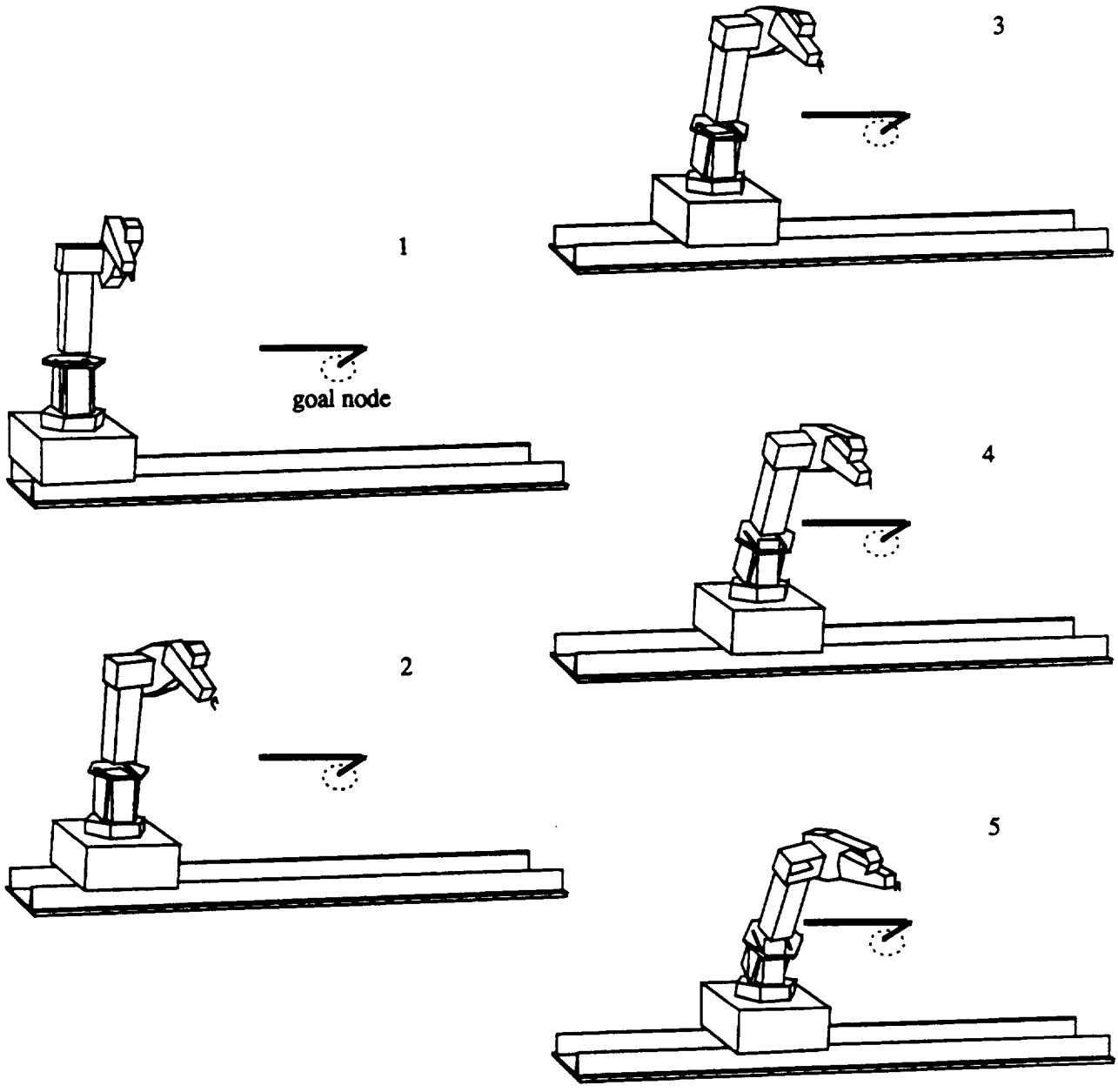


Figure 6.4: Path from Home to the Triangle's Node

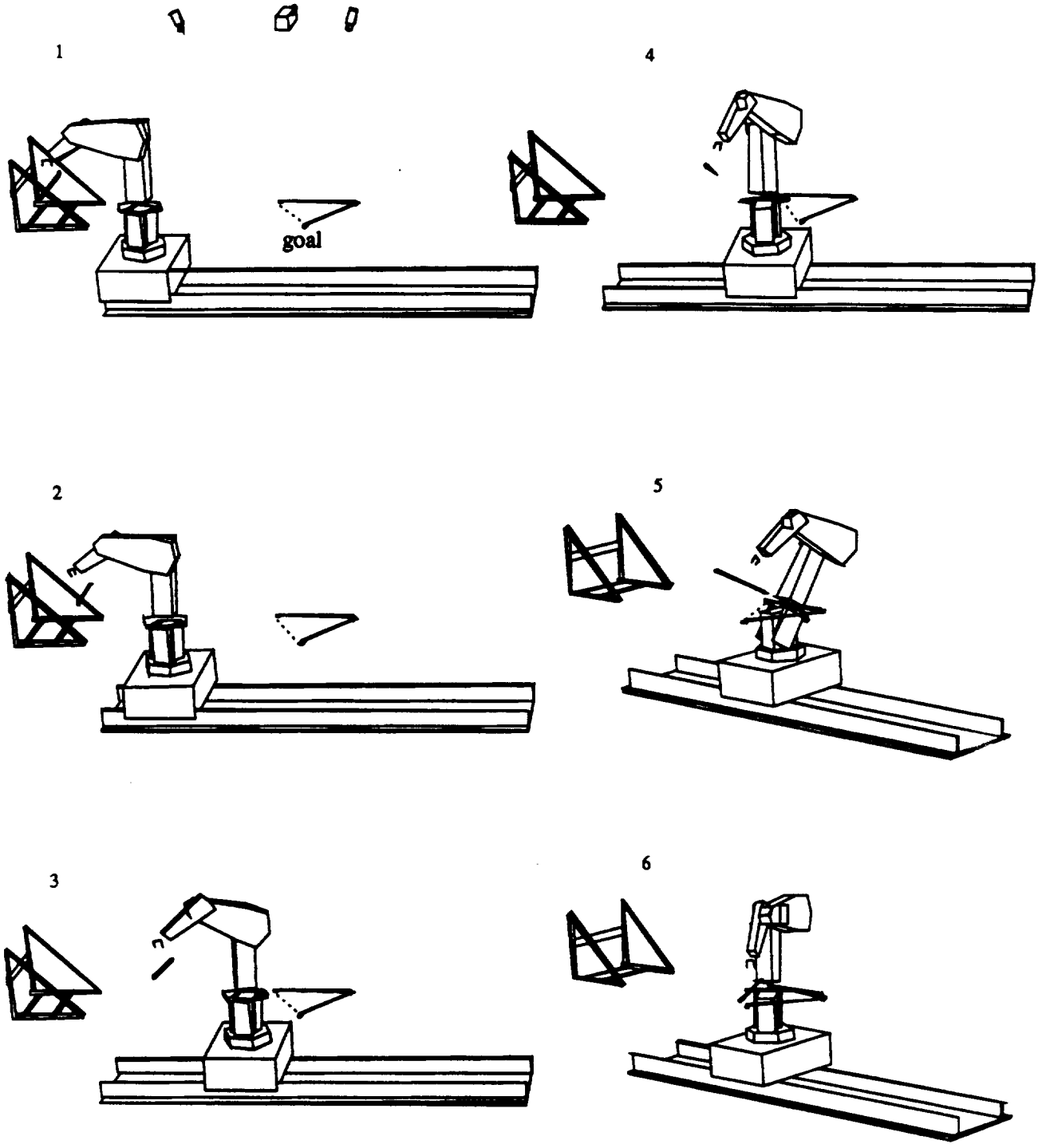


Figure 6.5: Path from Rack to Insertion Point

motion from the home (safe) position to the position above the node where the camera will be used to acquire the triangle structure location accurately. Figure 6.5 shows the path from the strut pick-up point at the rack to the insertion point above the triangle. These paths are plannable for all rack and triangle poses in the robot's workspace.

Although the current CIRSSE demonstration's paths are simple, uncluttered free space moves, the planner is capable of producing more complicated paths. Figure 6.6 shows the CIRSSE robot inserting the last strut into a tetrahedron. In this example, the local planner cannot find a direct path from the start to the goal, because the obstacle creates an unavoidable local minimum (b). After the local planner fails (four times), the global planner creates a subgoal near the obstacle. The local planner reaches this subgoal (c). Then it is called one final time and reaches the original goal (d).

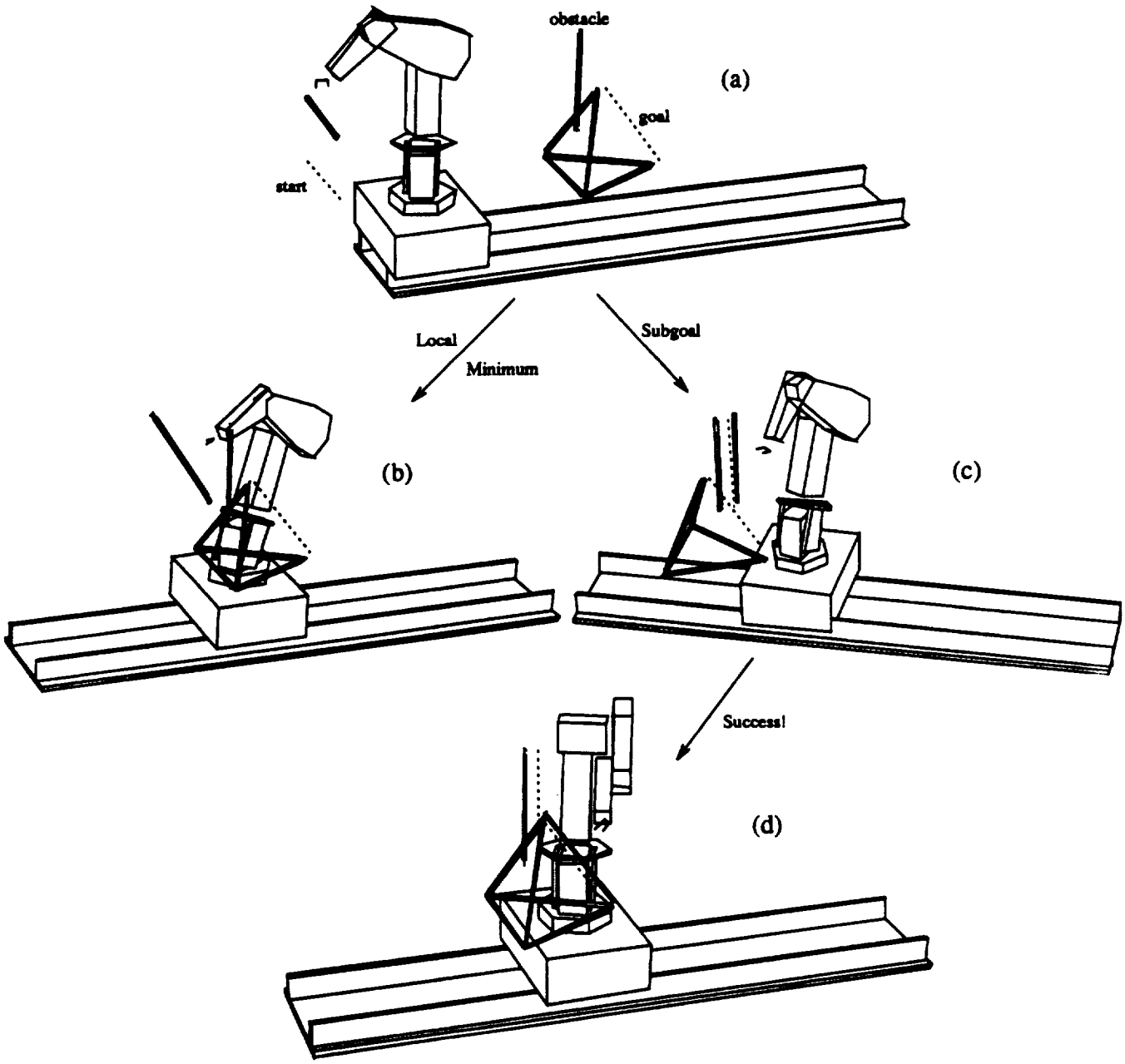


Figure 6.6: Global Subgoal Assists Local Planner

CHAPTER 7
NASA LANGLEY TESTBED

7.1 Physical Plant

Langley Space Center's robot consists of a Merlin 6000 six-DOF articulated arm mounted on a two-DOF cartesian carriage, see Figure 7.1. Another one-DOF rotating platform holds the truss structure assembly. Since the platform does not move during the strut insertion process, it is not modeled by our path planner. The path planner's model of the eight-DOF robot is stored in a definition file `/home/tseng/Langley/ pathplan/ctos/robot.def`, see Section 5.2.13 and Appendix D.

7.2 Langley Planner Requirements

Currently, the planner must move a strut from a bin into a truss assembly. This truss structure is more complicated than the current CIRSSE structure. Figure 7.2 shows a "cell" unit. It is an octahedron, an eight sided figure. By placing cells side by side as in Figure 7.3 [3], we can make arbitrarily large structures.

Figure 7.3 is currently being test assembled at Langley Space Center. The planner must plan paths for each strut without collision. The main obstacles are the robot itself, the struts already in the structure, and 12 triangular panels which are laid above and below the number 1, 2, and 3 cells (forming a pentagon).

7.3 Software

The planner has not been implemented on the Langley Space Center's computer systems. Our path planner for the Langley arm runs on the CIRSSE computer system using CTOS. Therefore, the discussion in Chapter 5 and 6 on CTOS

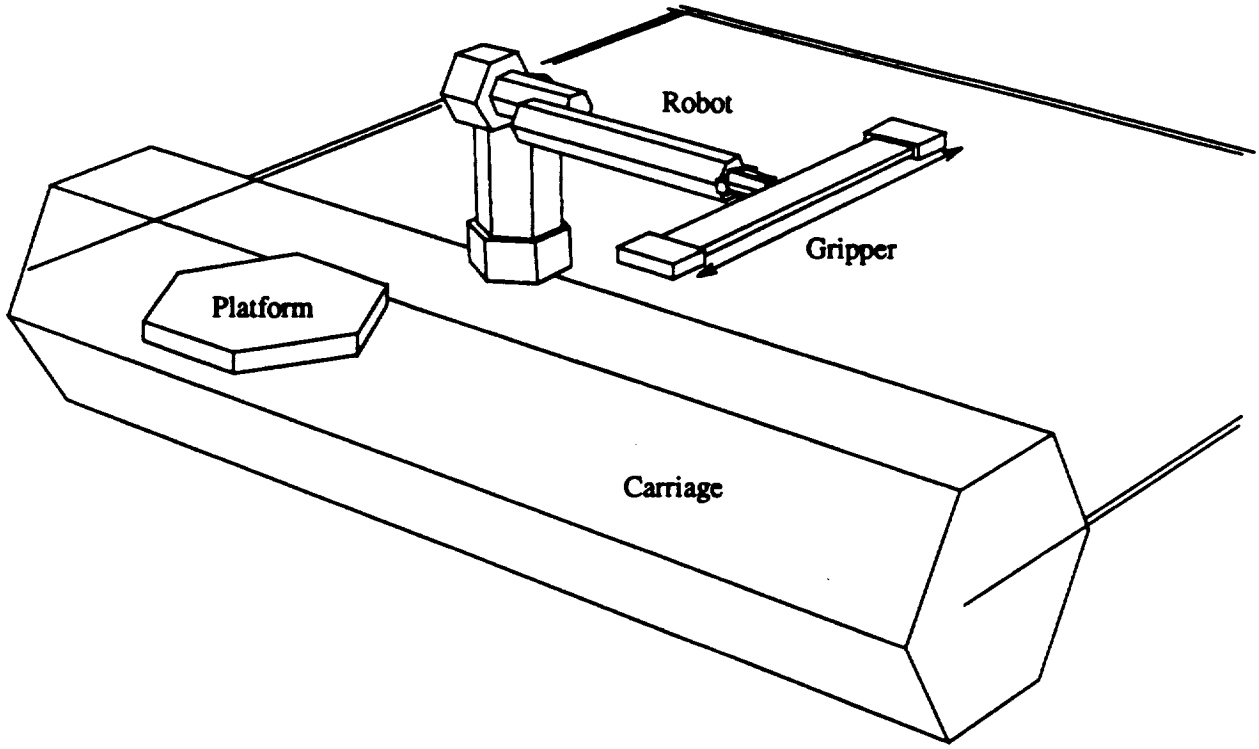


Figure 7.1: Langley Testbed Robots (model)

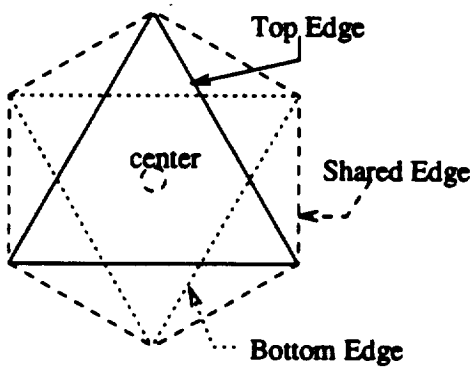


Figure 7.2: A Unit Cell

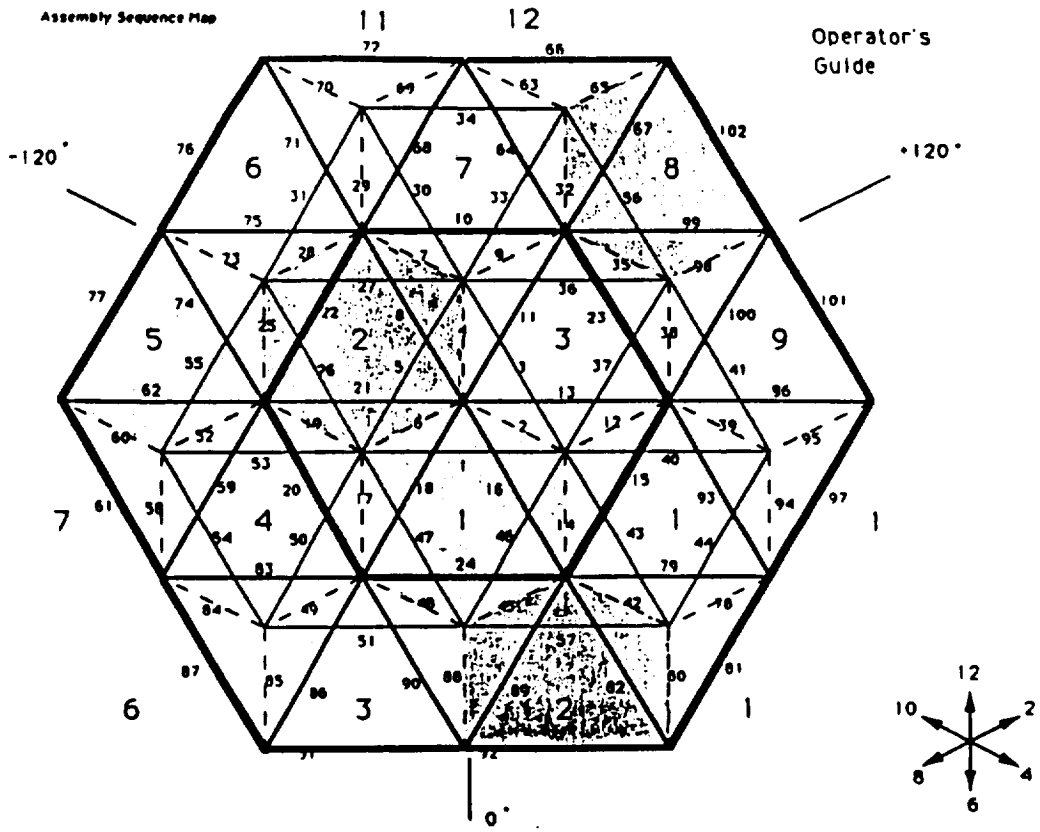


Figure 7.3: Langley Truss Structure with Sequence Numbers

programming still applies. We compile the planner by running `cmkmf` in the directory which contains the Langley robot's definition in its `robot.def` file (currently: `/home/tseng/Langley/pathplan/ctos`).

Presently, there are only two ways to run the planner:

1. CTOS Planner Test. Execute auxiliary input file; planner demonstration. Directory: /home/tseng/Langley/pathplan/ctos. Compile: cmkmf AppExec PathPlanner. Configuration file: pp_config. Execution: `app_bts pp_config`; application executive queries "input file?".

2. UNIX Planner Test. Stand-Alone planner demonstration using input file. Directory: /home/tseng/Langley/pathplan/ctos. Compile: cmkmf pp. Configuration file: none. Execution: `pp inputFileName`.

Both methods of execution produce an output file (*.slm) which is readable by Silma, a professional, graphic robotic engineering package.

7.4 Truss Structure Paths

Our experiments show that the path planner can plan most of the paths for the Langley testbed. Figure 7.4 shows the path from the strut pick-up point in the rack to the insertion point above the tetrahedron.

It works fast: most paths are computed in less than fifteen seconds with an accuracy of plus or minus 1mm. Table 7.4 shows the number of calls to the local planner and the total time that the path planner needed in order to insert the Langley struts shown in Figure 7.3. If the required accuracy is reduced to plus or minus 1cm, then the computation time drops by approximately 7 seconds (see Table 7.4). It does not speed up by ten times as one may be led to believe in Section 3.4.1 because the less accurate paths still must traverse the same amount

Table 7.1: Paths for Langley Robot (1mm accuracy)

Seq.#	Strut#	Time (s)	#calls	comments
1	1	11	1	
2	2	17	1	
3	3	10	1	
4	4	60	4	local plan collides 3 times.
5	5	14	1	near-collision with strut#1.
6	6	19	1	near-collision with strut#5.
7	7	13	1	
8	8	21	1	had hard time acquiring pose.
9	9	23	2	path#1 had a collision.
10	10	29	2	path#1 had oscillations.
11	11	22	1	near a singularity.
12	12	12	1	
24	61	75	6	failure, see Section 8.2.

Table 7.2: Paths for Langley Robot (for 1cm accuracy)

Seq.#	Strut#	Time (s)	#calls	comments
1	1	4	1	not 10x faster due to free space.
2	2	9	1	
3	3	4	1	
4	4	21	3	succeeds on third try.
5	5	7	1	near-collision with strut#1.
6	6	11	1	near-collision with strut#5.
7	7	6	1	
8	8	13	1	had hard time acquiring pose.
9	9	18	2	collision.
10	10	20	2	oscillations.
11	11	15	1	near a singularity.
12	12	6	1	
24	61	75	6	failure, see Section 8.2.

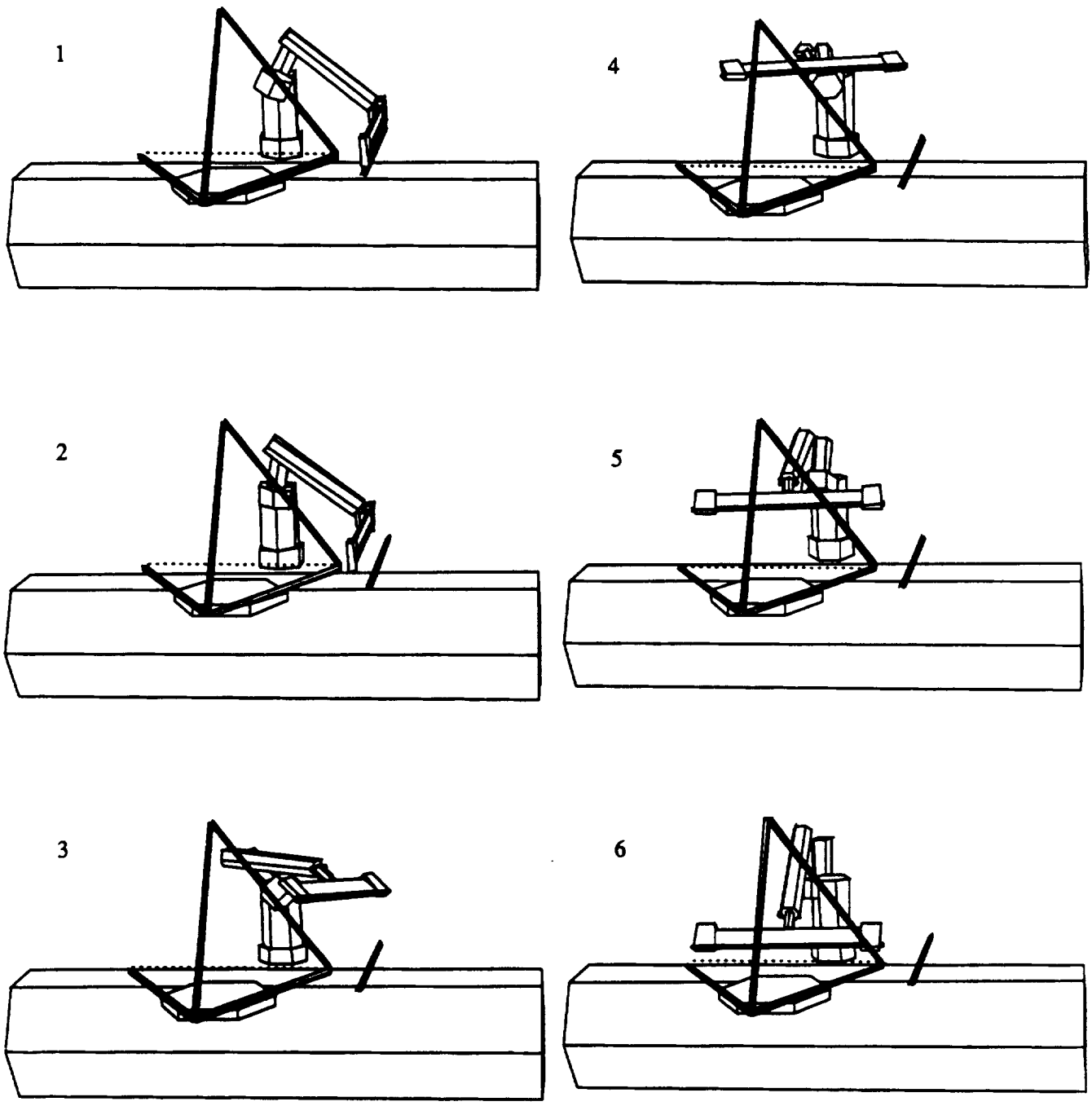


Figure 7.4: Path from Rack to Langley Structure Insertion

of free space as the more accurate paths: only the last centimeter of the path is ten times faster.

Figure 7.5 demonstrates the robustness of our local planner. This insertion into the Langley assembly has very tight tolerances. Munger's planner either would have collided with the structure, if his repulsion constant was too small, or it would have been repulsed by the very large cluster of obstacles, if his repulsion was made too strong. With the combination of flexible fields and object clustering, our planner had no problem finding this path.

Unfortunately, not all paths are plannable for the Langley structure. Figure 7.6 shows the final position of the six failed paths for Langley Strut #61. Attempts numbered 1, 4, and 6 fail due to local minima (which cause oscillations in #4). Attempt #2 collides due to a combination of factors. The largest of which is that the Langley gripper is so large that small steps can move the end of the gripper farther than the *THRESHOLD* adaptive field (Section 3.2.3). Attempt #3 flips the orientation of the goal's end points and is also rotating the long way around the axis of rotation; the path is simply too long, so the planner quits. Finally, in Attempt #5, the robot has been caught in a shoulder-elbow-wrist singularity (reducing the DOF to only 5). With no obstacles nearby to influence it, the path stagnates, and the planner quits. Chapter 8, Section 8.2 discusses our flexible field algorithm's failure to find a path for this example and future improvements which may allow it to find a solution.

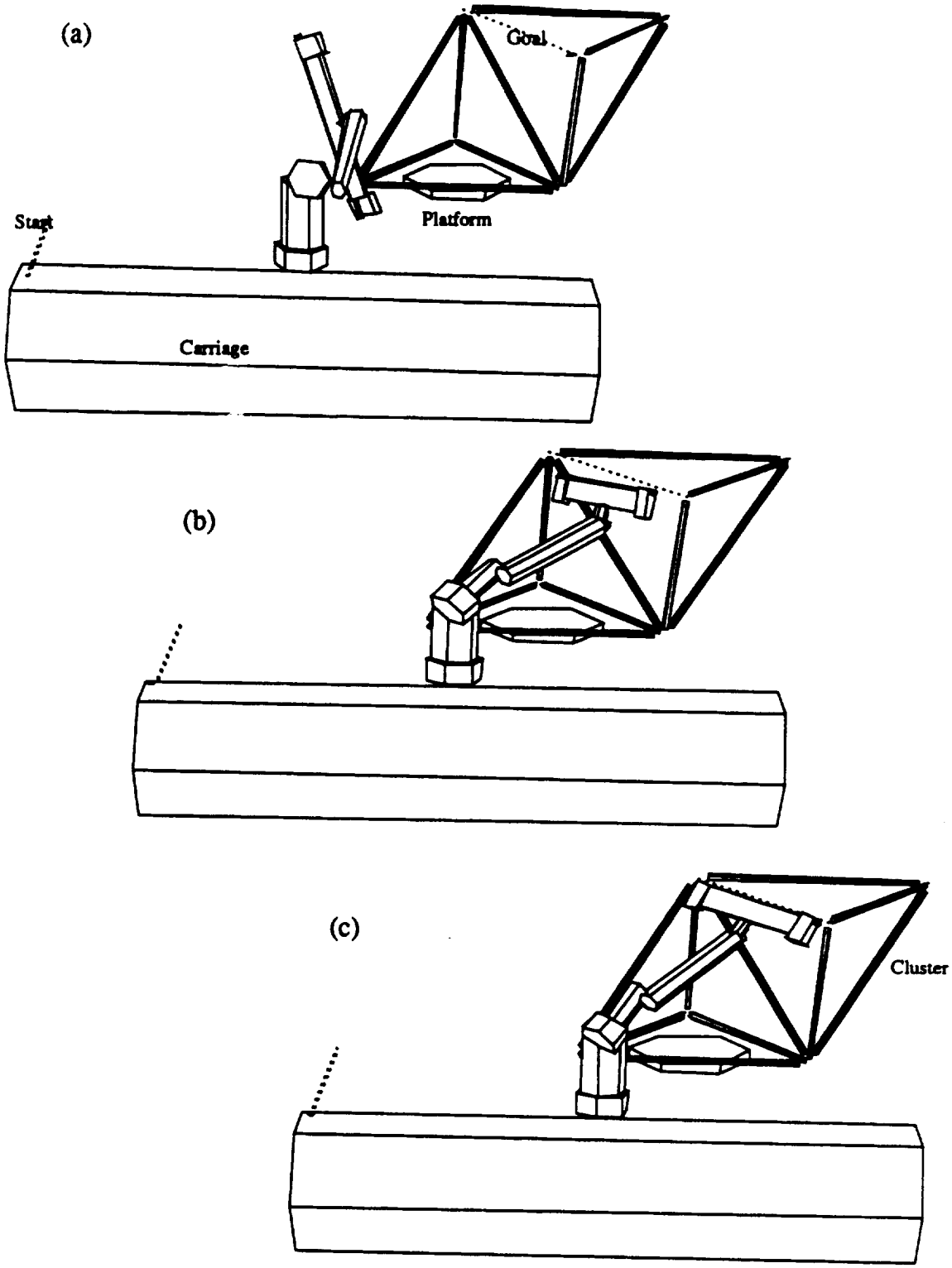


Figure 7.5: Example: Tight Fit and Large Cluster

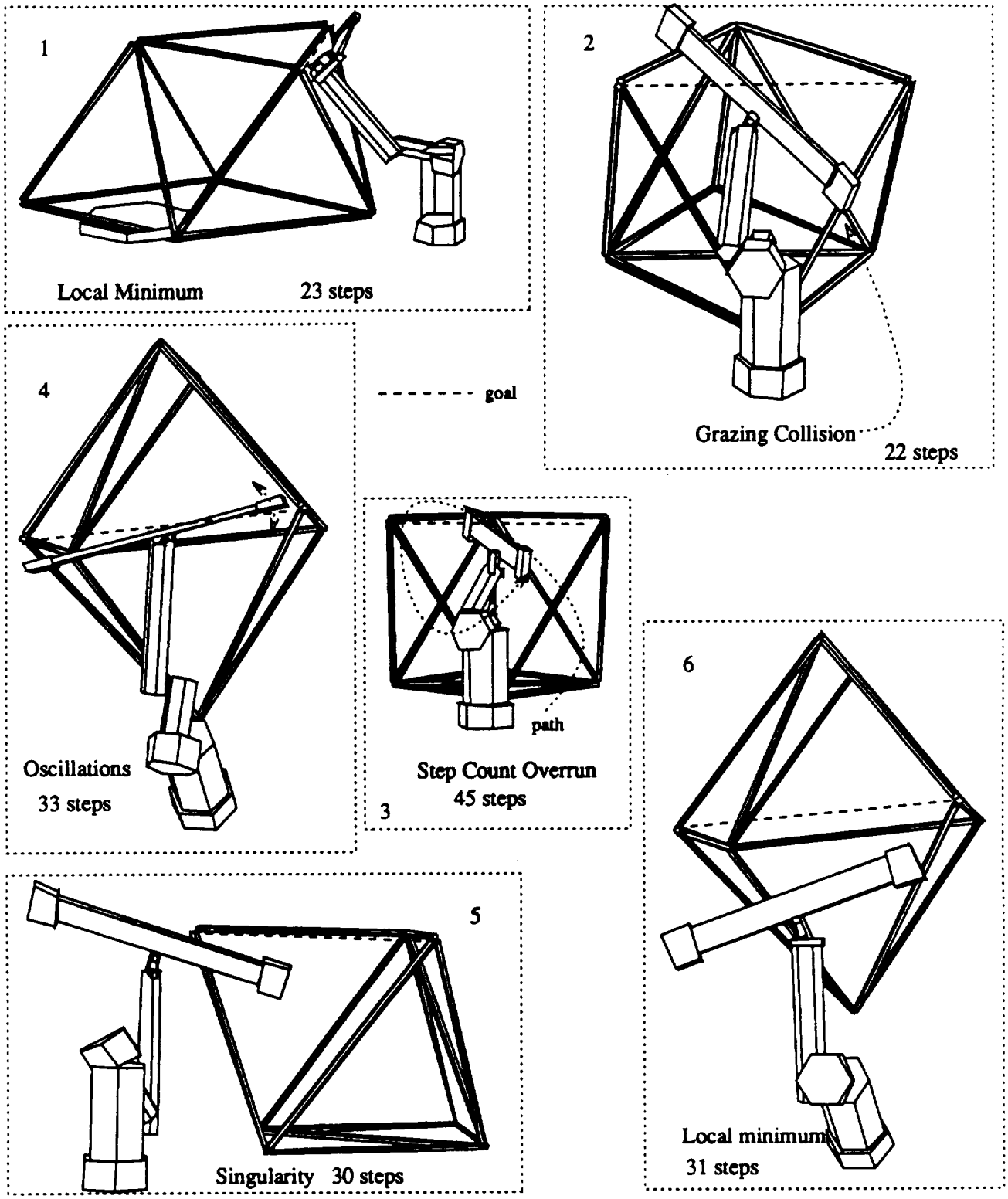

1
Local Minimum
23 steps
2
Grazing Collision
22 steps
goal
4
Oscillations
33 steps
path
Step Count Overrun
45 steps
3
6
Local minimum
31 steps
5
Singularity 30 steps

Figure 7.6:

CHAPTER 8
RESULTS AND CONCLUSIONS

The performance of the path planning algorithm has been tested for the two testbeds presented in the previous chapters. Our results and conclusions are presented in this chapter.

8.1 Computational Complexity

Munger computed the worst case computational complexity of his algorithm to be $O(n^3)$, our improvements' worst case complexity is also $O(n^3)$, therefore we have not raised the order of complexity.

Let us look at the complexity of our major improvements. First, we reduced the jacobian matrix to full rank in Section 3.2.1, the gaussian elimination is $O(n^3)$. Next, keeping the joints in range, Section 3.2.2, requires us to recompute the attraction joint vector each time a joint exceeds its range, since there are n joints, the complexity is $O(n)$. Section 3.2.3.1, clustering, has a worst case complexity $O(nm^2)$, where n is the robot's DOF and m is the number of obstacles. Finally, our flexible-field, repulsion-control factor, and variable step-size computations have constant computation times.

Let us note that since n, the robot's DOF, is constant and usually small, this parameter is not usually critical. m, on the other hand, is variable and can be quite large (Langley's structure has 102 struts and 12 planar panels). Thus, at first glance, the clustering's $O(nm^2)$ complexity seems to be a problem. However, two factors significantly lower its complexity. First, we ignore obstacles which are farther than a threshold distance. Thus, m is lowered to a much more manageable number, an almost constant number that we could call the environment's obstacle density (number of obstacles per cubic meter). Second, the $O(m^2)$ complexity is due

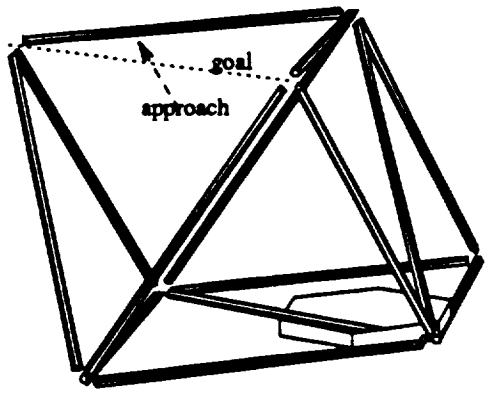


Figure 8.1: Example: Unattainable by Potential Field Method

to direction vector *comparisons*, a task which is many orders of magnitude faster than computing obstacle distances (which is only $O(m)$).

Our experience indicates that in practice, once the robot's DOF and the environment's obstacle density are given, the local path planner is fairly constant, computationally. A far more vexing problem is that the local planner is called on by the global planner to establish the visibility of graph edges. In the worst case, it is called $O(n^2)$ times where n is the number of subgoals. Since at worst, the number of subgoals equals twice the number of obstacles, this is very bad indeed. To eliminate the worst case, we have set a maximum number of calls to the local path planner (currently at 6). A better way might be to quit when any global path contains more than a certain number of nodes, assuming that too many nodes implies that the path is too contorted.

Finally, Munger's subgoal extraction algorithm is $O(m^2)$ where m is the number of struts. Our closest-failure-strut approach has a constant computation time.

8.2 Weaknesses

Our refinements to the local path planning algorithm have not completely solved the fundamental problem of local minima. Figure 8.1 shows a Langley truss structure into which our planner could not insert the strut (see Figure 7.6, Table 7.4, Strut #61). The insertion would not be difficult if the approach direction was not pointing from the interior of the structure, outward. This insertion, however, requires the strut to be slid into the interior of the structure and then rotated to the correct orientation. Thus the strut must rise against the gradient before it can ride down to the global minimum; this the local planner will never do. Our subgoals are ineffective aids because they are parallel to the existing struts and are exterior to the cell.

Lozano-Perez [6] and Schwartz-Sharir [25] have tried to solve this type of problem with subgoals at the local minima, but they have had problems because the attraction well of the local minimum is too large. Barraquand uses random motions to solve this problem [16]. If speed of computation is not necessary, a global method may be used; this particular problem has been solved by Weaver's Divide and Conquer Planner [4].

The local path planner is an iterative algorithm which converges slowly on the required accuracy. Comparing the execution times in Table 7.4 and Table 7.4, the maximum difference between plans with only one call to the local planner is 8 seconds. One way to save most of these 8 seconds would be to use an inverse kinematics routine once the local planner had found the goal to within some rough approximation (1cm). We would thus forgo the many iterations needed to converge on an accurate solution.

One feature that our algorithm does not provide is the ability to choose the final configuration of the robot. For reasons of safety, load bearing, and appearance, the operator may wish to choose the wrist flip, elbow up-down, or shoulder left-right configuration. These goals could be accomplished by using an attractive force to

the goal configuration instead of the goal EE pose, or by limiting the joint ranges of the robot.

This concludes the discussion of the proposed path planning algorithm.

LITERATURE CITED

[1] Rolf Münger (1991). *Path Planning for Assembly of Strut-Based Structures.* CIRSSE Report #91, Rensselaer Polytechnic Institute, Troy, NY.

[2] Rolf Münger (1992). *Assembly Path Planning.* Intelligent Robotic Systems for Space Exploration, ed. Alan A. Desrochers, Kluwer Academic Publishers, Boston, pp. 155 - 184.

[3] Alan A. Desrochers, editor (1992). *Intelligent Robotic Systems for Space Exploration* Rensselaer Polytechnic Institute, Kluwer Academic Publishers, Boston.

[4] J.M. Weaver, S.J. Derby (1992). *A Divide and Conquer Method of Path Planning for Cooperating Robots with Stringing Tightening.* Fourth Annual Conf. on Intelligent Robotic Systems for Space Exploration, Rensselaer Polytechnic Institute, Troy, NY, pp. 30 - 40.

[5] Tomás Lozano Pérez, M. A. Wesley (1979). *An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles.* Communications of the ACM, Vol. 22, 10, October 1979, pp. 560 - 570.

[6] Tomás Lozano Pérez (1983). *Spacial Planning: A Configuration Space Approach.* IEEE Transactions on Computers, Vol C-32, No. 2, February 1983, pp. 108 - 120.

[7] Francis Avnaim, Jean Daniel Boissonnat, Bernard Faverjon (1988). *A Practical Exact Motion Planning Algorithm for Polygonal Objects Amidst Polygonal Obstacles* IEEE 1988 International Conference On Robotics & Automation, Vol. 3, pp. 1656 - 1661.

[8] Walter Meyer, Powell Benedict (1988). *Path Planning and the Geometry of Joint Space Obstacles.* IEEE 1988 International Conference On Robotics & Automation, Vol. 1, pp. 215 - 219.

[9] Karen Anderson, Jorge Angeles (1989). *Kinematic Inversion of Robotic Manipulators in the Presence of Redundancies* International Journal of Robotics Research, Vol. 8, No. 6, December 1989, pp. 80 - 97.

[10] Jorge Angeles (1985). *On the Numerical Solution of the Inverse Kinematic Problem.* International Journal of Robotics Research, Vol. 4 No. 2, Summer 1985, pp. 21 - 37.

[11] J. Angeles, K. Anderson, X. Cyril, B. Chen (1988). *The Kinematic Inversion of Robot Manipulators in the Presence of Singularities.* Transactions of the ASME, Vol. 110, September 1988, pp. 246 - 254.

[12] R.A. Brooks, T. Lozano-Pérez (1983) *A Subdivision Algorithm in Configuration Space for Find-Path with Rotation.* Proc. of the 8th Int. Joint Conf. on Artificial Intelligence, Karlsruhe, FRG, pp. 799 - 806.

[13] Sungteg Jun, Kang G. Shin (1988). *A Probabilistic Approach to Collision-Free Robot Path Planning.* IEEE 1988 International Conference On Robotics & Automation, Vol. 1, pp. 220 - 225.

[14] Brad Paden, Alistair Mees, Mike Fisher (1989). *Path Planning Using a Jacobian-Based Freespace Generation Algorithm.* IEEE 1989 International Conference On Robotics & Automation, Vol. 3, pp. 1732 - 1737.

[15] C. DeMedio, F. Nicolò, G. Oriolo. *Robot Motion Planning Using Vortex Fields.* New Trends in System Theory, Genova, Italy, July 1990.

[16] Jerome Barraquand (1991). *Automatic Motion Planning for Complex Articulated Bodies.* Digital Computer Inc., Report #14, June 1991.

[17] Richard Volpe, Pradeep Khosla (1987). *Artificial Potentials with Elliptical Isopotential Contours for Obstacle Avoidance* IEEE Proceedings of the 26th Conference on Decision and Control, Vol. 1, December 1987, pp. 180 - 185.

[18] P. Khosla, R. Volpe (1988). *Superquadric Artificial Potentials for Obstacle Avoidance and Approach.* Proc. of the IEEE Int. Conf. on Robotics and Automation, Philadelphia, PA, 1778 - 1784.

[19] E. Rimon, D.E. Koditschek (1989). *The Construction of Analytic Diffeomorphisms for Exact Robot Navigation on Start Worlds.* Proc. of the IEEE Int. Conf. on Robotics and Automation, Scottsdale, pp. 21 - 26.

[20] Yutaka Kanayama (1988). *Least Cost Paths with Algebraic Cost Functions.* IEEE 1988 International Conference On Robotics & Automation, Vol. 1, pp. 75 - 80.

[21] Bernard Faverjon, Pierre Fournassoud (1987). *A Local Based Approach for Path Planning of Manipulators With a High Number of Degrees of Freedom.* IEEE 1987 International Conference On Robotics & Automation, Vol. 2, pp. 1152 - 1159.

[22] S. L. Campbell, C. D. Meyer Jr. (1979). *Generalized Inverses of Linear Transformations.* p. 251, Pitman; London, San Francisco, Melbourne.

[23] Patrick Henry Winston (1984). *Artificial Intelligence.* Addison-Wesley, pp. 87, 113 - 114.

[24] Elaine Rich (1983). *Artificial Intelligence.* McGraw-Hill Series in Artificial Intelligence, pp. 80 - 84.

[25] J.T. Schwartz and M. Sharir (1983). *On the 'Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds.* Advances in Applied Mathematics. Academic Press 4, pp. 298 - 351.

[26] L.W. Johnson, R.D. Riess (1981). *Introduction to Linear Algebra.* Virginia Polytechnic Institute, Addison-Wesley, pp. 280 - 287.

[27] J.J. Craig (1986). *Introduction to Robotics, Mechanics, and Control.* Addison-Wesley, Chapter 3.

[28] Josep Tornero, Greg Hamlin (1990). *Spherical-Object Representation and Fast Distance Computation for Robotic Applications.* CIRSSE Report #64, Rensselaer Polytechnic Institute, Troy, New York, September 1990.

[29] Donald Lefebvre, (1990). */usr2/testbed/stable/unix/demos/demo1/pathplanner/ppLib.c* Rensselaer Polytechnic Institute, Troy, New York, 1993.

[30] Keith Nicewarner, (1992). *The Geometric State Manager.* CIRSSE Tech Memo #21 (v. 1), Rensselaer Polytechnic Institute, Troy, New York, September 1992.

[31] L. Carmichael, (1992). *On the Use of the Inverse Kinematics for the 9-DOF Manipulator.* CIRSSE Tech Memo #18 (v. 1), Rensselaer Polytechnic Institute, Troy, New York, June 1992.

APPENDIX A
Imakefile File Example

```
/*  Imakefile for making PathPlanner, AppExec, and pp (UNIX version)  */

 /*  path planner includes  */
CPPFLAGS += -I/usr2/testbed/exp/unix/include
CPPFLAGS += -I/usr/old
CPPFLAGS += -L/usr/old

 /* path planner libraries */
LDLIBS += -lcore
LDLIBS += -lsuntool
LDLIBS += -lsunwindow
LDLIBS += -lpixrect
LDLIBS += -lpp
LDLIBS += -lmsg -lbts -lrec -lctos
LDLIBS += -lclifClient
LDLIBS += -lknatpt
LDLIBS += -lconfig
LDLIBS += -lkin
LDLIBS += -ltrans
LDLIBS += -ltranParams
LDLIBS += -lm


OBJS = alg.o env.o global.o graph.o graphics.o gpath.o lpath.o lst.o
model.o parser.o robot.o spec.o stack.o vector.o


AllTarget(libpp.a  PathPlanner  ppmain.o  AppExec)


UNIXBinTarget(PathPlanner,  PathPlanner.o  ppmain.o  $(OBJS))
UNIXBinTarget(AppExec,  AppExec.o)
UNIXBinTarget(pp,  main.o  ppmain.o  $(OBJS))
UNIXLibTarget(libpp.a,  ppLib.o)
```

APPENDIX B

Simulation Input File

```
{ Command sequence : Build a tetrahedron
  ------------------------------------ }

  Start

 {  first strut  }
  move (1.79, -0.8, 1.0, 0.9, -0.8, 1.0, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 1.0, 0.9, -0.8, 1.0)
  move (0, 0, 0, 1, 0.0, 0.0, -1.0)
  ungrasp (0, 0, 0, 1)
 {  second strut  }
  move (1.79, -0.8, 0.9, 0.9, -0.8, 0.9, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 0.9, 0.9, -0.8, 0.9)
  move (0, 0, 0, 2, 0.0, 0.0, -1.0)
  ungrasp (0, 0, 0, 2)
 {  third strut  }
  move (1.79, -0.8, 0.8, 0.9, -0.8, 0.8, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 0.8, 0.9, -0.8, 0.8)
  move (0, 0, 0, 3, 0.0, 0.0, -1.0)
  ungrasp (0, 0, 0, 3)
 {  fourth strut  }
  move (1.79, -0.8, 0.7, 0.9, -0.8, 0.7, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 0.7, 0.9, -0.8, 0.7)
  move (0, 0, 0, 4, 1.0, 0.0, 0.0)
  ungrasp (0, 0, 0, 4)
 {  fifth strut  }
  move (1.79, -0.8, 0.6, 0.9, -0.8, 0.6, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 0.6, 0.9, -0.8, 0.6)
  move (0, 0, 0, 5, 0.0, 1.0, -1.0)
  ungrasp (0, 0, 0, 5)
 {  sixth strut  }
  move (1.79, -0.8, 0.5, 0.9, -0.8, 0.5, 0.0, 0.0, -1.0)
  grasp (1.79, -0.8, 0.5, 0.9, -0.8, 0.5)
  move (0, 0, 0, 6, 1.0, 0.0, -1.0)
  ungrasp (0, 0, 0, 6)

  quit

{ Environment info
  ---------------- }
  X_graphics
  Diagnostics
  strutlength (0.89)
  structure_loc (0.8, -0.1, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0)

  strut (1.79, -0.8, 1.0, 0.9, -0.8, 1.0)
  strut (1.79, -0.8, 0.9, 0.9, -0.8, 0.9)
  strut (1.79, -0.8, 0.8, 0.9, -0.8, 0.8)
  strut (1.79, -0.8, 0.7, 0.9, -0.8, 0.7)
  strut (1.79, -0.8, 0.6, 0.9, -0.8, 0.6)
  strut (1.79, -0.8, 0.5, 0.9, -0.8, 0.5)

  tetra (2, 1, 0)
```

APPENDIX C

CTOS Application Cqnfiguration File

```
#  Example Config file for Planner and GSM's Viewer

chassis mars                                    # 'mars' is the machine name

sequencer mars

PREFIX SequHost 0
args  recSvr        REC_OPTIONS   = -rv -geometry 500x800+0 -name Demo_1
args  recSvr        XCTOS_DISPLAY = mars:0.0

PREFIX  mars  0                                 # put the following tasks on mars
chdir  /home/tseng/CIRSSE/pathplan/graphicstest
task  PathPlanner PathPlanner
task  AppExec     AppExec

CHDIR /usr2/testbed/stable/unix/bin/sun4
task  gsmServer gsmServer

systask viewer /usr2/testbed/stable/unix/bin/sun4/xctosParent
args    viewer XCTOS_PROG = /usr2/testbed/stable/unix/bin/sun4/fxctosviewer
args    viewer XCTOS_DISPLAY = mars:0.0         # display viewer on mars' screen
```

APPENDIX D
Robot Definition Files

CIRSSE 9-DOF Robotic Testbed Definition (/home/tseng/CIRSSE/pathplan/ctos/robot.def file):

```
/* robot parameters for CIRSSE 9 DOF PUMA arm plus platform (left arm) */
/* ================================================================== */

/* This file contains all parameters of the robot used in the assembly
** task. It's $included into robot.c.
** robot.c is general in the sense that all information about a specific
** robot is in robot.def.
** robot.c assumes a single chain robot with prismatic or revolute joints
** and a minimum of 6 DOF.
**-------------------------------------------------------------------------*/

$ifndef ROBOT_CODE
 /* degrees of freedom: */
 $define DOF 9

$else

/* robot is not symmetric due to gripper pneumatic cords */
$  undef SYMMETRIC_ROBOT

/* joint types. First entry is joint closest to the base. PRISM or REV. */
static int j_type[DOF] = {PRISM, REV, REV, REV, REV, REV, REV, REV, REV};


/* kinematic parameters on modified Denavit Hartenberg form.
** Arrays for a, d, alpha and theta.
** Units:   meters for a and d, degrees for alpha and theta.
** Indices: a[0] = a0   alpha[0] = alpha0   d[0] = d1   theta[0] = theta1
** In a revolute joint theta is the variable in q, whereas in a prismatic
** joint d is the variable in the joint vector q. Enter a 'Q' if the
** corresponding value is part of the q vector.
*/
static float     a[DOF] = {.32, .0, .0, .0, .0, .43182, -.02031, .0, .0};
static float     d[DOF] = {Q, .544, .0, .828, .243, -.09391, .433, .0, .0};
static float alpha[DOF] = {-90., 90.,-90., 90.,-90., .0, 90.,-90., 90.};
static float theta[DOF] = {.0, Q  , Q  , Q  , Q  , Q  , Q  , Q  , Q  };


/* end effector matrix.
** This homogeneous matrix describes the transformation from the last link
** to the gripper. In this case this is a simple translation along the
** z-axis.
** IMPORTANT! The vectors in the 3x3 matrix in the constant definition below
** ---------- are COLUMN vectors, even though they look like row vectors!
*/
static H_Matrix trE = {{{1.0, 0.0, 0.0},
                        {0.0, 1.0, 0.0},
                              {0.0, 0.0, 1.0}}, {0.0, 0.0, 0.24}};


/* Joint value ranges for left arm. First entries are closer to the base.
```

```
** Units are meters for prismatic and degrees for revolute joints.
*/
static float ql_min[DOF] =
  {-1.3216,-150.,-45.,-251.,-215.,-55.,-121.,-95.,-284.};
static float ql_max[DOF] =
  { 0.6096, 150., 45.,  74.,  34., 241., 144., 95., 284.};
/* Joint ranges for right arm */
static float qr_min[DOF] =
  {-0.6096,-150.,-45.,-248.,-215.,-55.,-129.,-95.,-284.};
static float qr_max[DOF] =
  { 1.3716, 150., 45.,  78.,  37.,238., 148., 95., 284.};


/* Joint weights used for solving the Jacobian equation. The values of this
** vector define a diagonal matrix Q with the elements of 'joint_weight' on
** the diagonal. The (in general redundant) Jacobian equation is solved such
** that q'Qq is minimized. (q' is the transpose of q).
*/
static joint_weight[DOF] = {16.0, 16.0, 32.0, 4.0, 6.0, 4.0, 2.0, 1.0, 1.0};


/* constants for definition of picture and model */
#define RX0            0.54
#define RY0            1.8

#define RX1a           0.17
#define RX1b          -0.35
#define RY1           -0.2
#define RZ1            0.36

#define RR3            0.09
#define RR3b           0.2
#define RY3a          -0.1683
#define RY3b          -0.1683 - 0.6604

#define RR4            0.09
#define RY4            0.2

#define RX5a          -0.15
#define RX5b           0.08
#define RX5c           0.482
#define RY5a           0.15
#define RY5b           0.07
#define RZ5a          -0.043
#define RZ5b           0.05

#define RX6a          -0.09
#define RX6b           0.05
#define RX6c          -0.06
#define RX6d           0.02
#define RX6m          -0.02
#define RR6            0.07
#define RY6a           0.06  /* was .07 */
#define RY6b          -0.423
#define RZ6a           0.05091
#define RZ6b          -0.04

#define RR7            0.04
#define RX7a           0.06
#define RX7b          -0.06
#define RZ7a           0.11
#define RZ7b           0.23

#define RZ9a           0.105
#define RZ9b           0.14
#define RR9            0.03
```

```
/* 'Link_Model' assumes a cylindrical link model. The first 6 parameters
** are the cylinder's endpoints, the 7th is the cylinder's radius and the
** last one is the link number.
*/
static void Get_Link_Models ()
{
  New_Link_Model (RX0/2.0, -RY0, 0.0, RX0/2.0, RY0, 0.0, RX0/2.0, 0);
  New_Link_Model (RX1a, RY1, 0.0, RX1b, RY1, 0.0, RZ1, 1);
  New_Link_Model (0.0, RY3a, 0.0, 0.0, RY3b, 0.0, RR3, 3);
  New_Link_Model (RX5a, 0.0, 0.0, RX5c, 0.0, 0.0, RY5a, 5);
  New_Link_Model (RX6m, RY6a, 0.0, RX6m, RY6b, 0.0, RR6, 6);
  New_Link_Model (RX7a, 0.0, RZ7a, RX7b, 0.0, RZ7a, RR7, 7);
  New_Link_Model (0.0, 0.0, RZ9a, 0.0, 0.0, RZ9b, RR9, 9);
}


/* The definition of the robot's picture on the screen. The picture is
** a wire frame - every line of this frame is defined here. The six
** first parameters of the 'Link_Line' procedure are  the two endpoints
** of the line in the local frame. Unit: meters.
** The last parameter is the link number.  Note: outdated SUN graphics.
*/
static void Get_Link_Pictures ()
{
  New_Link_Line (0.0, -RY0, 0.0, 0.0, RY0, 0.0, 0);
  New_Link_Line (RX0, -RY0, 0.0, RX0, RY0, 0.0, 0);

  New_Link_Line (RX1a, RY1,  RZ1, RX1b, RY1,  RZ1, 1);
  New_Link_Line (RX1a, RY1, -RZ1, RX1b, RY1, -RZ1, 1);
  New_Link_Line (RX1a, RY1,  RZ1, RX1a, RY1, -RZ1, 1);
  New_Link_Line (RX1b, RY1,  RZ1, RX1b, RY1, -RZ1, 1);

  New_Link_Line (-RR3b, RY3a, -RR3b,  RR3b, RY3a, -RR3b, 3);
  New_Link_Line (-RR3b, RY3a,  RR3b,  RR3b, RY3a,  RR3b, 3);
  New_Link_Line (-RR3b, RY3a, -RR3b, -RR3b, RY3a,  RR3b, 3);
  New_Link_Line ( RR3b, RY3a, -RR3b,  RR3b, RY3a,  RR3b, 3);

  New_Link_Line (-RR3, RY3a, -RR3,  RR3, RY3a, -RR3, 3);
  New_Link_Line (-RR3, RY3a,  RR3,  RR3, RY3a,  RR3, 3);
  New_Link_Line (-RR3, RY3a, -RR3, -RR3, RY3a,  RR3, 3);
  New_Link_Line ( RR3, RY3a, -RR3,  RR3, RY3a,  RR3, 3);

  New_Link_Line (-RR3, RY3b, -RR3,  RR3, RY3b, -RR3, 3);
  New_Link_Line (-RR3, RY3b,  RR3,  RR3, RY3b,  RR3, 3);
  New_Link_Line (-RR3, RY3b, -RR3, -RR3, RY3b,  RR3, 3);
  New_Link_Line ( RR3, RY3b, -RR3,  RR3, RY3b,  RR3, 3);

  New_Link_Line (-RR3, RY3a, -RR3, -RR3, RY3b, -RR3, 3);
  New_Link_Line (-RR3, RY3a,  RR3, -RR3, RY3b,  RR3, 3);
  New_Link_Line ( RR3, RY3a, -RR3,  RR3, RY3b, -RR3, 3);
  New_Link_Line ( RR3, RY3a,  RR3,  RR3, RY3b,  RR3, 3);

  New_Link_Line (-RR4, -RR4, -RR4,  RR4, -RR4, -RR4, 4);
  New_Link_Line (-RR4, -RR4,  RR4,  RR4, -RR4,  RR4, 4);
  New_Link_Line (-RR4, -RR4, -RR4, -RR4, -RR4,  RR4, 4);
  New_Link_Line ( RR4, -RR4, -RR4,  RR4, -RR4,  RR4, 4);

  New_Link_Line (-RR4, RY4, -RR4,  RR4, RY4, -RR4, 4);
  New_Link_Line (-RR4, RY4,  RR4,  RR4, RY4,  RR4, 4);
  New_Link_Line (-RR4, RY4, -RR4, -RR4, RY4,  RR4, 4);
  New_Link_Line ( RR4, RY4, -RR4,  RR4, RY4,  RR4, 4);

  New_Link_Line (-RR4, -RR4, -RR4, -RR4, RY4, -RR4, 4);
  New_Link_Line (-RR4, -RR4,  RR4, -RR4, RY4,  RR4, 4);
```

```
  New_Link_Line ( RR4, -RR4, -RR4,  RR4, RY4, -RR4, 4);
  New_Link_Line ( RR4, -RR4,  RR4,  RR4, RY4,  RR4, 4);

  New_Link_Line (RX5a, -RY5a, RZ5a, RX5a,  RY5a, RZ5a, 5);
  New_Link_Line (RX5a, -RY5a, RZ5b, RX5a,  RY5a, RZ5b, 5);
  New_Link_Line (RX5a, -RY5a, RZ5a, RX5a, -RY5a, RZ5b, 5);
  New_Link_Line (RX5a,  RY5a, RZ5a, RX5a,  RY5a, RZ5b, 5);

  New_Link_Line (RX5a, -RY5a, RZ5a, RX5b, -RY5a, RZ5a, 5);
  New_Link_Line (RX5a,  RY5a, RZ5a, RX5b,  RY5a, RZ5a, 5);
  New_Link_Line (RX5a, -RY5a, RZ5b, RX5b, -RY5a, RZ5b, 5);
  New_Link_Line (RX5a,  RY5a, RZ5b, RX5b,  RY5a, RZ5b, 5);

  New_Link_Line (RX5b, -RY5a, RZ5a, RX5c, -RY5b, RZ5a, 5);
  New_Link_Line (RX5b,  RY5a, RZ5a, RX5c,  RY5b, RZ5a, 5);
  New_Link_Line (RX5b, -RY5a, RZ5b, RX5c, -RY5b, RZ5b, 5);
  New_Link_Line (RX5b,  RY5a, RZ5b, RX5c,  RY5b, RZ5b, 5);

  New_Link_Line (RX5c, -RY5b, RZ5a, RX5c,  RY5b, RZ5a, 5);
  New_Link_Line (RX5c, -RY5b, RZ5b, RX5c,  RY5b, RZ5b, 5);
  New_Link_Line (RX5c, -RY5b, RZ5a, RX5c, -RY5b, RZ5b, 5);
  New_Link_Line (RX5c,  RY5b, RZ5a, RX5c,  RY5b, RZ5b, 5);

  New_Link_Line (RX6a, RY6a, RZ6a, RX6b, RY6a, RZ6a, 6);
  New_Link_Line (RX6a, RY6a, RZ6b, RX6b, RY6a, RZ6b, 6);
  New_Link_Line (RX6a, RY6a, RZ6a, RX6a, RY6a, RZ6b, 6);
  New_Link_Line (RX6b, RY6a, RZ6a, RX6b, RY6a, RZ6b, 6);

  New_Link_Line (RX6a, RY6a, RZ6a, RX6c, RY6b, RZ6a, 6);
  New_Link_Line (RX6b, RY6a, RZ6a, RX6d, RY6b, RZ6a, 6);
  New_Link_Line (RX6a, RY6a, RZ6b, RX6c, RY6b, RZ6b, 6);
  New_Link_Line (RX6b, RY6a, RZ6b, RX6d, RY6b, RZ6b, 6);

  New_Link_Line (RX6c, RY6b, RZ6a, RX6d, RY6b, RZ6a, 6);
  New_Link_Line (RX6c, RY6b, RZ6b, RX6d, RY6b, RZ6b, 6);
  New_Link_Line (RX6c, RY6b, RZ6a, RX6c, RY6b, RZ6b, 6);
  New_Link_Line (RX6d, RY6b, RZ6a, RX6d, RY6b, RZ6b, 6);

  New_Link_Line (-RR7, -RR7, RZ7a,  RR7, -RR7, RZ7a, 7);
  New_Link_Line (-RR7,  RR7, RZ7a,  RR7,  RR7, RZ7a, 7);
  New_Link_Line (-RR7, -RR7, RZ7a, -RR7,  RR7, RZ7a, 7);
  New_Link_Line ( RR7, -RR7, RZ7a,  RR7,  RR7, RZ7a, 7);

  New_Link_Line (-RR7, -RR7, RZ7a, -RR7, -RR7, RZ7b, 7);
  New_Link_Line ( RR7, -RR7, RZ7a,  RR7, -RR7, RZ7b, 7);
  New_Link_Line (-RR7,  RR7, RZ7a, -RR7,  RR7, RZ7b, 7);
  New_Link_Line ( RR7,  RR7, RZ7a,  RR7,  RR7, RZ7b, 7);

  New_Link_Line (-RR7, -RR7, RZ7b,  RR7, -RR7, RZ7b, 7);
  New_Link_Line (-RR7,  RR7, RZ7b,  RR7,  RR7, RZ7b, 7);

  New_Link_Line (RX7a, 0.0, RZ7a-0.04, RX7a, 0.0, RZ7a+0.04, 7);
  New_Link_Line (RX7b, 0.0, RZ7a-0.04, RX7b, 0.0, RZ7a+0.04, 7);
}


/* The following information deals with collision avoidance between links.
** It's a table with dimension (DOF+1)*(DOF+1). Insert a TRUE if a
** collision between the column and the row link is possible; then the
** program will do collision avoidance on this particular link-link pair.
** This table is clearly symmetric - so the program will only consider the
** upper right side of the diagonal (diagonal elements are of course FALSE).
*/
BOOLEAN l_l_check[DOF+1][DOF+1] =
 {{FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  TRUE,  FALSE, TRUE },
```

```
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  TRUE,  FALSE, TRUE },
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE, FALSE},
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, TRUE },
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, TRUE },
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,  FALSE, FALSE },
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE},
  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE}};
/*  0     1     2     3     4     5     6     7     8     9   */
#endif
```

NASA Langley 8-DOF Robot Definition (/home/tseng/Langley/pathplan/ctos/robot.def file):

```
/* robot parameters for Langley's 6 DOF arm plus 2 DOF platform  */
/* ============================================================ */

/* This file contains all parameters of the robot used in the assembly
** task. It's #included into robot.c.
** robot.c is general in the sense that all information about a specific
** robot is in robot.def.
** robot.c assumes a single chain robot with prismatic or revolute joints
** and a minimum of 6 DOF.
**--------------------------------------------------------------------*/

#ifndef ROBOT_CODE  /* this stuff included in libraray, .h files */

/* degrees of freedom: */
#define DOF 8
/* #define DOF 6*/
#define LANGLEY

#else   /* this stuff only for robot.c */

/* single arm system, as opposed to dual arms, used for gsm slot update size */
#define SINGLE_ARM

/* robot is not symmetric due to gripper pneumatic cords */
#define SYMMETRIC_ROBOT

/*  joint types. First entry is joint closest to the base. PRISM or REV. */
static int j_type[DOF] = {PRISM, PRISM, REV, REV, REV, REV, REV, REV};

/*  Arrays for a, d, alpha and theta.
** Kinematic parameters in modified Denavit Hartenberg form. (ie, DH except
** that the order is: rot(x), trans(x), trans(z), rot(z), to go from frame i-1
** to frame i. (start at frame 0).  Thus, frame i is moved by joint i,
** and link i is connected to link i. (ie. much nicer than DH) ).
** Units:   meters for a and d, degrees for alpha and theta.
** Indices: a[0] = a0   alpha[0] = alpha0   d[0] = d1   theta[0] = theta1
** In a revolute joint theta is the variable in q, whereas in a prismatic
** joint d is the variable in the joint vector q. Enter a 'Q' if the
** corresponding value is part of the q vector.
*/

static float alpha[DOF] = {-90., -90., 90., -90.,   0.0,  -90., 90., -90.};
static float     a[DOF] = { 0.0, 0.0,  0.0,  0.0,  .4394,   0.0, 0.0, 0.0};
static float     d[DOF] = {  Q,   Q, .6833, 0.0, -0.3048, 1.0338, 0.0, 0.0};
static float theta[DOF] = {-90., 90.,  Q  ,  Q ,     Q  ,   Q  ,  Q ,  Q };


/* end effector matrix.
** This homogeneous matrix describes the transformation from the last link
** to the gripper. In this case this is a simple translation along the
** z-axis.
** IMPORTANT! The vectors in the 3x3 matrix in the constant definition below
** ---------- are COLUMN vectors, even though they look like row vectors!
*/
static H_Matrix trE = {{{0.0, 1.0, 0.0},
                 {-1.0, 0.0, 0.0},
                       {0.0, 0.0, 1.0}}, {0.0, 0.0, 0.400}};


/* Joint value ranges for left arm. First entries are closer to the base.
```

```
** Units are meters for prismatic and degrees for revolute joints.
*/

static float ql_min[DOF] =
  {-3.0734, -5.373, -150., -240., -240., -720., -90., -720. };
static float ql_max[DOF] =
  { 2.8448,    0.0,  150.,   60.,   60.,  720.,  90.,  720. };

/* Joint ranges for right arm */
/* currently non-existent, bogus numbers */
static float qr_min[DOF] =
  {333. ,333., -45.,-250.,-225.,-45.,-110.,-100.};
static float qr_max[DOF] =
  {222. ,222., 45.,  70.,  45.,225., 170., 100.};


/* Joint weights used for solving the Jacobian equation. The values of this
** vector define a diagonal matrix Q with the elements of 'joint_weight' on
** the diagonal. The (in general redundant) Jacobian equation is solved such
** that q'Qq is minimized. (q' is the transpose of q).
*/
static joint_weight[DOF] = {120.0,48.0, 6.0, 6.0, 2.0, 1.0, 2.0, 1.0};


/* constants for definition of picture and model
*/

#define RR2            0.7
#define RX2            2.7
#define RX2b          -2.85
#define RY2            .995

#define RR3            0.20  /* extra leeway */
#define RZ3            0.0
#define RZ3b          -0.50  /* fudge don't want collision with 'cart' plane */

#define RR4            0.19
#define RX4           -0.21
#define RX4b           0.249
#define RZ4           -0.089

#define RR5            0.10
#define RY5           -0.12
#define RY5b           1.033

#define RR7            0.05
#define RY7            0.0
#define RY7b           0.20

#define RR8            0.05
#define RY8            0.93
#define RY8b          -0.93
#define RZ8            0.23
#define RZ8b        0.35


/* 'New_Link_Model' assumes a cylindrical link model. The first 6 parameters
** are the cylinder's endpoints, the 7th is the cylinder's radius and the
** last one is the link number.
** 'New_Link_Trian_Model' assumes a triangular plane segment model.  9 endpts,
** the radius, and the link number are the parameters.
*/
static void Get_Link_Models ()
{

  (void)New_Link_Model (RX2, RY2, 0.0,  RX2b, RY2, 0.0,   RR2, 2);
```

```
  New_Link_Model (0.0, 0.0, RZ3,  0.0,  0.0, RZ3b,  RR3, 3);
  New_Link_Model (RX4, 0.0, RZ4,  RX4b, 0.0, RZ4,   RR4, 4);
  New_Link_Model (0.0, RY5, 0.0,  0.0, RY5b, 0.0,   RR5, 5);
  New_Link_Model (0.0, RY7, 0.0,  0.0, RY7b, 0.0,   RR5, 7);
  (void)New_Link_Trian_Model (0.0,RY8,RZ8,  0.0,RY8b,RZ8, 0.0,0.0,RZ8b, RR8, 8);

}


/* The definition of the robot's picture on the screen. The picture is
** a wire frame - every line of this frame is defined here. The six
** first parameters of the 'Link_Line' procedure are  the two endpoints
** of the line in the local frame. Unit: meters.
** The last parameter is the link number.
*/
static void Get_Link_Pictures ()
{
/*
No SUNcore picture defined.
*/
}


/* The following information deals with collision avoidance between links.
** It's a table with dimension (DOF+1)*(DOF+1). Insert a TRUE if a
** collision between the column and the row link is possible; then the
** program will do collision avoidance on this particular link-link pair.
** This table is clearly symmetric - so the program will only consider the
** upper right side of the diagonal (diagonal elements are of course FALSE).
*/

BOOLEAN l_l_check[DOF+1][DOF+1] =
 {{FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },

  {FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE, FALSE, FALSE,  TRUE },

  {FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE, FALSE, FALSE,  TRUE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE },

  {FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE }};

/*    0      1      2      3      4      5      6      7      8     */

$endif
```

APPENDIX E
Planar Model

Our triangular planar model is defined by three vectors and a radius. It models the volume of space swept by a sphere of radius r over the entire area of a triangle defined by the three points **a**, **b**, and **c**, see Figure E.1. It can also be defined mathematically as the set of points S where S is defined as follows.

$$S(\mathbf{a},\mathbf{b},\mathbf{c},r) = \{\mathbf{p} \mid \mathbf{p} = \mathbf{a} + t_1\mathbf{x} + t_2\mathbf{y} + \mathbf{R}, \quad \forall t_1, t_2 \in [0..1], \forall |\mathbf{R}| \leq r\}$$

where

$$\mathbf{x} = \mathbf{b} - \mathbf{a}$$

and

$$\mathbf{y} = \mathbf{c} - \mathbf{a}$$

Adding this planar model, henceforth called a "triangle", to the planner's line segment, called "segment", requires us to make distance calculations for two more cases: distance from segment to triangle and distance from triangle to triangle. Munger has already calculated the distance from line to line, so we will use his work without repeating the derivation (see Munger [1] p. 7-13).

E.1 Distance from Segment to Triangle

We shall develop this calculation in the form of an algorithm outline. Words starting with capital letters are keywords (functions, variables).

Segment to Triangle Distance Algorithm:

1. Project the End Points of the Segment onto the plane of the Triangle; calculate the Distances from the end points to the plane, and calculate the Half-Planes

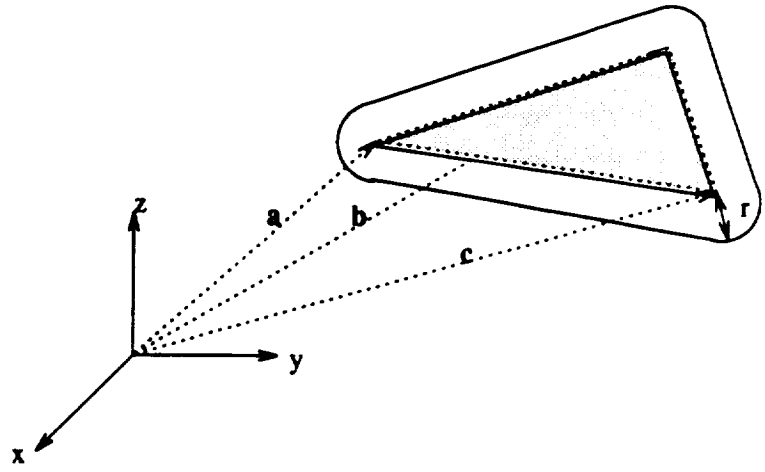


Figure E.1: Triangular Planar Model

which each end point is in.

2. if both end points are in the same Half-Plane then goto Step #3 else Find the Intersection of the Segment with the plane defined by the Triangle.

   (a) if the Intersection is In-The-Triangle then Report a collision.

   (b) else calculate the distance from the Segment to each of the three segments constructible from the Triangle's three verticies and Report the shortest distance, closest points, and vector.

3. if the both end points' projections are In-The-Triangle, then the closer of the two endpoints and its projection are the closest points on the Segment and the Triangle. Report these points, their distance, and the vector between them.

4. else calculate the distance from the Segment to each of the three segments constructible from the Triangle's three verticies and Report the shortest distance, closest points, and vector.

—end of algorithm—

There are three key procedures which are worth discussion, details can be found in module `vector.c`. The first procedure is the Projection of the endpoints onto the plane of the Triangle. This is done by finding the normal to the plane (again let **a**, **b**, and **c** be the points of the triangle):

$$\mathbf{normal} = (\mathbf{a} - \mathbf{b}) \times (\mathbf{a} - \mathbf{c})$$

Then the matrix

$$\mathbf{T} = (\mathbf{a} - \mathbf{b} \mid \mathbf{a} - \mathbf{c} \mid \mathbf{normal})^{-1}$$

is the transform from the world frame to the coordinate frame of the triangle plane **ab** and **ac**, and the **normal** to the plane. Hence the product

$$\mathbf{T}\,d$$

where

$$\mathbf{d} = \mathbf{a} - \mathbf{p}$$

and **p** is an endpoint of the segment, transforms the point **p** into the plane's coordinate system. From there the world coordinates can easily be recovered.

The second important procedure, In-The-Plane, determines if a co-planar point **p** is inside the triangle **abc**. See Figure E.2, if **ap** is in the shaded region then its cross products with **ab** and **ac** will have the same sign, ie. their dot product will be positive. If **ap** is in the *interior* of the vectors **ab** and **ac** then the cross products $\mathbf{ab} \times \mathbf{ap}$ and $\mathbf{ab} \times \mathbf{ac}$ will have the same sign (dot product). Thus we have a test for one vector **ap**'s inclusion between two other vectors **ab** and **ac**. Finally, if **p** is between two pairs of vectors of a triangle (**ab** and **ac** *and* **bc** and **ba**), then it is interior to the triangle.

The last procedure, calculating the distance from a segment to another segment has been described by Munger in his paper [1].

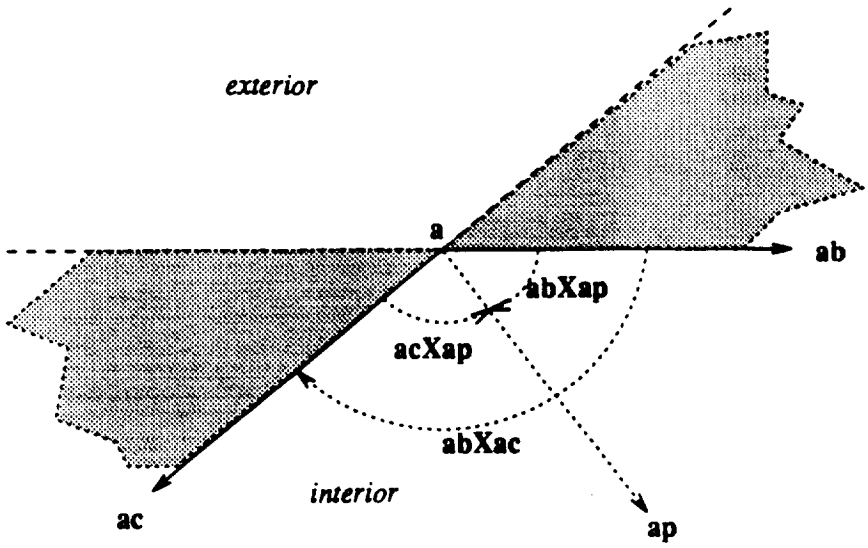


Figure E.2: In-The-Triangle()

E.2 Distance from Triangle to Triangle

This calculation draws heavily from the procedures already described. The following algorithm calls the *Segment to Triangle Distance Algorithm* five times, therefore this algorithm is at best five times slower than the previous algorithm, which in turn is slower than the segment to segment distance calculation. In sum, adding the triangle model has slowed down distance calculations considerably.

Triangle to Triangle Distance Algorithm:

1. Let the planes, p1 and p2, be defined as points (**a**,**b**,**c**) and (**d**,**e**,**f**), with radii, $r1$ and $r2$.

2. Find the closest segment–plane pair among the following pairs: **ab**–p2, **ac**–p2, **bc**–p2, **de**–p1, and **df**–p1, by calling the *Segment to Triangle Distance Algorithm* five times. Name the closest points **point1** and **point2**, and name the distance $d1$.

3. Return the closest points **point1** and **point2**. Return the closest distance $d1 - r1 - r2$.

—end of algorithm—

APPENDIX F

Header File Listings

Module **alg** (linear algebra):

```
   Data types
   ----------

  typedef struct var
  {
    double *e;
    int r, c;
    int size;
  } VAR;
--------------------
A variable (matrix, vector, or scalar). 'r' rows, 'c' columns, and 'size'
number of elements (r*c).  'e' is a pointer to a linear array of 'size'
doubles.  The order is left to right, then top to bottom.



   Procedure description
   ---------------------

   void Init_Vars ()
--------------------
This procedure initializes the "ALG" module and must be called before doing
anything else.


   void Exit_Vars ()
--------------------
Can be called in order to free all memory space used by the module. No function
should be called after 'Exit_Vars'.


   VAR *New_Var ()
------------------
Assigns a pointer to an initialized matrix to a pointer variable. Every pointer
variable must be initialized this way before using it.
After initialization the variable is empty (a 0x0 matrix).


   BOOLEAN Put (a, b)
        VAR *a;
        VAR *b;
--------------------
Puts the value of 'a' which can be a variable or an expression, into 'b'.
This function must be used for every assignment operation!

   VAR *V0 (r, c)
        int r;
        int c;
-----------------
Returns an r*c matrix with all elements equal to zero. Typed: "vee-oh".

   VAR *VOnes (r, c)
        int r;
        int c;
```

```
----------------
Returns an r*c matrix with all elements equal to one.

   VAR *VI (dim)
        int dim;
---------------
Returns a dim*dim unit matrix.

   VAR *Vuser (r, c, v1, v2, v3, ...)
        int r;
        int c;
        'r*c' doubles;
-----------------------------------
Returns an r*c matrix with user defined contents. The doubles in the parameter
list are filled in the matrix from left to right and from top to bottom.
The user must supply r*c doubles in the parameter list.
    Example:
Vuser (2, 3, 1.0, 2.0, 3.0,                                    [ 1  2  3 ]
            5.0, 4.0, 3.0);          creates                   [ 5  4  3 ]


   VAR *Vscmult (v, s)
        VAR *v;
        double s;
----------------------
Returns 'v' multiplied elementwise with 's'.
    Example:
[ 1  2  3 ]   multiplied with 2.0 is   [ 2  4  6 ].

   double Vector_Norm (v)
        VAR *v;
-----------------------
Returns the "two" norm (length) of variable v. 'v' must be a vector, i. e.
'v' must have either one row or one column.


   VAR *Vadd (a, b)
        VAR *a;
        VAR *b;
-------------------
Returns the elementwise addition of variables 'a' and 'b'. 'a' and 'b' must
have the same dimensions.

   VAR *Vsub (a, b)
        VAR *a;
        VAR *b;
-------------------
Returns the elementwise subtraction of variables 'a' and 'b'. 'a' and 'b' must
have the same dimensions.


   VAR *Vmult (a, b)
        VAR *a;
        VAR *b;
--------------------
Returns the product of variables 'a' and 'b'. The number of rows of 'a' must
equal the number of columns of 'b'.


   double Vdot (a, b)
        VAR *a;
VAR *b;
-------------------
Returns the dot product of column variables 'a' and 'b'.
(Compatibility required)
```

```
   VAR *Vtranspose (v)
        VAR *v;
----------------------
Returns the transpose of variable v.


   VAR *Vsolve (a, b, q, success)
        VAR *a;
        VAR *b;
        VAR *q;
        BOOLEAN *success;
-----------------------------
Returns the solution of the linear system a*x=b. 'a' and 'b' must have the
same number of rows. If 'a' is square, 'Vsolve' returns the solution, if
one exists (if 'a' is nonsingular) or the solution to the underdetermined
reduced set of equations (if 'a' is singular). If 'a' has more rows than
columns (system is overdetermined), then 'Vsolve' returns the least square
approximation. If 'a' has less rows than columns (system is underdetermined),
then the x minimizing x'Qx (' denotes the transpose) is returned. Q is the
INVERSE of a diagonal matrix with the elements of vector 'q' on the diagonal,
thus 'q' and 'x' have the same dimension. If 'q' is NULL, then Q=I
(unity matrix) is assumed and thus the minimum norm solution is returned.
If a solution was found, 'success' is set to TRUE, otherwise to FALSE.


   VAR *Vinv (a)
        VAR *a;
----------------
Returns the inverse of variable 'a'. 'a' must be squared.


   int Nb_Cols (v)
        VAR *v;
------------------
Returns the number of columns of variable 'v'.


   int Nb_Rows (v)
        VAR *v;
------------------
Returns the number of rows of variable 'v'.


   BOOLEAN Fill_Var (v, r, c, num, v1, v2, v3, ...)
        VAR *v;
        int r;
        int c;
        int num;
        'num' doubles;
-------------------------------------------------
This procedure is used to fill part of matrix 'v' with user defined values.
It starts filling at the element at row 'r' and column 'c' and fills up
from left to right and from top to bottom. It writes 'num' values into
the matrix. It is the user's responsibility to supply 'num' doubles after
the 'num' parameter. If the matrix would overflow over the bottom right
corner, an error occurs and no values are written at all.


   double Read_El (v, r, c)
        VAR *v;
        int r;
        int c;
--------------------------
Returns the element at row 'r' and column 'c' in variable 'v'.  Elements
start at r = 0 and c = 0.
```

```
   BOOLEAN Write_El (v, r, c, val)
        VAR *v;
        int r;
        int c;
        double val;
---------------------------------
```

Writes 'val' to the element at row 'r' and column 'c' in variable 'v'. The first element is (r,c) = (0,0).

```
   VAR *Vcut (src, r_src, c_src, r_size, c_size)
        VAR *src;
        int r_src;
        int c_src;
        int r_size;
        int c_size;
-----------------------------------------------
```

Returns a piece of variable 'src'. The top left corner of this piece is the element at row 'r_src' and column 'c_src' in variable 'src'. The piece has 'r_size' rows and 'c_size' columns. An error occurs, if the specified piece is not part of matrix 'src'. If r_size or s_size are zero then the rest of the elements in the row or column, respectively, are taken.

```
   BOOLEAN Paste (src, dest, r_dest, c_dest)
        VAR *src;
        VAR *dest;
       int r_dest;
        int c_dest;
-------------------------------------------
```

Pastes variable 'src' into variable 'dest'. 'src's top left corner goes to row 'r_dest' and column 'c_dest' in variable 'dest'. An error occurs if there is not enough room in 'dest' to complete the operation.

```
   BOOLEAN Swap_Rows (v, r1, r2)
        VAR *v;
        int r1;
        int r2;
---------------------------------
```

Rows 'r1 and 'r2' in variable 'v' are exchanged.

```
   BOOLEAN Swap_Cols (v, c1, c2)
        VAR *v;
        int c1;
        int c2;
---------------------------------
```

Columns 'c1 and 'c2' in variable 'v' are exchanged.

```
   BOOLEAN Print_Var (v)
        VAR *v;
------------------------
```

Variable 'v' is printed to the screen. 'Print_Var' doesn't care about the screen size, so large matrices may be hard to read.

```
   void Kill_Var (v)
        VAR *v;
--------------------
```

The memory space of variable 'v' is freed. Pointer 'v' is invalid after 'Kill_Var'.

Module **env** (environment):

```
   Procedure description
   ---------------------

   void Init_Env (source)
        LIST *source;
-------------------------
```

Reads the locations of all struts that are present in the environment at initialization time and the length of the struts in use. Both input file and CIRSSE interface are read. Then tetrahedra are extracted and intermediate steps are generated.

```
   double Get_Strut_Length ()
-----------------------------
```

Returns the strut length.

```
   BOOLEAN Symmetric ()
-----------------------
```

Returns TRUE if the struts are symmetric in the sense that the endpoints can be exchanged for assembly. If this is not the case, FALSE is returned.

```
   BOOLEAN Get_Tetra_Pos (xt, yt, zt, nb, p1, p2)
        int xt;
        int yt;
        int zt;
        int nb;
        Vector *p1;
        Vector *p2;
-------------------------------------------------
```

Transforms a strut position given in tetrahedron coordinates to cartesian coordinates of its endpoints p1 and p2. (xt, yt, zt) denote a tetrahedron in the structure and 'nb' denotes the number of the strut in this tetrahedron (1..6).

```
   MODEL *Get_First_Strut_Model (lp)
        LIST_EL **lp;

   MODEL *Get_Next_Strut_Model (lp)
        LIST_EL **lp;

   MODEL *Get_First_Thing_Model (lp)
        LIST_EL **lp;

   MODEL *Get_Next_Thing_Model (lp)
        LIST_EL **lp;

   MODEL *Get_First_Inter_Step_Model (lp)
        LIST_EL **lp;

   MODEL *Get_Next_Inter_Step_Model (lp)
        LIST_EL **lp;
---------------------------------------
```

These procedures are used to read the list of strut models, thing models (struts and planes) and intermediate step models and are equivalent to the standard list readout procedures described in the list module.

```
   BOOLEAN Add_Strut (p1, p2, strutId)
```

```
        Vector p1;
        Vector p2;
        int    strutId;
-----------------------------
Adds a strut to the environment. Its endpoints are at 'p1' and 'p2'. Its id
is 'strutId' (for gsm). Intermediate steps are deleted, tetrahedra reextracted
and intermediate steps recomputed.

   BOOLEAN Remove_Strut (p1, p2, rad, strutId)
        Vector *p1;
        Vector *p2;
        double *rad;
        int    *strutId;
-----------------------------------
Removes the strut closest to an imaginary strut with endpoints at '*p1' and
'*p2'. The closest strut is the strut whose center is closest to the center
of the imaginary strut between '*p1' and '*p2'. This strut's endpoints,
radius, and GSM identity number is returned in '*p1', '*p2', 'rad' and
'strutId', respectively. It is VERY IMPORTANT that Remove_Strut is called
before Grasp_Part so that Grasp_Part will inform gsm correctly.
FALSE is returned if there is no strut in the environment.
Intermediate steps are deleted, tetrahedra reextracted and intermediate steps
recomputed.


   MODEL *Get_Closest_Strut_Model (p1, p2, dmin)
        Vector p1;
        Vector p2;
        double *dmin;
-----------------------------------------------
Returns the model of the strut from strutlist closest to the position
defined by 'd1' and 'd2'. The distance is returned in 'dmin'. If there
is no strut in the strutlist, 'dmin' takes a negative value, otherwise
'dmin' is the distance between the centers of the struts.
```

Module **global** :

```
   Procedure description
   ---------------------

   void Warning (procname, msg)
        char *procname;
        char *msg;
------------------------------
Prints the calling procedure's name ('procname') and a warning message ('msg')
and returns.


   void Error (procname, msg)
        char *procname;
        char *msg;
----------------------------
Prints the calling procedure's name ('procname') and an error message ('msg')
and returns.


   void Fatal (procname, msg)
        char *procname;
        char *msg;
----------------------------
Prints the calling procedure's name ('procname') and an error message ('msg')
and exits the program with exit code 1.


   void Fatal_Malloc (procname)
        char *procname;
------------------------------
Prints the calling procedure's name ('procname') and the standard error
message "memory allocation failed" and exits the program.
This procedure is provided for convenience since every memory allocation
must be checked for failure.
It also returns with exit code 1.


   double Atan2 (x, y)
        double x, y;
--------------------
Like the built in math function atan2, but Atan2 (0.0, 0.0) = 0.0
```

Module **gpath** (global path planner algorithm):

```
   Procedure description
   ---------------------

   void Init_Path ()
--------------------
Must be called once before calling 'Find_Path' for initialization.


   LIST *Find_Path (path, p1, p2, dir, deprt, appr,  attempt_num)
        LIST *path1;
Vector p1;
Vector p2;
Vector dir;
int    attempt_num;
---------------------------------------
Plans a path leading from the current joint vector found in the robot module
to a position defined using the endpoints of the goal strut 'p1 and 'p2'
and the direction from which the goal is to be approached 'dir'.
The path is returned in list 'path'. The elements are joint vectors (type VAR).
If attempt_num is not 1 then planning starts at last global graph state
with the black lists and isteplist and "rotation insertion number" saved
from the previous call to Find_Path.
```

Module **graph** (graph theory):

```
   Data types
   ----------

  typedef struct graph
  {
    LIST   *nodelist;       /* the nodes (vertices) of the graph   */
    double  (*Weight) ();    /* function returning an edge's weight */
    BOOLEAN directed;       /* TRUE if graph is directed           */
  } GRAPH;


   Procedure description
   ---------------------


   GRAPH *New_Graph (Weight, directed)
        double (*Weight) ();
        BOOLEAN directed;
-------------------------------------
Creates a new graph data structure. The user must provide the function 'Weight'
which returns the weight of an edge to the graph module. It is declared as
follows:
   double Weight (node1, node2, edge)
        char *node1, node2, edge;
If the graph is directed, the module expects the weight of the edge going from
'node1' to 'node2'. If parameter 'directed' is TRUE, then a directed graph is
created.


   void Connect (graph, node1, node2, edge)
        GRAPH *graph;
        char  *node1;
        char  *node2;
        char  *edge;
-----------------------------------------
'node1' and 'node2' are connected by 'edge'. Any graph structure can be build
by just using this one procedure. If one of the nodes has been used in a
previous call of 'Connect', then the new edge is added to it, otherwise a
new node is created automatically. In a directed graph an edge pointing from
'node1' to 'node2' is created.


   void Connect_All (graph, nodelist, Get_Edge)
        GRAPH *graph;
        LIST  *nodelist;
        char  *(*Get_Edge) ();
-----------------------------------------
This procedure is useful for creating graphs in which every node is connected
to every other node. 'nodelist' contains the nodes of the graph and 'Get_Edge'
is a user provided procedure that is declared as follows:
   char *Get_Edge (node1, node2)
        char *node1, *node2;
This function must return the data associated to the edge between 'node1' and
'node2'. This can be a NULL pointer which means that this edge doesn't have
an equivalent data structure in the user's module. In fact, the 'Get_Edge'
parameter can be a NULL pointer, too. This is the case when the edges in the
graph module generally don't have an equivalent data structure in the user's
module. In a directed graph every pair of nodes will receive two edges
pointing in opposite directions.
```

```
   void Connect_All_Cond (graph, nodelist, Get_Edge, Condition)
        GRAPH *graph;
        LIST  *nodelist;
        char  *(*Get_Edge) ();
        BOOLEAN (*Condition) ();
-------------------------------------------------------------
This procedure works like 'Connect_Cond', the only difference is the
additional parameter 'Condition' which must be declared as follows:
   BOOLEAN Condition (node1, node2)
        char *node1, *node2;
Before creation of an edge 'Connect_All_Cond' will call this function. If it
returns TRUE, the edge is created, otherwise it isn't. This feature is useful
to set up visibility graphs: 'Condition' must return TRUE if 'node1' is visible
from 'node2' and FALSE otherwise.


   BOOLEAN Disconnect (graph, node1, node2, edge)
        GRAPH *graph;
        char  *node1;
        char  *node2;
        char  *edge;
-----------------------------------------------
This procedure is used to remove a single edge from the graph. 'edge' between
'node1' and 'node2' is removed. TRUE is returned if this edge existed, FALSE
otherwise.


   void Disconnect_All (graph)
        GRAPH *graph;
------------------------------
This procedure removes all edges from the graph. The nodes remain in the
graph!


   LIST *A_Star (graph, Estimate, start, goal, edge_path)
        GRAPH *graph;
        double (*Estimate) ();
        char  *start;
        char  *goal;
        LIST  **edge_path;
-----------------------------------------------------------
The A-Star algorithm tries to find the optimal path from 'start' to 'goal'.
Optimal means minimal sum of edge weights along the path.
'Estimate' is a pointer to a user provided function:
   double Estimate (node)
        char *node;
It must return an estimate of the cost to go from 'node' to 'goal'. If
this estimate is always lower than the actual cost, A-Star will find the
optimal path.
If a path exists, A-Star will find it and return a list of the nodes it
passed. In parameter 'edge_path' it returns a list of the edges it went
through. The two lists have the same length. The first edge is the edge
between the first and the second node, so the last entry in the edge list
is always a NULL. If no path exists, NULL is returned.


   void Kill_Graph (graph)
        GRAPH *graph;
-------------------------
Deletes the nodes, edges and the graph data structure making 'graph'
invalid. The user's data for the nodes and edges are of course left intact.
```

Module **graphics** (SUNcore and X Windows interface):

```
   Constants
   ---------

#define BLACK  0
#define WHITE  1
#define RED    2
#define YELLOW 3
#define BLUE   4
#define GREEN  5
#define GRAY   6

#define X_GRAPHICS     2
#define SUN_GRAPHICS   1
#define NO_GRAPHICS    0


   Data types
   ----------

typedef struct line
{
  Vector p1;
  Vector p2;
  int    color;
  int    style;
} LINE;
--------------------
LINE is a line on the SUNcore screen. 'color' is defined above. 'style' is
either SOLID, DASHED, or DOTTED.

typedef struct character
{
  char   c;
  Vector pos;
  int    color;
} CHAR;
------------------------
CHAR represents a SUNcore character on the screen.


typedef struct seg
{
  LIST   *linelist;
  LIST   *charlist;
  int     segnum;
  BOOLEAN active;
} SEG;
------------------
SEG represents a segment that contains a number of lines and characters.  The
lines (LINE) are stored in 'linelist' and the characters (CHAR) in 'charlist'.
'segnum' is the SUNcore segment number. 'active' is TRUE if the segment is
nonempty and must be included in updates and rotations.


   Procedure description
   ---------------------

   void Init_Graphics (source)
        LIST *source;
----------------------------
Initializes SUNcore in the current window and displays a coordinate system.
```

```
Parameter 'source' is a list of expressions from the parser. If expression
'B&W' is found, the graphics are displayed in black and white, even on a
color screen. This can be useful for screendumps.
If a 'ZOOM (x)' expression is found, then the display is enlarged or
shrunk according to x.


   void Init_Text (source)
LIST *source;
-----------------------------
Initializes a new SunVIEW window so that stdout and stdin is mapped to i
Parameter 'source' is a list of expressions from the parser.  If expression
'Diagnostics' is found then the diagnositics window is displayed.


   BOOLEAN Graphics_Active ()
-----------------------------
Returns TRUE if SUNcore has been successfully initialized, FALSE otherwise.


   void Exit_Graphics ()
------------------------
Should be called before exiting the program.

   void Spin_Graphics ()
------------------------
Enables the user to rotate the picture around the vertical or the horizontal
screen axis by moving the mouse horizontally or vertically respectively.
This procedure ends in the current orientation when the user presses the
middle mouse button.

   SEG *New_Segment ()
----------------------
Returns a new segment.

   void Kill_Segment (seg)
        SEG *seg;
--------------------------
Kills segment 'seg'.

   void Update_Segment (seg)
        SEG *seg;
----------------------------
Redraws segment 'seg'. This is needed when there are changes in certain lines
or characters in the segment that are not yet reflected on the screen.

   void Update_All_Segments ()
------------------------------
Redraws all segments at once.

   void No_Update ()
--------------------
After this procedure is called, the screen is not updated when primitives are
inserted in or deleted from a segment. This is useful when deleting many
primitives at once to avoid repeated reconstruction of the segment.
Updating is turned back on by calling 'Update_Segment' on any segment.

   LINE *New_Line (color, style)
        int color;
        int style;
--------------------------------
Returns a new line with given color and style (SOLID, DOTTED, DASHED).

   void Kill_Line (l)
        LINE *l;
--------------------
```

```
Kills line 'l'.

   void Set_Line_Pos (l, p1, p2)
        LINE  *l;
        Vector p1;
        Vector p2;
-------------------------------
Changes line 'l's position.

   void Get_Line_Pos (l, p1, p2)
        LINE  *l;
        Vector p1;
        Vector p2;
-------------------------------
Returns line'l's position.

   void Insert_Line (seg, l)
        SEG  *seg;
        LINE *l;
---------------------------
Inserts line 'l' into segment 'seg' and displays it immediately, if there
was no previous 'No_Update'.

   BOOLEAN Delete_Line (seg, l)
        SEG  *seg;
        LINE *l;
-------------------------------
Deletes line 'l' from segment 'seg' and reflects the change immediately, if
there was no previous 'No_Update'.

   CHAR *New_Char ()
--------------------
Returns a new character.

   void Kill_Char (c)
        CHAR c;
---------------------
Kills character 'c'.

   void Change_Char ()
        CHAR  *c;
        Vector pos;
        int    color;
        char   ch;
---------------------
Changes position, color and letter of character 'c'.

   void Insert_Char ()
        SEG  *seg;
        CHAR *c;
---------------------
Inserts character 'c' into segment 'seg' and displays it immediately, if there
was no previous 'No_Update'.

   BOOLEAN Delete_Char (seg, c)
        SEG  *seg;
        CHAR *c;
-------------------------------
Deletes character 'c' from segment 'seg' and reflects the change immediately,
if there was no previous 'No_Update'.

   void Flush_Text ()
-----------------------------
Allows the SunVIEW system tty window to printf.  This is a necessary step
because SunCORE somehow disables the SunVIEW windowing environment...
```

```
   void More_Text ()
-------------------------------
If 'step_text' is TRUE then the text is Flushed and execution awaits any
mouse button.  Otherwise, the text is just Flushed
```

Module lpath (local path planner algorithm):

```
   Procedure description
   ---------------------


   void Init_LPath ()
--------------------
Must be called before the first call of 'Local_Path_Plan'.


   void Prep_Local_Path_Plan ()
------------------------------
Called before each call to 'Local_Path_Plan' so that goal tolerances are set.


   void Normalize_A_Vector (p1, p2, dir)
        Vector p1;
        Vector p2;
        Vector *dir;
---------------------------------------
Makes vector 'dir' length 1 and orthogonal to the line defined by 'p1' and
'p2'. 'dir' will remain in the plane defined by the line through 'p1' and
'p2' and the line along the old 'dir'.


   MODEL *Get_Min_Obstacle ()
----------------------------
returns a pointer to the model of the closest object to the robot.


   BOOLEAN Valid_Strut (p1, p2)
        Vector p1;
        Vector p2;
-----------------------------
Returns TRUE if 'p1' and 'p2' is a valid position for an intermediate step.


   BOOLEAN Local_Path_Plan (q, p1, p2, d, path, pict, seg, rot_normal, arm,
                            deprt, appr)
        VAR    *q;
        Vector  p1;
        Vector  p2;
        Vector  d;
        LIST   *path;
        LIST   *pict;
        SEG    *seg;
        BOOLEAN rot_normal;
ARM_TYPE arm;
double  deprt, appr;
-------------------------------------------------------------------------
Plans a path using a potential field method. The initial joint vector 'q'
is assumed and the path will lead the (real or imaginary) payload strut to
endpoint positions 'p1' and 'p2'. The goal will be approached in direction
'd'. The path will be returned in list 'path' which will contain a joint
vector (VAR) for each step. List 'pict' will contain the lines to display
the path and segment 'seg' will be used. If 'rot_normal' is TRUE, then the
angle less than 180 deg will be used to rotate the gripper from its start
to its goal orientation, which is normally better. If it is FALSE, the other
sense of rotation will be used. which involves an angle of rotation of more
than 180 deg.
```

```
   void New_Start_Dir (dir)
        Vector dir;
---------------------------
```

Must be called before 'Local_Path_Plan' if the path must leave the start position in a particular direction. This direction is AGAINST the vector 'dir', so 'dir' is normally the approach vector of the robot's gripper in start position.

Module lst (list data structure):

Data types

```
typedef struct list_element
{
  struct list_element *next;     /* pointer to the next list element */
  char                *data;     /* pointer to the data              */
} LIST_EL;
```

Elements of type LIST_EL form the chain of list elements. 'next' points to the next element in the list, 'data' points to the user data represented by this list element.

```
typedef struct list
{
  struct list_element *first, *last; /* pointers to beginning and end of list*/
  int                  length;       /* number of elements in the list       */
  char                **index;       /* pointer to index array               */
} LIST;
```

LIST is the main list data structure. 'first' and 'last' point to the first and the last element in the LIST_EL chain. 'length' stores the number of elements currently in the list. 'index' has a pointer to an array of user data pointers that allow fast random list access. If an index doesn't exist, 'index' is NULL.

Procedure description

```
LIST *New_List ()
```

Creates a new list (allocates and initializes a LIST data structure) and returns a pointer to it.

```
BOOLEAN Insert (lst, data)
     LIST *lst;
     char *data;

BOOLEAN Insert_As_First (lst, data)
     LIST *lst;
     char *data;
```

'Insert' and 'Insert_As_First' are the two procedures to build a list. 'Insert' adds the element 'data' at the end, 'Insert_As_First' at the beginning of the list. Existing indeces are destroyed by both procedures.

```
BOOLEAN Delete (lst, data)
     LIST *lst;
     char *data;
```

Deletes element 'data' from the list 'lst'. Returns TRUE if 'data' was found in the list, FALSE otherwise. An existing index is destroyed.

```
BOOLEAN Is_In_List (lst, data)
     LIST *lst;
```

```
        char *data;
--------------------------------
```

Returns TRUE if 'data' is found in 'lst', FALSE otherwise.

```
   char *Get_First (lst, current)
        LIST     *lst;
        LIST_EL **current;

   char *Get_Next (current)
        LIST_EL **current;
--------------------------------
```

'Get_First' and 'Get_Next' allow sequential access to the list. A procedure using these functions typically looks as follows:

```
void Sequential_Access_Example (lst)
     LIST *lst;
{
  LIST_EL *lp;
  DATA_ITEM *data;

  data = (DATA_ITEM *)Get_First (lst, &lp);
  while (lp)
    {
      Process_Data_Item (data);
      data = (DATA_ITEM *)Get_Next (&lp);
    }
}
```

In this example the list holds elements of type DATA_ITEM. Since both 'Get_First' and 'Get_Next' return pointers to type char, a type cast is necessary in most cases. The pointer variable 'lp' points to the current list element. It is initialized to the first list element by 'Get_First' and updated to the next element by 'Get_Next'. When the end of the list is reached, 'lp' is assigned NULL, so loop control can be done using 'lp'. If a program contains nested loops, it is important to declare a separate element pointer variable for every loop that goes through a list.
Note that 'lp' does NOT point to the data element of type DATA_ITEM, but to the list element of type LIST_EL that represents this data element!

```
   char *Get_Nth (lst, n)
        LIST *lst;
        int   n;
-------------------------
```

This function is used for random access. If an index exists, the n-th element is returned very quickly, otherwise the function steps through the list sequentially and thus takes a little longer if 'n' is large.
n=0 returns the first element. If 'n' is too large, NULL is returned.

```
   char *Get_This (current)
        LIST_EL **current;
-------------------------
```

returns the data element represented by the list element that 'current' points to. 'current' is left unchanged.

```
   char *Get_Last (lst)
        LIST *lst;
----------------------
```

Returns the last data element of list 'lst'.

```
   void Build_Index (lst)
        LIST *lst;
```

```
-------------------------
Creates an array of pointers to the data elements in the list. Once this index
exists, random accesses using function 'Get_Nth' become much faster. Any
function that changes the list will destroy the index automatically!


   BOOLEAN Append (lst, lst2)
        LIST *lst;
        LIST *lst2;
----------------------------
Appends the elements of 'lst2' to 'lst'. The list elements are duplicated in
this process, so changing 'lst2' after 'Append' has no effect on 'lst'.
TRUE is returned if 'Append' was successful, FALSE otherwise.
An existing index of 'lst' is destroyed automatically!


   int List_Length (lst)
        LIST *lst;
-----------------------
Returns the number of elements in the list 'lst'.


   void Empty_List (lst)
        LIST *lst;
-----------------------
Removes all list elements from list 'lst' leaving just the LIST data structure.
Any existing index is destroyed automatically.
Note that the data elements themselves are NOT affected in this process!


   void List_Apply_F (lst, Function)
        LIST *lst;
        void (*Function) ();
-----------------------------------
Applies the user defined function 'Function' to all elements of the list 'lst'.
This function must be declared as follows:

void User_Function (data)
     char *data;

'data' is the current data element in the list.


   void Kill_List (lst)
        LIST *lst;
----------------------
Removes all list elements and the LIST data structure itself, so 'lst' is
invalid after 'Kill_List'. Any existing index is of course deleted too.
Note that the data elements themselves are NOT affected in this process!
```

Module **model** (strut and plane obstacle definitions):

```
   Constants
   ---------

#define STRUT_TYPE 0
#define TRIAN_TYPE 1


   Data types
   ----------

typedef struct model
{
  int type;            /* type of model; triangle or strut */
  Vector p1, p2, p3;   /* vector rep (from world) of verticies of model */
  Vector  dir;         /* direction of approach to model (if used as subgoal)*/
  double r;            /* radius of swept sphere model */
  int id;              /* frame id number for gsm identification */
} MODEL;
------------  ------
MODEL represents either a swept sphere line segment or planar triangular
segment geometrical model. The  volume is the volume
that a sphere of radius 'r' sweeps when moving from point 'p1' to point 'p2'
on a straight line, or between the triangle defined by 'p1', 'p2',and 'p3'.


   Procedures
   ----------

   MODEL *New_Model ()
----------------------
Returns a new instance of a model.

   void Set_Model_Parameters (m, p1, p2, p3, r)
        MODEL *m;
        Vector p1;
        Vector p2;
        Vector p3;
        double  r;
-----------------------------------------
Changes all parameters of model 'm'.

   void Set_Model_Strut_Pos (m, p1, p2)
        MODEL *m;
        Vector p1;
        Vector p2;
--------------------------------
Changes the endpoints of model 'm' leaving its radius unaffected.

   void Set_Model_Radius (m, r)
        MODEL *m;
        double r;
-----------------------------
Changes model 'm's radius leaving its endpoints unaffected.

   void Set_Model_Dir (m, dir)
        MODEL *m;
        double dir;
-----------------------------
Changes model 'm's direction leaving its e    oints unaffected.

   void Set_Model_Id (m, id)
```

```
        MODEL *m;
        int id;
------------------------------
Sets model 'm's frame id.

   double Model_Distance (m1, m2, p1, p2)
        MODEL  *m1;
        MODEL  *m2;
        Vector *p1;
        Vector *p2;
---------------------------------------
Computes the shortest distance between models 'm1' and 'm2'. It returns the
distance and the two closest points on the line segments inside the swept
sphere cylinder or triangle (parameters 'p1', 'p2').

   void Get_Model_Strut_Pos (m, p1, p2)
        MODEL  *m;
        Vector *p1;
        Vector *p2;
---------------------------------
Returns the model endpoints in 'p1' and 'p2'.

   void Get_Model_Radius (m, r)
        MODEL  *m;
        double  *r;
---------------------------------
Returns the model radius in 'r'.

   void Get_Model_Dir (m, dir)
        MODEL  *m;
        double  *dir;
---------------------------------
Returns the model direction in 'dir'.

   void Get_Model_Id (m, id)
        MODEL *m;
        int *id;
------------------------------
Returns model 'm's frame id.

   void Swap_Strut_Endpoints (m)
        MODEL *m;
------------------------------
Exchanges the model's endpoints.

   void Kill_Model (m)
        MODEL *m;
---------------------
Kills model 'm' (frees its memory space).
```

Module **parser** :

```
   Data types
   ----------

   typedef struct expression
   {
     char *keyword;
     LIST *par_list;
   } EXP;
----------------------------
EXP represents expressions in an input file. An expression consists of a
keyword and optionally a number of parameters in parentheses, separated
by commas. Examples:
  keyword
  keyword (parameter)
  keyword (parameter1, parameter2, parameter3, parameter4)
'keyword' points to the keyword string converted to uppercase. 'par_list'
contains a list of strings that represent the parameters. They are also
converted to uppercase.


   Procedure
   ---------

   LIST *New_Source (fname)
        char fname[];
---------------------------
This procedure opens the text file 'fname' and parses it according to the
module's grammar. If the file doesn't exist or there are syntax errors, then
a NULL pointer is returned, otherwise a list of expressions as found in the
file is returned.
It is possible to set 'fname' to NULL. In this case, an empty expression list
will be returned without error message.

   EXP *Get_First_Exp (source, lp, keyword)
        LIST *source;
        LIST_EL **lp;
        char *keyword;

   EXP *Get_Next_Exp (lp, keyword)
        LIST_EL **lp;
        char *keyword;
-----------------------------------------
'Get_First_Exp' and 'Get_Next_Exp' are very similar to 'Get_First' and
'Get_Next' in the list module. In fact, if 'keyword' is NULL, they are
equivalent. If 'keyword' is a string, then 'Get_First_Exp' will return
the first expression with this keyword and 'Get_Next_Exp' will return the
next occurrence of an expression with this keyword from the current point
in the list. The matching is case insensitive. The readout procedures are
compatible to the list module in the sense that a part of the expression
list can be read with the procedures in the list module and then a particular
keyword can be searched from that point using 'Get_Next_Exp'. If no matching
keyword is found, NULL is returned and 'lp' is set to NULL.

   char *Get_Keyword (exp)
        EXP *exp;
-------------------------
Returns the uppercase keyword string of expression 'exp'.

   int Nb_Par (exp)
        EXP *exp;
-----------------
```

```
Returns the number of parameters of expression 'exp'.

   char *Get_Par (exp, nb)
        EXP *exp;
        int nb;
-------------------------
Returns uppercase parameter string number 'nb' in expression 'exp'. The first
parameter has 'nb' = 0. If 'nb' is too large, NULL is returned.

   BOOLEAN Get_Double (par, v)
        char  *par;
        double *v;
-----------------------------
Converts string 'par' into double 'v'. Returns TRUE if successful,
FALSE otherwise.

   BOOLEAN Get_Int (par, v)
        char *par;
        int  *v;
--------------------------
Converts string 'par' into int 'v'. Returns TRUE if successful,
FALSE otherwise.

   void Kill_Source (source)
        LIST *source;
---------------------------
Kills the source list 'source'. Kills all expressions in it and the
list itself.
```

Module **ppLib** (path planner CTOS message library):

```
   Constants
   ---------
$define  MSG_PP                 (MSG_USER+300)
$define  MSG_PP_INITIALIZE      (MSG_PP+ 1)
$define  MSG_PP_SHUTDOWN        (MSG_PP+ 2)
$define  MSG_PP_STARTPATH       (MSG_PP+ 3)
$define  MSG_PP_PLANPATH        (MSG_PP+ 4)
$define  MSG_PP_REPLANPATH      (MSG_PP+ 5)
$define  MSG_PP_ALTPLANPATH     (MSG_PP+ 6)
$define  MSG_PP_DEMO_EXEC       (MSG_PP+ 7)
$define  MSG_PP_INVKIN          (MSG_PP+ 8)

$define  PP_NAME                "PathPlanner"
$define  ALT_PP_NAME            "AltPathPlanner"
$define  PP_TEMPDIR             "/usr2/testbed/tmp/"
$define  PP_ENTPT_FILENAME      "pathseg"

/* $define  STRUT_LENGTH          0.89 */
$define  PP_DEFAULT_SPEED       0.30
$define  PP_DEFAULT_BLEND       0.50

$define  X_OFFSET    ( 0.32)
$define  S_OFFSET    ( 0.1491)
$define  X_MIN       ( 0.50)
$define  Y_MIN       (-1.301)
$define  Y_MAX       ( 0.60)


  Data Types
  ----------

typedef struct
    {
    ARM_TYPE       arm  ;
    JOINT_VECTOR9  jnts ;
    }
STARTPATH_TYPE ;
--------------
holds a joint vector and the arm it belongs to.


typedef struct
    {
    ARM_TYPE       arm   ;
    TRANSFORM      pos   ;
    double         speed ;
    double         time  ;
    BOOL           strut ;
    double         deprt ;
    double         appr  ;
    }
PLANPATH_TYPE ;
----------------
Data block to be passed by CTOS message to PathPlanner, for planning paths.


   Procedures
   ----------

/*****************************************************************************
  ppInitialize  - initialize path planner
```

```
-------------------------------------------------------------------------
  RETURNS:        OK or ERROR indicating success in initializing path planner

  PARAMETERS:     TID_TYPE  tid       - TID of task calling ppStartPath
                  char      *filename - environment configuration file
*************************************************************************/
/*************************************************************************
  ppShutdown  - shut down path planner
-------------------------------------------------------------------------
  RETURNS:        (none)

  PARAMETERS:     TID_TYPE  tid       - TID of task calling ppStartPath
*************************************************************************/
/*************************************************************************
  ppStartPath  - set robot joint position to start of path
-------------------------------------------------------------------------
  RETURNS:        OK if joint positions were set,
                  ERROR if not set, e.g. if joint value is out of range

  PARAMETERS:     TID_TYPE       tid  - TID of task calling ppStartPath
                  ARM_TYPE       arm  - LEFT_ARM or RIGHT_ARM
                  JOINT_VECTOR9 *jvec - vector of joint positions
*************************************************************************/
/*************************************************************************
  ppResetPosition  - reads current joint positions and calls ppStartPath
-------------------------------------------------------------------------
  RETURNS:        char* to file name, or NULL if error occurred

  PARAMETERS:     TID_TYPE   tid      - TID of task calling ppResetPosition
                  ARM_TYPE   arm      - LEFT_ARM or RIGHT_ARM
*************************************************************************/
/*************************************************************************
  ppPlanPath  - plan a path and write to file
-------------------------------------------------------------------------
  RETURNS:        char* to file name, or NULL if error occurred

  PARAMETERS:     TID_TYPE   tid      - TID of task calling ppPlanPath
                  ARM_TYPE   arm      - LEFT_ARM or RIGHT_ARM
  TRANSFORM *destPos  - destination position of gripper
  double     speed    - motion speed [ratio of full speed]
  double     time     - time to complete path segment [seconds]
  BOOL       strut    - TRUE if carrying strut
  double     deprt    - length of depart segment
  double     appr     - length of approach segment

  Note: speed OR time should be specified, and the other set to zero.
*************************************************************************/
/*************************************************************************
  ppReplanPath  - request alternate path for previous destination
-------------------------------------------------------------------------
  RETURNS:        char* to file name, or NULL if error occurred

  PARAMETERS:     TID_TYPE   tid      - TID of task calling ppPlanPath
                  ARM_TYPE   arm      - LEFT_ARM or RIGHT_ARM
*************************************************************************/
```

Module **ppmain** (path planning procedures):

```
   Constants
   ---------

#define DEPAR_LEN   0.20
#define APPR_LEN    0.15


/* instructions constants that can be returned to main */
/*#define ERROR      0*/   /* illegal instruction received - skip          */
#define MOVE         1     /* start path planning                          */
#define GRASP        2     /* add payload to robot, remove a strut from env */
#define UNGRASP      3     /* remove payload, add strut to env             */
#define UNGRASP_FREE 4     /* same without position specification. Place
                              strut where robot left it.                   */
#define ADD_STRUT    5     /*   d strut to environment                     */
#define REMOVE_STRUT 6     /*   ove strut from environment                 */
#define JOINTS       7     /*   : joint vector directly                    */
#define VIEW         8     /* stop and display for user (graphics mode only)*/
#define QUIT_PP      9     /* quit this program                            */
#define ARM_LEFT    10     /* sets to left arm.                    */
#define ARM_RIGHT   11     /* sets to right arm.                   */


   Procedures
   ----------

   void Init_Instructions (srcLst)
LIST *srcLst) ;
-----------------------------------
Prepares the 'srcLst' for parsing.


   int Strut_Parameters  (expr, p1, p2, nbpar, retcode)
EXP *expr;
Vector *p1;
Vector *p2;
int nbpar;
int retcode;
-----------------------------------------------------
Reads the strut position from 'expr'.  'expr' must have 4 or 6 parameters
depending on the reresentation method.  There should be 'nbpar' more parameters
in 'expr'.  If no errors occur, then the strut position is returned in 'p1'
and 'p2'.  Else, ERROR is returned.


   int Get_Instruction (p1, p2, dir, qv, arm)
Vector *p1;
Vector *p2;
Vector *dir;
VAR *qv;
ARM_TYPE arm;
-------------------------------------------
Gets the next instruction from a file or from the CIRSSE interface.  The
instruction parameters are assigned to 'p1', 'p2', 'dir', 'qv'. The procedure
returns the instruction code number.


   void Output_Path (filename, pathLst, arm)
char *filename;
LIST *pathLst;
ARM_TYPE arm;
-------------------------------------------
Writes the path to a file, either in a Silma readable file, or a CIRSSE kntpt
```

depending upon CTOS_ACTIVE flag and PREVIEWER flag.

```
   STATUS Initialize_Path_Planner (configFile)
char *configFile;
-------------------------------------------
```

Initializes all path planner modules, must be called before planner is invoked.

```
   void Shutdown_Path_Planner (void)
-----------------------------------
```

Cleans up memory in planner's stack. Exits graphics routines.

```
   STATUS Demo_Exec (void)
-------------------------
```

Reads instructions from an input file for setup and execution.

Module **robot** :

```
   Procedures
   ----------

   void Init_Robot (source)
        LIST *source;
--------------------------
The robot is initialized and oriented according to the ROBOT command in the
input file. If no ROBOT command exists, the robot coordinate system is equal
to the world coordinate system. (This is the case at CIRSSE)

   BOOLEAN Joints_In_Range (qq)
        VAR *qq;
------------------------------
Returns TRUE if all joint values in 'qq' are within range. FALSE otherwise.
Inserts joint values into the list 'out_of_range'.

   BOOLEAN Set_Pos_Joints (new_q)
        VAR *new_q;
--------------------------------
Sets the robot to the pose defined by joint vector 'new_q' and updates the
transformation matrices.

   VAR *Get_Pos_Joints ()
----------------------------
Returns a New_Var to the current robot.c state of the joint vector.

   int Get_First_Locked_Joint (current)
LIST_EL **current;
---------------------------------------
Returns the first joint which is out of range.

   int Get_Next_Locked_Joint (current)
        LIST_EL **current;
-------------------------------------
Returns the next joint.

void Empty_Locked_Joint_List ()
-----------------------------
Empties the list 'out_of_range'.

int Length_Locked_Joint_List ()
-----------------------------
Returns the number of joints which are out of range.

   void Update_Model ()
----------------------
Updates the swept sphere models of the links to the current joint vector.

   void Update_Picture ()
-------------------------
Updates the wire frame picture of the robot to the current joint vector.

   BOOLEAN Is_Revolute_Joint (nb)
        int nb;
---------------------------------
Returns TRUE is link 'nb' is a revolute joint, FALSE otherwise. The first
link is link 1.

   MODEL *Link_Model (nb)
        int nb;
------------------------
```

```
Returns the model of link 'nb' or NULL if there is no swept sphere model of
this link. If the model was not up to date, it is automatically updated.
The first link is link 1.

   BOOLEAN Consider_For_Self_Collision (nb1, nb2)
        int nb1;
        int nb2;
-------------------------------------------------
Returns TRUE if link 'nb1' and link 'nb2' could collide and thus have to be
considered in the collision avoidance procedure.
The first link is link 1.

   BOOLEAN Consider_For_Arm_Collision (nb)
        int nb;
---------------------------------------
Returns TRUE if link 'nb' could collide with an obstacle in the environment
and thus has to be considered in the collision avoidance procedure.
The first link is link 1.

   Vector Origin (nb)
        int nb;
--------------------
Returns the origin of the coordinate frame of link 'nb'. Origin (0) returns
the origin of the robot's coordinate system.

   void Axis (nb, origin, dir)
        int     nb;
        Vector *origin;
        Vector *dir;
-----------------------------
Returns the axis of rotation (revolute) or the direction of motion (prismatic)
of joint 'nb'. The origin is also returned.

   double Joint_Value (nb)
        int nb;
-------------------------
Returns the current joint value of joint 'nb'. The first joint is number 1.

   void Joint_Range (nb, lower, upper)
        int    nb;
        double *lower;
        double *upper;
-------------------------------------
Returns the lowest and the highest possible value of joint 'nb'. The first
joint is number 1.

   double Joint_Weight (nb)
        int nb;
-------------------------
Returns the weight of joint 'nb' used for solving the Jacobian equation.
High values lead to high velocities of that joint.
The first joint is number 1.

   void Grasp_Part (p1, p2, rad, strutId)
        Vector p1;
        Vector p2;
        double  rad;
int    strutId;
-------------------------------
Puts a swept sphere cylinder as described by 'p1' and 'p2' and 'rad' in the
robot's gripper. The positions of the endpoints will change in this process,
but the length of the cylinder will be retained.  Connects gsm frame to tool.

   void Ungrasp_Part (p1, p2, strutId)
        Vector *p1;
        Vector *p2;
```

```
int    strutId;
----------------------------
Empties the robot's gripper and returns the last endpoint positions of the
payload.  Disconnects strutId's frame from tool.

   BOOLEAN Robot_Carrying ()
----------------------------
Returns TRUE if the robot is currently carrying a payload, FALSE otherwise.

   void Set_Pos_To_Payload (m)
        MODEL *m;
-----------------------------
Sets the endpoint positions of model 'm' to the positions they would have,
if the model was the payload of the robot.

   void Set_Pos_To_Gripper (m)
        MODEL *m;
-----------------------------
Sets the endpoint positions of model 'm' to the positions they would have,
if the model was the robot's gripper.
```

Module **spec** (CIRSSE/RAL machine specifications):

```
   Constants
   ---------

#define EARTH      0x23005a06
#define MARS       0x13004882
#define MERCURY    0x1700c726
#define JUPITER    0x1700cdd7
#define SOL        0x21000411
#define VENUS      0x5240dfe5
#define NEPTUNE    0x51006ee6
#define MOON       0x51001639


   Procedure description
   ---------------------

   BOOLEAN Graphics_OK ()
-------------------------
Returns TRUE if the machine on which the program is running has a graphics
screen and suncore is available.


   BOOLEAN Color_OK ()
----------------------
Returns TRUE if the machine on which the program is running has a color screen.
```

Module **stack** :

```
          Data types
          ----------

   typedef struct stack_el
   {
     struct stack_el *prev;
     char *data;
   } STACK_EL;
---------------------------
STACK_EL represents an entry of a stack. 'prev' is a pointer to the previous
entry and 'data' points to the user data represented by this stack element.


          Procedure description
          ---------------------

   void New_Stack (sp)
        STACK_EL **sp;
----------------------
Must be called before using a stack to initialize stack pointer 'sp'. 'sp' is
declared as follows:

Definition:
    STACK_EL *sp;

  Initialization:
    New_Stack (&sp);


   void Push (sp, data)
        STACK_EL **sp;
        char *data;
-----------------------
Places 'data' on stack 'sp'.


   char *Pop (sp)
        STACK_EL **sp;
----------------------
Returns the last data entry and removes it from the stack.


   char *Read_Top (sp)
        STACK_EL **sp;
----------------------
Returns the last data entry without changing the stack.
```

Module **usrFlags** (compiler directives):

```
   Constant Flags
   --------------

#define  MAN_IN_LOOP         /*  comment out to remove man from the loop    */
#define  STEP_BY_STEP        /*  prompt for viewer updates?                 */
#define  CTOS_ACTIVE /* indicates that PathPlanner.c, not main.c, is active */
#define  DIAGNOSTICS
#undef   PREVIEWER      /* we are in preview mode, not current display mode */
```

Module vector :

```
   Data types
   ----------

   typedef struct
   {
     float x, y, z;
   } Vector;
------------------
Represents a 3x1 vector with elements 'x', 'y', and 'z'.

   typedef struct
   {
     Vector v1, v2, v3;
   } Matrix;
----------------------
Represents a 3x3 matrix with column vectors 'v1', 'v2', and 'v3'.

   typedef struct
   {
     Matrix m;
     Vector v;
   } H_Matrix;
-----------------
Represents a 4x4 homogeneous transformation matrix. 'm' is the 3x3 matrix
for rotation in the upper left corner. 'v' is the translation vector in
the upper right corner. The last row is not stored, it is assumed to be
[0 0 0 1].



   Procedure description
   ---------------------

   Vector Vec (x, y, z)
        double x;
        double y;
        double z;
----------------------
Creates a vector with elements x, y, z and returns it.

   Matrix Mat (v1, v2, v3)
        Vector v1;
        Vector v2;
        Vector v3;
-------------------------
Creates a matrix with column vectors v1, v2, v3 and returns it.


   H_Matrix H_Mat (m, v)
        Matrix m;
        Vector v;
-----------------------
Creates and returns a homogeneous matrix with matrix m and 4th column vector v.


   Vector Add (a, b)
        Vector a;
        Vector b;
-------------------
Returns the sum of vectors a and b. (elementwise)
```

```
   Vector Sub (a, b)
        Vector a;
        Vector b;
--------------------
Returns the difference a-b. (elementwise)


   Vector Mul (a, s)
        Vector a;
        double s;
--------------------
Multiplies the elements of a with s and returns the result.


   Vector Div (a, s)
        Vector a;
        double s;
--------------------
Divides the elements of a by s and returns the result.


   Vector Neg (a)
        Vector a;
-----------------
Returns the elementwise negation of vector a.


   Vector Null_Vector ()
------------------------
Returns the null vector [0 0 0].


   double Dot_Prod (a, b)
        Vector a;
        Vector b;
------------------------
Returns the scalar or dot product of vectors a and b.


   Vector Cross_Prod (a, b)
        Vector a;
        Vector b;
--------------------------
Returns the cross product of vectors a and b.


   Matrix Dyad_Prod (a, b)
        Vector a;
        Vector b;
--------------------------
Returns the outer or dyadic product of vectors a and b.


   double Length (a)
        Vector a;
--------------------
Returns the length (absolute value) of vector a.


   Vector Scale (a, len)
        Vector a;
        double len;
-----------------------
Changes the length of vector a to 'len' and returns the result.
Prints an error message if 'a' is a null vector.
```

```
   Vector Scale_If_Longer (a, len, limit)
        Vector a;
        double len;
        double limit;
----------------------------------------
If the length of vector a is longer than 'limit' then its length is changed
to 'len', otherwise it is returned unchanged.


   Vector Normal (a, b)
        Vector a;
        Vector b;
----------------------
Returns vector 'a' projected on a plane normal to vector 'b'.


   Vector Center (a, b)
        Vector a;
        Vector b;
----------------------
Returns a vector whose endpoint is in the center between the endpoints of the
vectors a and b.


   BOOLEAN Parallel (a, b)
        Vector a;
        Vector b;
-------------------------
Returns TRUE if vectors a and b are parallel, FALSE otherwise.


   double Distance_Point_Line (point, line_p1, line_p2, line_result)
        Vector point;
        Vector line_p1;
        Vector line_p2;
        Vector *line_p_result;
-----------------------------------------------------------------
Returns the shortest distance between 'point' and the line bounded by the
points 'line_p1' and 'line_p2'. 'line_result' will contain the point on the
line which is closest to 'point'.


   Vector MxV_Prod (m, v)
        Matrix m;
        Vector v;
------------------------
Returns the product of matrix m and vector v.


   Vector HMxV_Prod (hm, v)
        H_Matrix hm;
        Vector v;
---------------------------
Returns the product of the homogeneous matrix hm and vector v. The 4th element
of 'v' and the result are omitted and assumed to be 1.


   Matrix MxM_Prod (m1, m2)
        Matrix m1;
        Matrix m2;
-------------------------
Returns the product of matrices m1 and m2.
```

```
   H_Matrix HMxHM_Prod (hm1, hm2)
        H_Matrix hm1;
        H_Matrix hm2;
--------------------------------
Returns the product of the homogeneous matrices hm1 and hm2.


   double Det (m)
        Matrix m;
----------------
Returns the determinant of matrix m.


   Matrix Transpose (m)
        Matrix m;
----------------------
Returns the transpose of matrix m.


   Matrix Inv (m, ok)
        Matrix m;
        BOOLEAN *ok;
--------------------
Returns the inverse of matrix m. If inversion was possible, 'ok' is set to
TRUE, otherwise to FALSE.


   Matrix I_Matrix ()
--------------------
Returns the identity matrix [1 0 0
                             0 1 0
                             0 0 1]


   BOOLEAN Plane_Line_Intersection (plane_p,plane_d1,plane_d2, line_p,line_d,
                                    t_plane1, t_plane2, t_line, distance)
        Vector plane_p;
        Vector plane_d1;
        Vector plane_d2;
        Vector line_p;
        Vector line_d;
        double *t_plane1;
        double *t_plane2;
        double *t_line;
        double *distance;
-------------------------------------------------------------------------------
Intersects the plane defined by location vector 'plane_p' and direction
vectors 'plane_d1' and 'plane_d2' with the line defined by location vector
'line_p' and direction vector 'line_d'. If this intersection is possible,
TRUE is returned and the parameters of the intersection point for both
the plane and the line are returned. The two equations for the intersection
point are:
  ip = plane_p + t_plane1 * plane_d1 + t_plane2 * plane_d2
  ip = line_p  + t_line   * line_d
If intersection is not possible (line is parallel to the plane) then FALSE
is returned and the distance between the plane and the line is returned in
'distance'.


   double Distance_Line_Line (a1, a2, b1, b2, a_result, b_result)
        Vector a1;
        Vector a2;
        Vector b1;
        Vector b2;
        Vector *a_result;
        Vector *b_result;
```

```
-------------------------------------------------------------
Returns the distance between the line bounded by a1 and a2 and the line
bounded by b1 and b2. The points of closest distance on the lines are
returned in 'a_result' and 'b_result'.

    Distance_Seg_Plane (p1, p2, a, b, c, s_result, p_result)
  Vector p1;
  Vector p2;
  Vector a;
  Vector b;
  Vector c;
  Vector *s_result;
  Vector *p_result;
-------------------------------------------------------------
Returns the distance between the segment bounded by p1 and p2
and the planar triangle bounded by the verticies a, b, and c.
The points closest on the segment and the plane are returned
in s_result and p_result, respectively.


   Vector Random_Vector (len)
        double len;
----------------------------
Returns a random vector of maximum length 'len'.


   void Print_Vector (a)
        Vector a;
-----------------------
Prints vector a to the screen.


   void Print_Matrix (m)
        Matrix m;
-----------------------
Prints matrix m to the screen.


   void Print_H_Matrix (hm)
        H_Matrix hm;
--------------------------
Prints the homogeneous matrix hm to the screen.
```

PART II: Planning Collision Free Paths for Two Cooperating Robots Using a Divide-and-Conquer C-Space Traversal Heuristic

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

A method has been developed to plan feasible and obstacle-avoiding paths for two spatial robots working cooperatively in a known static environment. Cooperating spatial robots as referred to herein are robots which work in 6D task space while simultaneously grasping and manipulating a common, rigid payload. The approach is configuration space (c-space) based and performs selective rather than exhaustive c-space mapping. No expensive precomputations are required. A novel, divide-and-conquer type of heuristic is used to guide the selective mapping process. The heuristic does not involve any robot, environment, or task specific assumptions. A technique has also been developed which enables solution of the cooperating redundant robot path planning problem without requiring the use of inverse kinematics for a redundant robot.

The path planning strategy involves first attempting to traverse along the configuration space vector from the start point towards the goal point. If an unsafe region is encountered, an intermediate via point is identified by conducting a systematic search in the hyperplane orthogonal to and bisecting the unsafe region of the vector. This process is repeatedly applied until a solution to the global path planning problem is obtained. The basic concept behind this strategy is that better local decisions at the beginning of the trouble region may be made if a possible way around the "center" of the trouble region is known. Thus, rather than attempting paths which look promising locally (at the beginning of a trouble region) but which may not yield overall results, the heuristic attempts local strategies that appear promising for circumventing the unsafe region.

Although this method cannot guarantee finding a solution even if one exists, and in spite of its $O(k^{n-1})$ (where $k = 2$ or 3 as implemented) complexity for n degree of freedom problems, it has demonstrated the ability to solve a variety of practical yet potentially difficult path planning problems within a reasonable amount

of computation. The method inherently handles singularities and is applicable to robots having any number and type of joints. Parallel processing could be used to greatly reduce solution time.

Because the main emphasis of the path planning method is to produce a feasible path without regard to any type of optimality, the paths developed are often rather inefficient. Thus, a configuration space based algorithm was developed to modify any feasible path found by the planner into a more efficient one, where efficiency is measured by the length of the c-space trajectories.

Although the key motivation behind this work was to address the path planning problem for two cooperating robots, the methods developed are directly applicable to single robots as well. The path planner is implemented in C and utilizes polytope models of the robots and obstacles for purposes of interference detection. The path planner is demonstrated via computer graphics simulation on a Sun SparcStation 1 for several single and cooperating robot cases, including cooperating nine degree of freedom (1P-8R) robots.

CHAPTER 1
Introduction

1.1 Motivation

Robotics is a technology with a promising future. The explosion of knowledge resulting from past and present research efforts will manifest itself in robotic systems capable of emulating the human attributes of mobility, dexterity, intelligence, and sensory perception. There will be mobile bases with multiple cooperating arms having extensive sensing capability which are able to receive high level instructions and translate those instructions into a specific sequence of low level actions required to execute the desired task. Robotic systems of the near future will strive for increased flexibility, improved reliability, and greater autonomy.

One issue which arises in attempts to develop more autonomous robotic systems is the path planning problem. The path planning problem involves determining if a continuous and obstacle avoiding path exists between a robot's start and goal positions, and, if so, to determine such a path. If the mathematical space of concern is considered to be the configuration space (c-space) of the robot, then the problem is effectively that of finding a connected graph through c-space between the start and goal positions which traverses only feasible and collision free points. This path planning problem can become very computationally intensive. In fact, an upper bound on the complexity of the n degree of freedom (dof) path planning problem is $O(n^n)$, i.e., complexity of the path planning problem is exponential in the number of dof [1-3]. To illustrate the rapid growth in complexity with number of dof, note that a six dof problem would be more complex than a two dof problem by a factor of 6^6 / 2^2, or 11,664.

A subset of the general path planning problem just described is the path

planning problem for two cooperating robots. Robotic cooperation herein refers to the scenario whereby both robots simultaneously grip and manipulate a common, rigid, payload. Since the two arms grasp the object rigidly, the relative position and orientation of the two grippers must be invariant during the motion. As an example of two arm cooperation, refer to Figure 1.1, where two nine degree of freedom robots are shown cooperatively manipulating a long, cylindrical payload.

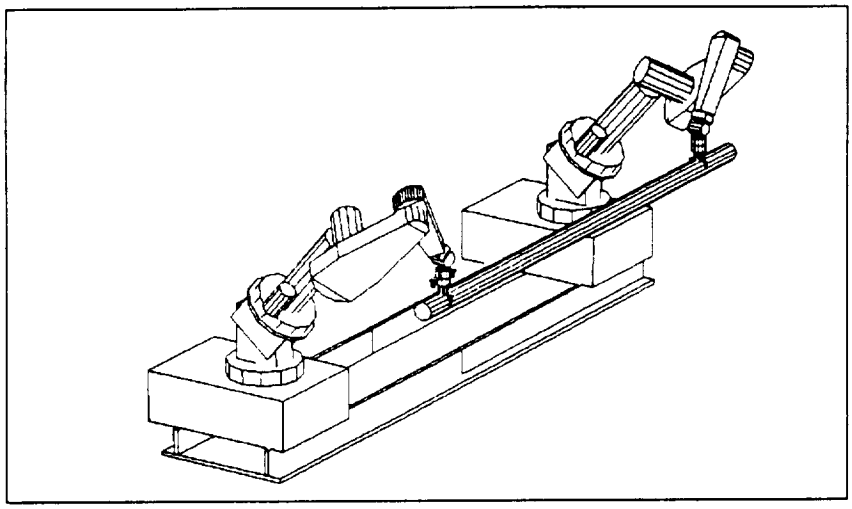


Figure 1.1: Two 9-DOF Robots Working Cooperatively

The effective number of degrees of freedom or mobility, m, for two spatial robots working cooperatively in six dimensional task space can be simply computed from:

$$m = n_1 + n_2 - 6 \tag{1.1}$$

where n_i represents the number of degrees of freedom for robot i, and the '-6' term results from the closure constraint imposed by cooperation.

There are many potential applications for two arm cooperation. For example, a space station will most likely be built using robots to minimize the expense and risk of putting humans into space. In order to be most effective, the robot arms

would likely cooperate and be autonomous or at least semi-autonomous. The attractiveness of lightweight robots for space applications increases the likelihood that robotic cooperation would be necessary to manipulate large or massive payloads. In industry, robotic cooperation might be employed for moving very large or very flexible payloads which exceed the capacity of a single arm or require support at more than one point. Cooperating robots could also be used to manipulate two parts with mating surfaces but which are not fastened to each other.

The addition of a second manipulator for cooperative work leads to an inherently complex system. One key research issue and open problem associated with a system of cooperating robots is the path planning problem. The cooperating robot path planning problem must consider not only collision avoidance but also the kinematic closure requirement that both robots are able to reach their respective grasp positions at all times. Dooley [4] shows how the closure constraint plus obstacle constraints for cooperating planar robots can combine to produce a configuration space containing many unusually shaped unsafe regions and relatively little safe space. One can conclude both intuitively and from Dooley's work that the path planning problem in the cooperating robot case will typically be more difficult than in the single robot case.

Numerous approaches to the general single arm path planning problem have appeared in the literature. Most do not appear directly suited to the case of two cooperating robot arms. Many of these approaches do, however, attempt to find a path while applying some heuristic to selectively search configuration space. The only *practical* planners to date for a general six degree of freedom (dof) robot involve simplifications or heuristics and are not complete, i.e., they cannot guarantee finding a solution even if one may exist. Many of the approaches in the literature which do address path planning for cooperating robots consider only planar systems and cannot be practically extended to the case of two robots having six or

more dof each. Some researchers have solved the cooperating arm path planning problem with multi-dof spatial (working in 6D task space) robots but they present results only for relatively (or completely) obstacle-free environments. The difficulty which researchers have experienced in trying to solve the general cooperating robot path planning problem is evidence of the inherent complexity of the problem and highlights the need for further study.

The work presented herein was funded by the Center for Intelligent Robotic Systems for Space Exploration (CIRSSE), a NASA sponsored research center at Rensselaer Polytechnic Institute (RPI), and is part of CIRSSE's efforts to develop autonomous and teleoperated single and cooperating robot systems for use in space. The CIRSSE testbed, a computer graphics representation of which is shown in Figure 1.2, includes two nine dof robots which may work independently or cooperatively.

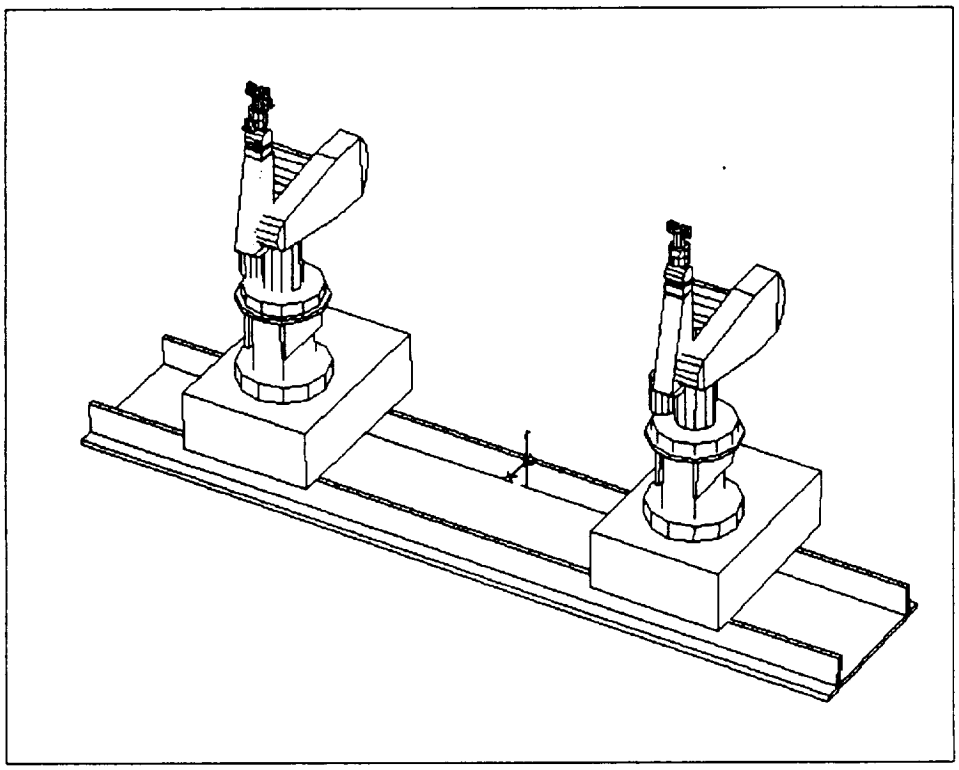


Figure 1.2: The CIRSSE Testbed

Each nine dof robot consists of a 6 dof (6R) Puma 560 mounted to a 3 dof (1P-2R) platform. As shown in the figure, each platform has a translate, a rotate, and a tilt axis. The testbed has extensive sensing capabilities, including various CCD cameras, laser range finding, and force/torque sensing end effectors. The principle motivation for this work was the desire to develop a practical and potentially useful path planner for cooperating robot scenarios on the CIRSSE testbed. Nonetheless, the strategy herein is completely general and no assumptions are made which would limit the usefulness of the approach to specific robots, environments, or tasks.

1.2 Direction of this Work

This section briefly summarizes the assumptions, goals, strategy, and results of the work presented in this thesis.

1.2.1 Assumptions

This work assumes the following:

1. Forward kinematic models of the robots are available.
2. Inverse kinematic models of the robots are available for six dof robots or for the final six links of redundant robots.
3. Geometric models of the robots, payload, and obstacles are available.
4. Obstacles in the workspace are static.
5. Feasible and collision free start and goal joint configurations of the robots are known, as are the start and goal positions of the payload.
6. Motion between the specified start and goal positions may be arbitrary.
7. The planner may ignore robot dynamics.

1.2.2 Goals

The goals of this work are to develop a planner capable of solving the cooperating robot path planning problem while satisfying the following:

1. The planner shall locate reasonable collision-free paths in a time frame suitable for off-line path planning.

2. The planner shall be capable of modifying a feasible path into a more efficient one.

3. The planner shall be applicable to various robotic systems and various tasks.

4. The planner shall be practical for cooperating six dof manipulators as a minimum, and ideally for cooperating redundant robots.

5. The planner output shall be a sequential listing of closely spaced knot points in joint space which represent the discretization of a continuous, feasible, and obstacle avoiding path connecting the start and goal configurations.

1.2.3 Strategy

This thesis presents a new method for performing global path planning for two cooperating spatial robots in a static environment. Like the single arm planner presented by Dupont [5], the principle strategy is to minimize the computationally expensive mapping of configuration space by performing mapping on an as required basis. The planner satisfies the goals outlined in Subsection 1.2.2. The approach is based around a novel, divide-and-conquer style heuristic for traversing through c-space. This c-space traversal heuristic is directly applicable to the single robot path planning problem and can be easily tailored to the case of two cooperating robots. In all cases the dimensionality of the c-space considered is an accurate reflection of the actual problem complexity. Computationally expensive precomputations and

exhaustive c-space mappings are avoided. This thesis also presents a technique which allows the path planning method to be applied to cooperating redundant robots without requiring the use of inverse kinematics for a redundant robot. The path planning method is applicable regardless of the number and type of joints in the robot and for any number of obstacles in the workspace. A *string tightening* algorithm is presented to modify any safe path found by the planner into a more efficient one, where efficiency is measured by the length of the joint space trajectory.

The path planning method involves first attempting to traverse a c-space vector from the start to the goal of one of the robots. If this vector passes through unsafe space, the hyperspace orthogonal to and bisecting the unsafe segment of the vector is systematically searched to identify an intermediate goal point for consideration as a via point. An attempt is made to traverse from the last safe point to the intermediate goal point. This process is repeated as necessary until the attempted traversal to the newest intermediate goal point is entirely safe. At that point, progression is attempted toward all previous guide points in the opposite order in which they were found, where guide points include not only previous intermediate goal points but also the safe points found on the goal end of each unsafe region which invoked a search. When progression to a particular guide point is not entirely safe, that point is permanently dismissed and progression is attempted toward the next guide point in the specified sequence. The progression continues until an attempt has been made to progress to the global goal point. If that attempted progression is not entirely successful the overall process is repeated until the global goal point has been safely traversed to.

Unfortunately, the path planning method presented herein is not complete, i.e., it cannot guarantee finding a solution even if one exists. Though certainly undesirable, this lack of completeness does not seem unreasonable since researchers have thus far been unable to develop algorithms which achieve both completeness

and practicality for reasonably difficult yet practical path planning problems for more than a few degrees of freedom. Since our emphasis was toward achieving a potentially useful path planner for cooperating robots with at least six dof each, we sacrificed completeness in exchange for the possibility of solving some practical yet potentially difficult problems as quickly as possible.

Unlike some path planning techniques which are geometric model data structure specific, our planner may be used with any geometric modeling scheme which allows for interference detection. Our implementation utilizes polytope representations of the links of the robots and of the obstacles in the workspace as presented by Schima [6]. The polytope scheme was chosen because it allows for detailed modeling of objects while enabling relatively fast interference checking. The potential speed of the collision detection is enhanced by the fact that the method simply needs a *yes* or a *no* regarding collisions and does not require distance or direction information. Mapping a particular point in c-space involves verifying that the closure constraint can be met, updating polytope models of the robot links and payload, and performing two phase interference detection calculations on the resulting polytopes. The first phase of interference detection simply checks for interference between spheres which bound each polytope. If the spheres intersect indicating possible collision then the second phase of interference detection must be invoked. The second phase accurately determines whether or not two polytope models intersect.

1.2.4 Results

Despite its simplicity, the methodology presented herein appears to solve the cooperating robot path planning problem better than other approaches presented in the literature. The method is also applicable to the single robot path planning problem. The approach does, however, suffer from one drawback currently afflicting all *practical* motion planners which can handle six or more dof, namely that it may

fail to find a solution even if one exists. An upper bound on the complexity of the planner is $O(k^{n-1})$ for an n dof problem, where $k < n$. For our implementation, $k = 3$ for all cases except cooperating redundant manipulators for which $k = 2$. This compares favorably to the actual problem complexity which has an upper bound of $O(n^n)$.

Sample results are included for a single six dof robot, a single nine dof robot, cooperating six dof robots, and cooperating nine dof robots. The results illustrate the planner's ability to solve practical yet potentially difficult problems. The path planner was implemented in the C programming language and runs on a Sun SparcStation 1. Paths found by the planner are animated using CimStation, a commercially available computer graphics robot simulation package [7]. Typical time required to find a feasible path for cooperating nine dof robots (the most complex scenario considered) with several workspace obstacles is less than 30 minutes. After finding a feasible path, the string tightening process for cooperating nine dof robots typically requires about 15 minutes of computation. Parallel processing could be used to significantly reduce execution times for both the path planning and string tightening routines since both involve a large number of independent calculations.

1.3 Overview of Thesis

The remainder of this thesis is presented in seven main chapters:

- Literature Review
- Problem Statement
- Divide-and-Conquer C-Space Traversal Heuristic
- Utilizing the Heuristic for Robot Path Planning
- Implementation and Results

- Discussion of the Path Planning Strategy
- Conclusions and Future Work

A literature review on published techniques for single and cooperating robot path planning is discussed in Chapter 2. The path planning problem is formally defined in Chapter 3. Chapters 4 and 5 present the central contribution of this thesis, namely a new c-space traversal heuristic and means for utilizing the heuristic to solve single and cooperating robot path planning problems. In Chapter 6, implementation details and results are presented for application of the path planning strategy with string tightening to four cases: a single six dof robot, a single nine dof robot, two cooperating six dof robots, and two cooperating nine dof robots. A discussion of the path planning strategy is given in Chapter 7. Finally, Chapter 8 presents some conclusions and some areas for future work.

CHAPTER 2
Literature Review

This chapter presents a literature review on the subject of robot path planning. The chapter is organized into the following main sections:

- Path Planning for Single Robots
- Path Planning for Cooperating Robots
- Other Related Areas of Research
- Summary of the Literature Review

While we are specifically interested in path planning for cooperating robots, an understanding of current methods for a single robot is pertinent to determining the possible suitability of extending those methods to consider cooperating robots. Hence, the discussion below includes methods for both single robots, presented in Section 2.1, as well as for cooperating robots, presented in Section 2.2. Other related areas of path planning research are briefly discussed in Section 2.3. Finally, a brief summary of the literature review is presented in Section 2.4. A brief review of literature regarding algorithms for string tightening is presented later in Section 5.3.1.

2.1 Path Planning for Single Robots

Published approaches to the single robot path planning problem are discussed in this section. Most of these approaches can be characterized as one of the following three types:

- A graph search approach

- A potential fields approach
- A human assisted approach

These categorizations are not mutually exclusive, and a combination of them is often used in a path planning strategy. These approaches are discussed below.

2.1.1 The Graph Search Approach

One approach to solving the path planning problem could be referred to as the graph search type approach. Such an approach will work directly in the configuration space (c-space) in attempt to find a connectivity graph of safe points between an initial configuration and a goal configuration [5,8-33].

Configuration space as first suggested by Lozano-Perez and Wesley [33] refers to the n-dimensional space formed by the n values of the joint variables of a robot with n degrees of freedom (dof). Thus, each possible configuration which the robot can assume is represented by a point in the configuration space. The robot path planning problem is thus equivalent to the motion planning problem of a point in c-space. The concept of c-space is illustrated in Figure 2.1. Consider the 2R planar manipulator shown in Figure 2.1a. If it is assumed that each joint has both upper and lower limits then the resulting c-space is rectangular as shown in Figure 2.1b. If there were no joint limits the resulting c-space would be toroidal.

C-space can be divided into two regions: safe and unsafe. Safe space refers to the locus of all points in configuration space which represent feasible and collision free configurations. All space which is not safe for any reason is simply categorized as unsafe space.

A path planning technique is considered *complete* if it can either guarantee finding a solution if one exists or prove that one does not exist. Early efforts at developing complete graph search techniques indicate that path planning in this fashion is exponential in the number of degrees of freedom. For example, Reif [1]

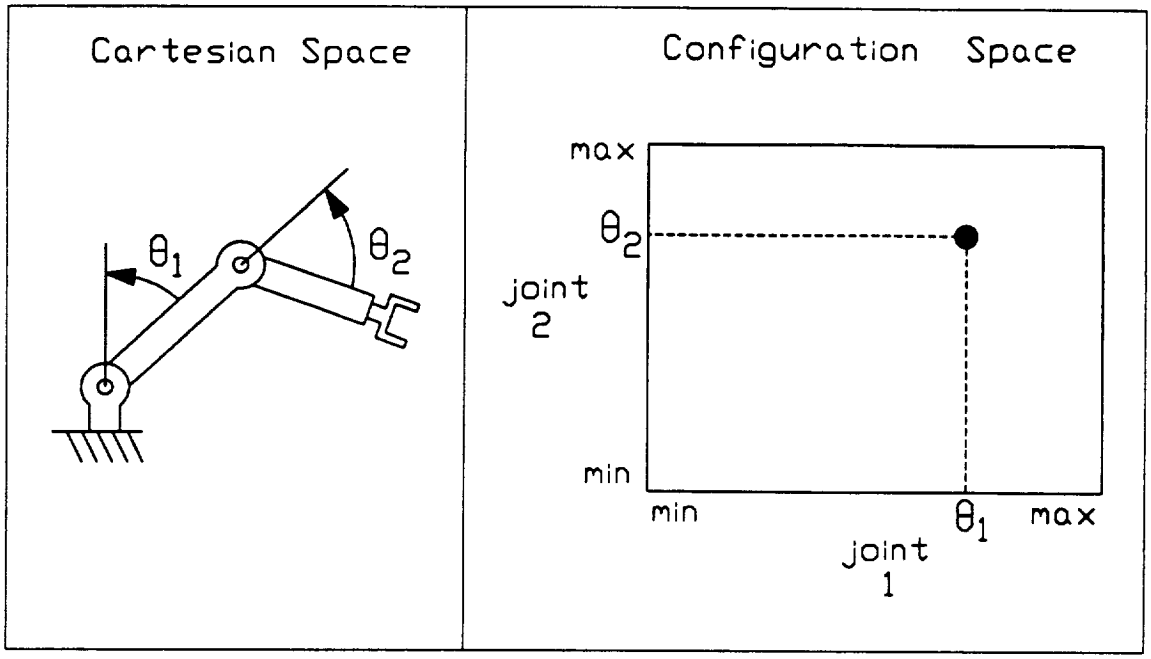


Figure 2.1: A 2D Planar Robot and its Configuration Space

examined the 3D *generalized* mover's problem. The mover's problem (often referred to as the piano mover's problem) involves path planning for a single solid object. The *generalized* mover's problem involves path planning for an object which may consist of multiple objects kinematically linked together (such as a robot arm). The fact that Reif could show this generalized problem is PSPACE-hard is evidence that the computational bounds for robot motion planning problems in fact grow exponentially with degrees of freedom. An explanation of PSPACE-hardness may be found in [34]. An upper bound on complexity of the robot path planning problem is $O(n^n)$ for an n dof robot [2, 3].

Most graph search techniques utilize global world knowledge. In addition, many use an A* type of heuristic search to find a feasible path. The A* algorithm is a common search procedure whereby paths to the goal are built and compared based

on a heuristic estimate of the cost remaining to reach the goal. The algorithm continually expands the most promising path until a solution is found. Unfortunately, searching for the optimal path has led most researchers to transform all obstacles into c-space [9,17-19,21-25,28,30,31]. Because of the higher order complexity of such a technique, the more successful works involved simplifications to reduce problem dimensionality [12, 17, 18, 26]. The basic shortcoming of the A* type searches is the fact that they tend to exhaustively map out concavities encountered in trying to go between the start and the goal. A 2D example of this phenomenon is illustrated in Figure 2.2. The likely computational expense of such an approach makes it impractical for motion planning for robots with more than a few dof. The A* algorithm can also be applied bidirectionally by considering extending the path from both the start and the goal positions. Bidirectional searching can be effective since it is generally easier to move from a cluttered space to an open space than vice versa.

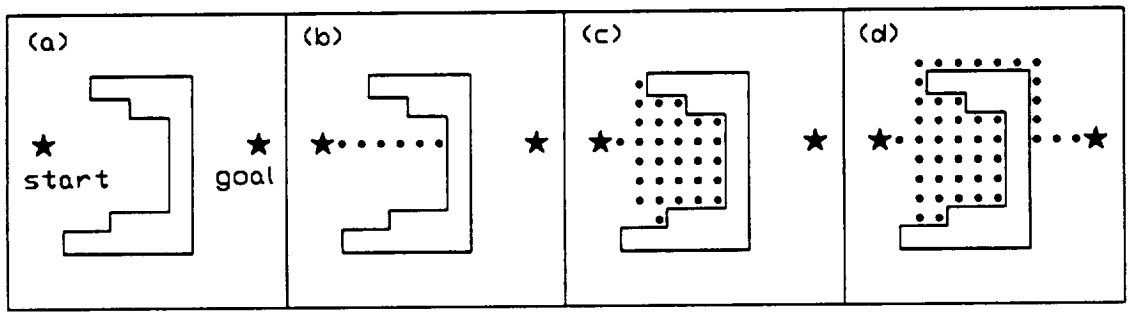


Figure 2.2: Exhaustive Mapping of Concavities Using A* Heuristic

Other complete techniques which are not computationally practical for higher degrees of freedom are presented by Branicky [35], Canny [13], and Paden [36]. Kondo [37] has reported a fast and complete algorithm for six dof robots, but the algorithm's speed is only demonstrated for apparently simple problems.

Chen and Hwang [38] present a complete solution technique with performance

commensurate with task difficulty. Essentially, they use a global planner to select regions of collision free space which connect the start and goal and then use a local planner to solve the path planning problem within each region. The resolution of the regions is only as fine as necessary to find a solution using a heuristic to select promising regions for further subdivision. In this way, easy problems may be solved relatively quickly and yet an extremely difficult problem may be resolved to whatever level is required to obtain a solution or conclude one does not exist. Their algorithm solves a relatively simple yet practical disassembly task for a five dof Adept robot in three minutes on a 16 MIPS workstation.

Sharir [32] notes the mathematical complexity and size of the general complete solution of robot motion planning in an n-dimensional c-space and presents a graph search algorithm aimed at solving it. Sharir develops an algorithm which is conceptually applicable to a system of arbitrary dimension. His algorithms can be most easily described by considering the 2D problem of planning the movement of a line segment in a planar space containing polygonal obstacles. The line segment is free to translate but may not rotate. Sharir's algorithm groups the 2D c-space into closed polygonal regions which are homogeneous (completely safe or unsafe). Then the problem of motion planning becomes that of searching for a connectivity graph between the initial and final positions in the polygon which contains those points. While this approach is interesting and successful in 2D, Sharir acknowledges that both the breaking down of regions in configuration space and the graph search suffer from higher order explosion; to the point of intractability.

The mathematical complexity of the general motion planning problem has resulted in many techniques which reduce the problem dimensionality via simplifications. Some such simplifications have included allowing only cartesian manipulators [24], requiring arm seperability (small wrists which orient a spherical payload) [15,17,18,23,26,28,39-41], or allowing only certain motions and obstacle

types [12, 20, 26, 42]. None of these constraints can be used for path planning for two cooperating robot arms.

Gupta [43] presents a sequential search strategy which plans the motion of each robot link successively starting from the base. While not complete for robots with three or more links, this technique is very efficient and may be useful for quickly solving some simple problems.

One technique which has been used for path planning in c-space involves hypothesizing a path and then testing it at a finite number of intermediate points for collisions. The path is repeatedly modified heuristically until a solution is found. Lewis [44] suggested precomputing commonly used path segments referred to as freeways and recommended the use of intermediate safe points to avoid detected collisions. However, he presented no mechanism by which to determine these intermediate safe points.

Pieper [45] applied various cartesian heuristics to attempt to bypass obstacles. The arm could fold to move in front of or above detected obstacles. Pieper found that certain obstacle arrangements resulted in the manipulator oscillating between obstacles. In addition, the algorithm generally failed if the only acceptable path led between two obstacles.

Glavina [46] presents a heuristic path planning method which combines goal-directed searches with randomized searches as needed. The algorithm proceeds straight in c-space from start towards goal until an obstacle boundary is encountered. At that point, the point slides along the obstacle boundary if and only if such motion will reduce the distance to the goal. In 2D, sliding is attempted by searching for a safe point along a line orthogonal to the desired direction passing through the first point which violated an obstacle boundary. This concept is illustrated in Figure 2.3. If this sliding alone is not sufficient to clear the obstacle, a new subgoal is created at random and the process is repeated until a feasible path to the goal is found.

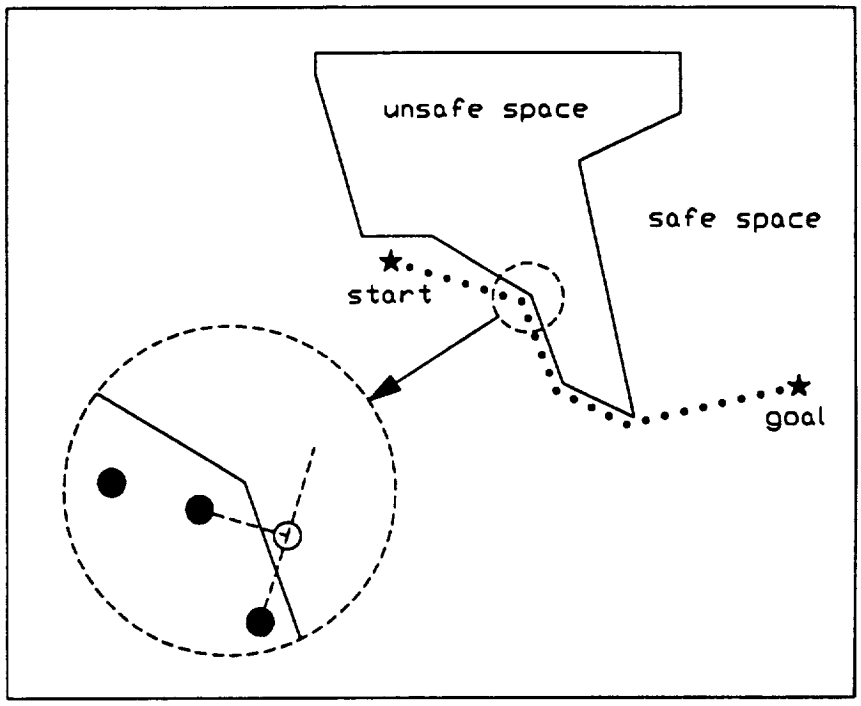


Figure 2.3: Goal Directed Sliding

Glavina has results for a 2D prototype and hopes to extend the procedure to a six dof general purpose manipulator. For the six dof problem, Glavina proposes perhaps checking 10 possible sliding directions corresponding to each direction of the basis axes of the 5D hyperplane along which sliding can be attempted. Further research is planned to determine if it is necessary to expand the set of test vectors beyond this set.

Many papers have dealt with the motion planning of polygons or polyhedral objects [8, 11, 13, 15, 18, 24, 47]. While this is the simplest form of the motion planning problem, this research is useful for mobile robot path planning and forms a foundation for planning problems of higher dimensionality. The actual methods used, however, have generally not extended into higher dimensions easily due to the added complexity of that space. Mobile robot path planning has been an attractive

research area because of the low dimensionality involved and because of the practical applications of mobile robots [9, 21, 22].

Lozano-Perez and Wesley [26, 33] present a visibility graph (vgraph) technique for polygonal and polyhedral objects. Vgraphs are graphs whose nodes are the vertices of polyhedral c-space obstacles. Nodes which are visible to each other are linked and assigned a weight proportional to the distance between them. The graph is then searched for the optimal path. It is difficult to effectively apply vgraphs to problems in more than two dimensions. For example, the vgraphs can be constructed from the vertices of polyhedra, but the shortest path no longer lies in the visibility graph.

Rovetta [48] presents a more recent variation on the vgraph method whereby all obstacles which impede the traversal straight from start to goal are grouped into a single monoobstacle consisting of the convex hull of the individual problem obstacles. Such an approach reduces computation and produces more efficient paths but it may convert a solvable problem into an unsolvable one.

Two other free space searching techniques include generalized cones [49] and voronoi diagrams [8, 50, 51]. The first technique produces a safe path by piecing together the centerlines of generalized cones whose sides are the faces of the obstacles. The generalized cone algorithm translates a polygonal moving body along the axes of the generalized cones and rotates it at cone intersections. This algorithm may fail when an object must translate and rotate simultaneously to avoid obstacles. A voronoi diagram is a collection of surfaces that are equidistant from two or more obstacles. A safe path is found by traversing appropriate regions of these surfaces. These two techniques have the desirable feature of keeping the robot as far from obstacles as possible. In a narrow corridor, this is a desirable feature. In cases which much open space, however, it may yield a much longer path than necessary. It is difficult to apply either of these techniques in more than 2D.

An interesting path planning technique is presented by Lumelsky [52-56]. He makes three assumptions: (1) The arm endpoint can move through a simple curve, (2) when the arm hits an obstacle, it can identify the contact point on the arm, and (3) the robot can follow an obstacle boundary. While only local information is used, Lumelsky's algorithm is complete. He reduces planning a path for a robot to planning a path for a point on the surface of some manifold. In two dimensions he is able to apply his algorithm using the "same turn first" strategy for traversing the surface of any obstacles encountered in the straight line path from start to goal. His work has yet to be implemented for more than two degrees of freedom since, in that case, there are an infinite number of possible directions to follow on the obstacle boundary. To simplify this situation, Petroz and Sirota [57] suggest cutting the obstacles in the higher dimensional c-space with planes to limit the boundary following directions to right and left. The difficulties with this approach are that an infinite number of such planes exist and that a solution will typically need to employ more than one such plane.

Lozano-Perez and Wesley [24, 25, 33] describe an approach for motion planning which is based on the idea of expanding obstacles. This approach essentially involves the expansion of the obstacles in such a manner as to reduce the path planning problem for an n-dimensional shape to an equivalent problem for a single point in that n-dimensional space, where it is the expansion of the obstacles that allows the equivalence. Computational complexity becomes excessively burdensome for cases of dimensionality greater than two. Very little is known about how to apply Lozano-Perez's algorithm to systems with three or more degrees of freedom, although Lozano-Perez has expanded the procedure to consider cartesian manipulators (robots with three prismatic joints).

Warren [58] presents a vector based algorithm currently being developed for planning the path of a robot among irregularly shaped obstacles. In this technique,

a c-space vector is created from the start position to the goal position. If this vector crosses unsafe space, a second vector is used to determine a new intermediate goal and the previous goal is stored for later use. This second vector is drawn from the centroid of the obstacle though the midpoint of the unsafe potion of the initial vector and continues until reaching a point in safe space. The overall procedure is applied repeatedly until the ultimate goal can be reached. A 2D illustration of this approach is shown in Figure 2.4. This technique has a divide-and-conquer flavor to it but has

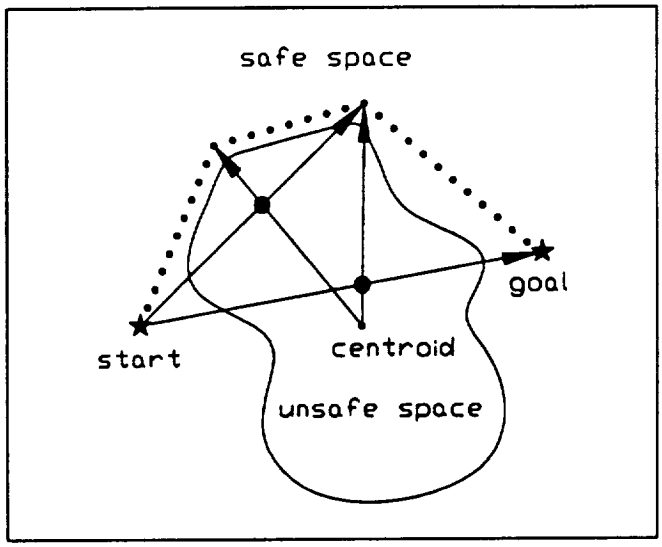


Figure 2.4: Vector Based Divide-and-Conquer

drawbacks which limit its effectiveness to only a few dof. These drawbacks include requiring exhaustive mapping of obstacles and having no guarantee of finding a safe point along the vector from the centroid through the midpoint of the unsafe region.

A recent divide-and-conquer based approach is a heuristic approach for cartesian manipulators presented by Lee [59]. Lee divides the cartesian robot pick-and-place task into a vertical departure motion, an intermediate planar motion, and a vertical approach motion. The 2D vgraph algorithm is used to solve each phase of the problem using heuristics to address part rotation about a vertical axis. This

approach cannot be practically applied to spatial manipulato: ath planning problems.

Dupont [5] addresses the path planning problem for kinematically redundant manipulators. The basic philosophy employed by Dupont is that of performing selective rather than exhaustive mapping of configuration space thereby minimizing the exponential growth problems associated with complete graph search techniques. The strategy which Dupont follows involves first creating a path which is linear in joint space (c-space) between the start and goal positions. This path is discretized and checked for collisions at each point along the path. Dupont attempts to traverse around regions along the initial path where collisions occur by applying some heuristics to choose a cartesian strategy direction that will likely allow circumvention of the trouble regions. The Jacobian is then used to determine the possible safe c-space moves that achieve the desired task space strategy directions. Octree representations are used to determine if collisions occur for a given configuration. Dupont's algorithm successfully planned obstacle avoiding paths for a seven dof redundant manipulator.

A somewhat similar approach is taken by Kondo et al [60]. Although Kondo's intended application is the movement of parts and assemblies within CAD system representations (this type of problem is often referred to as the piano mover's problem), the nature of that problem directly parallels the robot motion planning problem. Kondo's basic approach is similar to Dupont's in that he tries to restrict the amount of c-space referred to during a path search (by selectively mapping c-space) in order to avoid executing unnecessary collision detections. Kondo uses octrees and combines a global strategy with a local strategy. The global strategy finds the limits of free space which are encountered in going from the start toward the goal and from the goal toward the start. The local strategy then enumerates only cells along the boundary of the free space which was encountered in attempting

to traverse directly between the start and goal positions. It is in this manner that Kondo's algorithm greatly reduces the typically burdensome amounts of computation and storage required to fully define an octree representation of the workspace. In addition, by looking only from the start towards the goal (and vice versa) until a collision occurs, Kondo is avoiding searching potentially large islands of safe space which are unreachable. Using the piano mover's analogy and trying to move the piano from the hallway to the dining room, Kondo's algorithm will avoid searching the bedroom if there is no possible way the piano could have gotten into the bedroom. Kondo applied his algorithm to determine a collision free path for moving a heat exchanger between two positions in a CAD model of a nuclear power plant.

2.1.2 The Potential Fields Approach

An alternate type of approach is based on the use of artificial potential fields. Such an approach typically regards obstacles as a source of repelling potential field, while the desired goal position represents a strong attractor [61]. The hope is that the moving body can safely traverse from its initial position to the desired goal position simply by following the potential gradient of the resulting field. The square of the inverse of distance to obstacles and the negative of the inverse of distance to the goal are commonly used obstacle and goal potentials, respectively. The potential fields approach is typically implemented in task space [62, 63] although some researchers have examined implementing it in configuration space [64, 65]. Some advantages and disadvantages of potential fields approach are noted below.

2.1.2.1 Advantages of the Potential Fields Approach

1. They are faster than other algorithmic methods developed to date.

2. They are readily extended to systems of higher dimensionality.

3. They inherently tend to produce paths which avoid obstacles with significant clearances.

2.1.2.2 Disadvantages of the Potential Fields Approach

1. They tend to have difficulty with local minima, particularly for systems of higher dimensionality.
2. It is difficult to maintain a global nature since the strength of the attractors and repellors generally is significant only over small distances.
3. They can have difficulty dealing with arbitrarily shaped obstacles.
4. Implementation in c-space requires knowledge of c-space obstacles.
5. The expression for the obstacle potential becomes cumbersome when there are many concave objects.
6. They are not as thorough as graph search techniques.
7. The solutions which are found are not generally not optimal.
8. They require robot to obstacle distance and direction information, a more computationally expensive requirement than a simple *yes* or *no* regarding interference.
9. They typically disallow motion very near or along obstacle surfaces, yet docking, parts mating, and other common tasks require navigation near or along the boundary of the safe configuration space.

Hirukawa and Kitamura [66] claim to avoid the deadlocks at local minima by forming a graph in cartesian space of the positions farthest from obstacles. The end effector tries to follow this graph to the goal while the robot links are attracted

toward the lines of the graph. The formation of the graph involves global world knowledge.

Some researchers' efforts to address the local minima problem involve combining the potential fields planning approach with a higher level global planner [65,67-70].

Warren [65, 71, 72] presents several techniques for global path planning using potential fields. One approach is to first choose a trial path and then to modify that path by the addition of intermediate points until it represents an acceptable solution. The intermediate points are found using the potential function. By choosing the trial path as a series of more closely spaced points than the entire global problem, Warren greatly reduces (but does not eliminate) the possibility of being caught in local extrema of the field. Another approach utilizes the penalty function simply to modify the unsafe regions of a trajectory initially proposed by the planner. The result is that the path is modified only where it intersects an obstacle thereby reducing global sensitivity to the local minima problem. Warren illustrates his techniques for several cases: a 2D revolute manipulator, a mobile robot capable of translation only, and a mobile robot capable of translation and rotation.

Munger [70] takes an approach much like Warren's described above in that he divides the global problem into a series of shorter problems which go through some number of safe intermediate points. The idea then is to solve a series of shorter problems which can be combined to yield a global solution. Munger applies his algorithm to a nine degree of freedom manipulator assembling struts to form a tetrahedron. The workcell for Munger's application is relatively uncluttered. Applying this technique to general robot path planning problems is potentially troublesome due to the difficulty in identifying the intermediate points appropriately so as to enable a solution to be found.

Kim and Khosla [73] propose a different means to handle the local minima problem. Their approach uses harmonic function based potential functions with the property that they are free from local minima in a singularity free space. The *panel method*, a tool from computational fluid mechanics, is used to solve the potential flow problem. For point mobile robots this ensures well behaved potential functions which can be solved quickly even with complex and concave obstacles. For nonpoint robots the geometry introduces *structural local minima* which are positions where the robot can no longer safely move along the potential's gradient. Kim and Khosla have applied this method to a bar shaped mobile robot and a 3 dof planar manipulator. They note that it should be possible to extend their technique to 3D problems by using 3D harmonic functions. Their work also illustrates that the local minima problem still persists even with obstacles having simple shape.

Other means of addressing the local minima problem include generalized potentials [74], a Laplacian approach [75], and a local minima free technique for generalized disc obstacles in a generalized sphere world [76].

Faverjon et al [77] address the problem of having the potential function discourage paths near obstacles by basing the potential function on the object approach velocity.

Barraquand and Latombe [78] present an algorithm which is geometrically similar to Glavina's (see Section 2.1.1). Barraquand combines potential functions and graph search techniques to solve problems with a high number of dof. The algorithm builds a graph connecting local minima of a potential function in c-space and searches this graph for sequences which will produce a solution. Local minimum are connected to each other using a Monte-Carlo randomized motion as needed to escape the first local minimum followed by a gradient motion based on the potential function. The local minima graph is searched depth first with random backtracking. The algorithm is complete since, given due computation time, an exhaustive search

would eventually result. Barraquand presents results for a relatively simple problem with a 31 dof manipulator which was solved in 15 minutes. The planner's ability to quickly solve more difficult but practical problems is not demonstrated in [78].

Lozano-Perez [79] present a task-level approach which involves both potential fields and c-space graph search methods. Lozano-Perez solves the pick-and-place problem by decomposing it into nearly independent, computationally feasible, subproblems. The two main subproblems are the grasp locations and approaches thereto (at both the pick and place ends of the motion) and the gross translational motion from the general locality of the pick location to the general locality of the place location. A grasp position is determined by transforming the obstacles at the place location to their equivalently limiting positions at the pick location and searching the resulting c-space for a feasible grasp position. Having determined the grasp points, Lozano-Perez uses a potential fields approach (and some trial and error) to determine an arbitrary free approach/departure point in the vicinity of both the pick and the place locations. The final phase of Lozano-Perez's task planning is then to plan the free motion plan between the departure point and the approach point. This is done using c-space obstacle mapping and includes the assumptions that orientation may be neglected and that the first three robot joints invoke 3D translation. Exhaustive mapping of the resulting 3D c-space is avoided by progressing in 2D slices within that space until a solution is found.

2.1.3 The Human Assisted Approach

The mathematical complexity of a computed complete (even if suboptimal) solution to the general motion problem apparently make it intractable for more than a few degrees of freedom. Humans seem to possess some natural abilities to "see" solutions to many motion planning problems for which computing a solution is still difficult or excessively computationally intensive. It is precisely this apparent human

ability that the human assisted approach to path planning attempts to capitalize on.

In its simplest form, human assisted path planning is accomplished on-line. This usually involves moving the robot using a teach pendant and storing a series of points along a collision-free path. The points can later be re-played in sequence to execute the desired task.

More typically, the human assisted approach employs computer graphics models of the robot and its environment. The user can then perform the motion planning in an off-line graphical manner. It is usually possible to display multiple views to allow the user to detect any potential collisions. More advanced systems can automatically perform the collision checking. Systems which can compute estimated task execution time can also allow the user to search for a very efficient path. As the number of times a particular task is to be repeated increases, the benefits of obtaining a very efficient path become more pronounced.

Some systems presented in the literature which are suitable for the human assisted approach to off-line path planning are presented by Derby [80], Hornick and Ravini [81], Stobart [82], and Han [83].

More recently, advances in telerobotics has produced systems in which people may be employed as on-line path planners. Telerobotics, as described by Weisbin [84], includes three main paradigms of control:

1. *Teleoperation*, in which a human directly controls the remote device in real time

2. *Robotics* - in which the remote device is preprogrammed

3. *Supervisory Control* - in which the human controller gives high level commands which are decomposed and executed by the machine under human supervision.

Human assisted path planning would typically be involved in paradigm (1), whereas

autonomous path planning could be integrated into paradigm (3) to eliminate some of the burden on the operator.

2.2 Path Planning for Cooperating Robots

While they are inherently similar, there are some key differences between motion planning for single manipulators and for cooperating robots. Some of these differences are shown in Table 2.1. These differences are discussed later in Section 4.1.

Single Robot Path Planning	Cooperating Robot Path Planning
Typically relatively large amounts of free space available.	Closure constraint leads to comparatively little free space.
Translations and rotations may often be decoupled.	End effector orientation important for maintenance of feasible configurations.
Task space heuristics often effective for path planning.	Second robot makes effective use of task space heuristics very difficult.
C-space approaches inherently handle multiple arm configurations.	Configuration of second robot must be considered explicitly.

Table 2.1: Single Robot vs Cooperating Robot Path Planning

In comparison to the single robot path planning problem, the cooperating robot path planning problem has thus far received relatively little attention in the research community. Perhaps this is because an efficient exact algorithm for single robot planning is yet to be developed. Nonetheless, several researchers have specifically considered the cooperating robot path planning problem. Their efforts are summarized below.

Chien [85] presents a path planning technique for two cooperating planar robots each having two links and three revolute joints. Chien's solution process

involves dividing the subspace into two 2D subspaces, one for each of the two possible configurations of the second robot given a specified configuration of the first. These two subspaces are connected by transition configurations for which the configuration in each of the two subspaces is the same. The "same turn first" strategy, an algorithm which guarantees finding a solution if one exists, is used to search for a sequence of moves within and between the two 2D subspaces which will connect the start and goal configurations. While this technique is complete, its practicality is apparently limited to planar robots.

Koga and Latombe [86] present a complete planning technique for cooperating arms with only a few degrees of freedom. The technique is based upon systematic searches of c-space grids. They present another planner which is not complete but is practical for more dof. This technique uses randomized potential fields planning techniques similar to Latombe's prior single arm work [78] discussed in Section 2.1.2. The technique has been implemented for redundant planar manipulators. Unlike other research discussed herein, Koga and Latombe allow the the grasp positions of the robots to be altered during a manipulation by temporarily halting motion of the payload and repositioning an end effector. Thus far, their potential fields planner requires some assumptions which significantly affect the planner's reliability. Difficulty was also experienced with more than a few obstacles.

An analytical technique for single robot path planning involves the use of kinematic constraints to pose the path planning problem as an analytical optimization problem. Seereeram and Wen [87] present an example of such a technique by posing the path planning problem as a finite time nonlinear control problem and solving it using a Newton Raphson type algorithm. This approach represents the requirement of obstacle avoidance with a set of inequalities on the configuration variables. Such approaches are still under development and may prove useful in the future for solving practical problems for robots with many dof. Lim and Chyung [88] apply

a similar technique to the cooperating robot path planning problem by reformulating the problem as a non-linear optimization problem. Their methodology essentially involves determining an admissible trajectory for the object being grasped, where admissibility involves reachability by both robots. This method determines a feasible path by employing optimization methods to modify the cartesian straight line/constant rotation path of the object. Since the feasibility of an object path is investigated at the joint level, the resulting solution is in joint space. No provisions are made for collision detection or obstacle avoidance. Lim presents results for determination of a simple trajectory for two cooperating five degree of freedom RHINO robot arms. It is unclear whether Lim's methodology would be applicable to more difficult problems requiring obstacle avoidance and arm configuration changes.

Hu [89] presents an approach to control multiple cooperating redundant manipulators. While control rather than path planning is Hu's primary concern, the approach allows use of the redundancy to avoid collisions between the robots and obstacles while traversing a specified task space trajectory. Determination of a suitable task space trajectory for the payload would still require some type of higher level path planner.

McCarthy and Bodduluri [90] examine the design and motion planning problem for closed kinematic chains such as cooperating robots. Their motion planning algorithm utilizes selective mapping of c-space and involves subdividing c-space into hypercubes until a safe path may be found by traversing edges of the hypercubes. A 2D depiction of this algorithm is given in Figure 2.5. Figure 2.5a shows a bounded 2D space, some circular obstacles, and prescribed start and goal points (S and G, respectively). The space is subdivided at the start point (Figure 2.5b), and further subdivided at the goal point (Figure 2.5c). Finally, all non-empty regions with reachable vertices are subdivided until a solution is found (Figure 2.5d). This type of approach is referred to as *cell decomposition*. McCarthy and Bodduluri solve

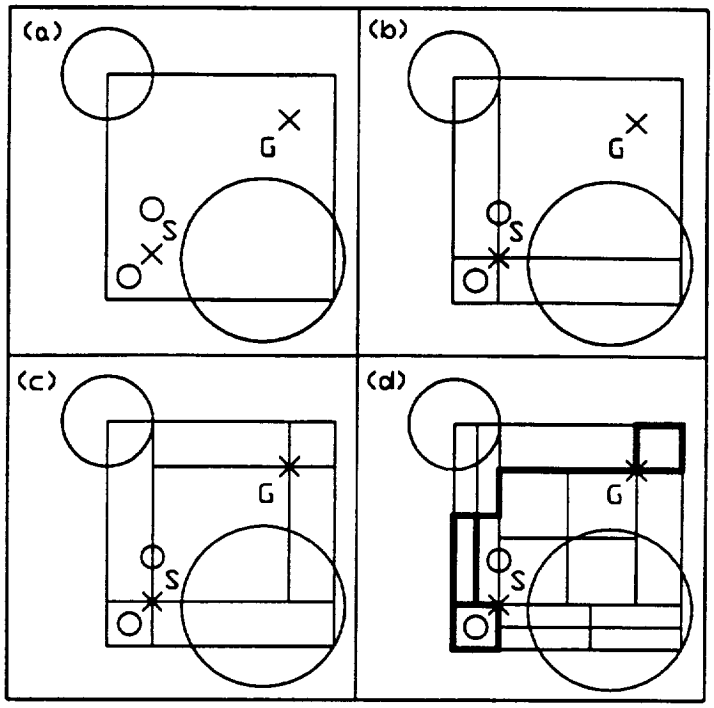


Figure 2.5: Hypercube Subdivision Algorithm

the cooperating Puma problem for several cases for which maintaining closure and avoiding collisions between the robots appear to be the main concerns. The closure constraint utilized is simplified by modeling each puma as a 3R-1S robot and then requiring a constant length between the S joints of each robot.

Chen and Duffy [91] also present a path planner for two cooperative Puma robots. Their approach is to find a feasible position for the first three links of one of the robots along a discretized path from start to goal. For each point along this discretized path the possible closure configurations (cones) are investigated to find a feasible and collision free configuration for the second robot. Because of some simplifications and assumptions it does not appear as though their approach would be successful for problems much more difficult than the relatively simple example illustrated in [91].

2.3 Other Related Areas of Research

Other related areas of path planning research which will not be discussed in depth in this thesis include:

- Mobile robot path planning
- Coordination of multiple robots
- Piano Mover's problem
- Nonholonomic motion planning

These areas of research are briefly discussed below.

2.3.1 Mobile Robot Path Planning

While all robot path planning problems have inherent similarities, mobile robot path planning differs in many ways from path planning for general manipulators. Some of the key differences as identified by McKerrow [92] are summarized in Table 2.2. These differences result in path planning for manipulators being more complex than path planning for mobile robots. The path planning problem for a 2D mobile robot in the presence of known stationary obstacles has many real-time optimal (minimum time or minimum distance) solutions. Many researchers of the mobile robot path planning problem have also considered dynamic obstacles and/or unknown environments. Such results are made possible by the limited dimensionality of the mobile robot path planning problem. Since we are concerned with path planning for manipulators, no detailed discussions will be given to path planning techniques suitable only for mobile robots. Areas where the algorithms used to solve mobile robot path planning problems may impact the general manipulator path planning problem have been included in earlier discussion.

Mobile Robot Path Planning	Manipulator Path Planning
Movement restricted to 3D.	End effector may move in 6D.
Whole robot moves from start to goal.	End effector and payload move from start to goal.
Robot typically occupies a fixed volume.	Volume occupied by robot changes as joints move.
Model of environment typically incomplete.	Exact location and description of objects in the workspace are typically known.
Dead-reckoning control of a mobile robot is subject to significant errors which accumulate.	Typically assume high accuracy and repeatability of joint motions.

Table 2.2: Mobile Robot vs Manipulator Path Planning

2.3.2 Coordination of Multiple Robots

Coordination of robots is typically done assuming the individual paths of the robots are known with the timing to be determined so as to avoid collisions. Research into the coordination of multiple robots will not be discussed herein since it does not appear that cooperating robot path planning research will benefit directly from it at this time.

2.3.3 Piano Mover's Problem

As mentioned earlier, the nature of the robot path planning problem is very similar to the piano mover's problem. The piano mover's problem involves planning a collision free path between two poses for a single, rigid object amongst obstacles. Because of the inherent similarities between manipulator path planning and the piano mover's problem, many algorithms such as vgraphs, voronoi diagrams, and graph search methods may be applied to either. Earlier discussions include such algorithms. There are also a number algorithms which are specific to a particular subset of mover's problems and are not applicable to the robot path planning

problem.

A recent survey of the status of motion planning research including the mover's problem is provided by Hwang et al [93]. Hwang suggests that, as a result of problem complexity, future research should concentrate on heuristic algorithms that run in a few seconds at the expense of failing to find a solution to very hard, pathological, puzzle-like problems.

2.3.4 Nonholonomic Motion Planning

The complexity of a certain class of motion planning problems is compounded by *nonholonomic* constraints. Nonholonomic constraints are constraints on the derivatives of configuration variables which cannot be integrated. For example, a unicycle may maneuver to achieve any position and orientation, but its direction of motion at any one instant is constrained. Path planning for single and cooperating robots is holonomic. The nonholonomic problem is much more difficult and efforts for developing implementable algorithms are just beginning. A review of the current status of motion planning with nonholonomic constraints may be found in [93].

2.4 Summary of the Literature Review

This section presents a summary of the above literature review. The summary is presented in four sections:

- Difficulties with Complete Solutions
- Practical Incomplete Solutions
- Potential Fields Solutions
- Cooperating Robots

2.4.1 Difficulties With Complete Solutions

Many complete algorithms have been developed for solving the motion planning problem. However, it appears as though the mathematical complexity of such techniques renders them computationally intractable when applied to a reasonably difficult robot motion planning with six or more dof. A general, practical, and complete solution to the motion planning problem has not yet been developed.

There are a number of complete approaches which attempt to achieve solution time commensurate with problem difficulty. The computational practicality of these techniques for reasonably difficult yet practical path planning problems remains to be demonstrated.

2.4.2 Practical Incomplete Solutions

As a result of problem complexity, *practical* techniques used to solve the single robot motion planning problem for six or more dof involve some heuristics or simplifying assumptions and lack completeness. Some typical simplifications include:

- Simplified models of the robots and obstacles
- Decoupling of rotations from translations
- Compact wrists and payloads
- Restrictions on allowable motions and allowable obstacles

These simplifications and heuristics are typically robot and/or task specific and would not be expected to perform well in more general cases or for two robots working cooperatively due to the differences presented earlier.

The speed and success of the most useful algorithms can be attributed to their pruning of the search space by reducing problem dimensionality or by their ability to selectively map c-space thereby avoiding intractable exhaustive mappings.

2.4.3 Potential Fields Solutions

The potential fields approach to single arm path planning constitutes an effective way to combine the constraints resulting from several obstacles for many simple cases, but the fact that motion planning using potential fields is based solely on local information has led to some difficulty in achieving effective high level planning. The most effective potentials fields approaches determine a sequence of intermediate via points between which there are no local minima.

2.4.4 Cooperating Robots

Of the work which has been published for path planning of cooperating robots, much of it is limited in effectiveness to planar systems. The researchers who have addressed cooperating robots with six or more degrees of freedom have apparently been successful only in solving problems which appear to be relatively simple.

Research pertaining to path planning for cooperating robots utilizing potential fields appears to be still in its early stages. Results so far have been limited to redundant planar systems with only a few obstacles.

CHAPTER 3
Statement of the Problem

This chapter presents a formal definition of the robot path planning problems being addressed by this thesis. Some general background information is given in Section 3.1. Sections 3.2 and 3.3 discuss assumptions and goals, respectively. Formal definitions of the single and cooperating robot path planning problems are given in Sections 3.4 and 3.5, respectively.

3.1 Background

A robot can be described by its *forward kinematic equation*

$$\mathbf{T}_0^m = f(\Theta) \tag{3.1}$$

where $\mathbf{T}_0^m \in \mathcal{R}^m$ represents the task space transformation (position and/or orientation) of the end effector and $\Theta = (\theta_1, \ldots, \theta_n) \in \mathcal{R}^n$ represents the robot's joint configuration, where n is the number of degrees of freedom (dof). For spatial robots with three translational and three rotational dof, $m = 6$.

A robot's *inverse kinematic equation*

$$\Theta = f(\mathbf{T}_0^m) \tag{3.2}$$

identifies joint configurations Θ which would result in a specified task space transformation $\mathbf{T}_0^m$. For a non-redundant robot capable of achieving any desired position with any desired orientation (within workspace limits), $n = m$, and Equation 3.2 will possess only a finite number of solutions Θ for a given $\mathbf{T}_0^m$. For redundant robots $n > m$ and equation 3.2 is underdetermined, indicating that an infinite number of robot configurations Θ exist which produce the end effector transformation $\mathbf{T}_0^m$. The problem of solving Equation 3.2 for a redundant robot is referred to as the

redundancy resolution problem. A robot with $n < m$ has fewer dof than required to arbitrarily position and orient its end effector in the workspace. The inverse kinematic equation for such a robot is overdetermined, i.e., it will have solutions only for transformations which lie in the limited workspace of the robot.

3.2 Assumptions

This section restates the assumptions presented in Subsection 1.2.1 and provides a discussion regarding each assumption.

Assumption 1 *Forward kinematic models of the robots are available.*

Discussion: A robot may be represented using the Denavit-Hartenberg convention from which the forward kinematic model (Equation 3.1) can be easily derived [94].

Assumption 2 *Closed-form inverse kinematic models of the robots are available for six dof robots or for the final six links of redundant robots.*

Discussion: This thesis addressed full spatial robots for which $n \geq m = 6$ (see Section 3.1). Most commercial six dof robots satisfy one of the following sufficient conditions which enables a closed-form inverse kinematic solution [94]:

1. Three adjacent joint axis intersect.
2. Three adjacent joint axis are parallel to one another.

Unimation Puma manipulators, which will be used in the case studies for this thesis, satisfy the first condition. In general, multiple solutions will exist representing various possible robot configurations. For redundant robots, it is assumed that the final six links can be treated as a single six dof robot for which a closed-form inverse kinematic model is available. The usefulness of this assumption regarding redundant manipulators will become evident later in this thesis.

The path planning strategy in this thesis does not require inverse kinematics for single robot path planning problems.

Assumption 3 *Geometric models of the robots, payload, and obstacles are available.*

Discussion: Robots and their environment may be represented by some form of *geometric model.* Some typical forms of geometric modeling include boundary representations (b-reps), constructive solid geometry (csg), and polytope representations. The geometric model will contain knowledge of the geometry, position, and orientation of the robot links, the payload, and each obstacle in the workcell. The only constraint regarding geometric modeling is that a facility for performing collision detection is required. Neither the source of this geometric information nor the data structure format of the geometric model is important from the perspective of the path planner. For static obstacle path planning purposes, the geometric model need only consist of a geometric description of the robots, payload, and objects in the environment.

Assumption 4 *Obstacles in the workspace are static.*

Discussion: The added complexity of a dynamic environment make it unlikely that a practical planner for cooperating multi-dof robots with dynamic obstacles will be developed anytime soon.

Assumption 5 *Feasible and collision free start and goal joint configurations of the robots are known, as are the start and goal positions of the payload.*

Discussion: There are several key consequences of this assumption. First, note that the grasp positions are inherently defined by this assumption. The determination of suitable grasp positions is highly task specific, potentially very difficult,

and beyond the scope of this thesis. Secondly, note that specifying the start and goal *joint* configurations as opposed to the start and goal *task space* configurations eliminates the need for the path planner to choose particular solutions to the inverse kinematics at the start and goal positions. It is reasonable to assume that the start joint configurations are known since some single arm planning must have been done to position the robots at their initial positions. Requiring that the goal joint configurations be known is more demanding than simply specifying a task space goal for the payload. Typically even non-redundant robots would have several possible configurations (such as elbow up or elbow down) which satisfy a particular task space goal. The solvability of the path planning problem can depend upon which joint configuration is specified as the goal. An example where the choice of goal joint configurations determines the solvability of a path planning problem is illustrated in Figure 3.1. Figure 3.1a shows the start position for two cooperating 3R planar

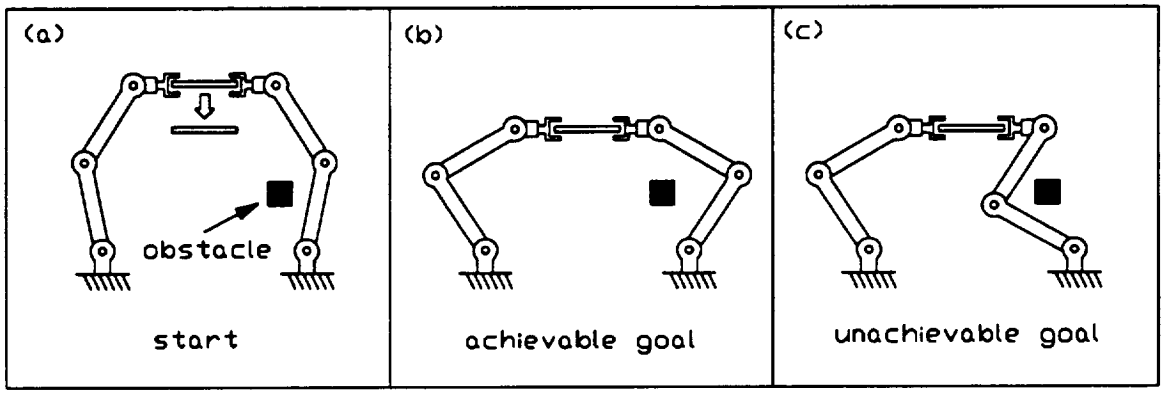


Figure 3.1: Choice of Goal Joint Angles May Affect Solvability

robots. Figure 3.1b shows a choice of goal joint configurations which result in a solvable problem for the case illustrated. As shown in Figure 3.1c, a different choice of goal joint configurations which produce the same task space goal can result in an

unsolvable problem. In the case of redundant robots some form of redundancy resolution is required to specify the goal joint angles. Redundant robots will typically possess one or several regions in c-space which yield a desired task space goal.

It is a clear disadvantage to require the goal joint configurations be specified at the outset of the problem since this information must come from some higher level source and may directly determine the existence of a solution. However, a few incidental advantages arise from the extra knowledge required by Assumption 5:

- *Path cyclicity concerns are eliminated.* A path planner will often be required to execute a task which is repetitive in task space. Path planners which do not specify the start and goal joint angles for a particular path planning problem often suffer from *path cyclicity* problems whereby the robot does not achieve the same configuration on subsequent repetitions of identical task space tasks.

- *Path planning problems may be attacked either from start to goal or vice versa.* The ability to attempt to solve a path planning problem from either direction (or even from both directions simultaneously) may prove to be beneficial if the algorithm or heuristic being used happens to be more successful in one direction than in the other for a particular path planning problem. For example, planning a path to remove a peg from a hole would intuitively seem much simpler than planning a path to put the peg in the hole. The 2-D problem illustrated earlier in Figure 2.2 is one which would have proven easier to solve backwards if using an A* graph search approach. As discussed earlier in Section 2.1.1, the ability to search bidirectionally is often valuable.

- *A preferred goal robot configuration may be achieved.* In some cases it may be desirable to supply the path planner with a specified goal robot configuration rather than allowing the path planner to choose any which satisfy the goal position/orientation in task space. For example, a reliability analysis or

robot flexibility analysis might be used to prescribe a preferred goal robot configuration.

Our need for Assumption 5 stems from the fact the our approach is configuration space based. This will become clear as our solution technique is presented later in this thesis.

Assumption 6 *Motion between the specified start and goal positions may be arbitrary.*

Discussion: This assumption illustrates that interest is solely to move from start to goal without restriction on the path. This is the most general form of the path planning problem and is acceptable for solving the vast majority of problems. As an example of a task for which this assumption would not be valid, consider two robots cooperatively manipulating a trough of water. Clearly such a task would impose a constraint on the motion between the start and goal positions such that the trough would remain level so as not to spill the water. Another example requiring restricted motion involves contact between the robot/payload and its environment. Although such cases are not considered herein, some discussion of how they might be addressed is presented later in Section 5.4.

In cases where a specific task space path must be followed the problem becomes one of configuration selection or redundancy resolution rather than a classical path planning problem. For example, a nine dof robot performing arc welding along a specified task space path is not a nine dimensional path planning problem but rather a much simpler three dimensional redundancy resolution problem.

Assumption 7 *The planner may ignore robot dynamics.*

Discussion: This assumption is valid when considering only static obstacles and since a time optimal trajectory is not sought. Algorithms which consider

dynamics typically assume that an initial path is given and dynamic optimization is done locally along the path [95]. Under dynamic optimization, path curvature becomes an important characteristic.

3.3 Goals

This subsection restates and discusses the goals presented in Subsection 1.2.2.

Goal 1 *The planner shall locate reasonable collision-free paths in a time frame suitable for off-line path planning.*

Discussion: It appears as though the search for an optimal path and/or a real time solution for non-trivial path planning problems with more than a few dof will remain computationally intractable for some time to come (See Chapter 2).

Goal 2 *The planner shall be capable of modifying a feasible path into a more efficient one.*

Discussion: It is typically possible to modify a suboptimal path found by a path planner to produce a smoother, more efficient path.

Goal 3 *The planner shall be applicable to various robotic systems and various tasks.*

Discussion: Some path planning techniques perform well only with specific types of robots or for certain types of tasks due to their use of simplified, case specific assumptions or heuristics. We would like our solution technique to remain free of any assumptions which would limit its use as a general-purpose path planner.

Goal 4 *The planner shall be practical for cooperating six dof manipulators as a minimum, and ideally for cooperating redundant robots.*

Discussion: It should be noted that the practicality of a path planning technique for a robot with six or more dof is important since at least six dof are required to arbitrarily position and orient an end effector. Many of the path planning techniques discussed in Chapter 2 are not practical for robots with six or more dof.

Goal 5 *The planner output shall be a sequential listing of closely spaced knot points in joint space which represent the discretization of a continuous, feasible, and obstacle avoiding path connecting the start and goal configurations.*

Discussion: This goal is consistent with integrating a path planner into the CIRSSE testbed system using a traditional three-step decomposition of the *move robot* problem. The three steps are path planning, trajectory generation, and motion control. A *trajectory generator* may be used on the output of the path planner to provide timing information consistent with the dynamic constraints of the system. The knot points determined by the path planner shall be spaced closely enough that the trajectory generator need not be concerned with maintaining the closure requirement between knot points. Execution of the time parameterized trajectory may be carried out by a motion control system. Some fine compliance will typically be required due to inaccuracies in the robot model or tracking errors at the control level. Such compliance could be either passive, such as a compliant end effector, or active, such as compliance based on force/torque feedback. Details of the trajectory generation and motion control steps are separate areas of research which are beyond the scope of this thesis.

3.4 Single Robot Path Planning Problem Statement

Per the background and assumptions stated above, the single robot path planning problem may be formally defined as follows:

Given:

1. A single robot described by its forward kinematic equation, Equation 3.1.
2. Geometric models of the robot, the payload, and workspace obstacles.
3. Start and goal joint configurations of Θ_s and Θ_g, respectively.

Determine:

A closely spaced sequence of k joint space knot points $(\Theta_1, \ldots, \Theta_k)$, where $\Theta_1 = \Theta_s$ and $\Theta_k = \Theta_g$, which represent a discretization of a feasible and collision free c-space path connecting Θ_s and Θ_g.

3.5 Cooperating Robot Path Planning Problem Statement

Per the background and assumptions stated above, the cooperating robot path planning problem for two cooperating spatial robots, referred to as robots 1 and 2, may be formally defined as follows:

Given:

1. Two robots work cooperatively satisfying the closure constraint:

$$\mathbf{T1}_0^6 \mathbf{T}_{r1}^{r2} = \mathbf{T2}_0^6 \tag{3.3}$$

   where $\mathbf{T}_{r1}^{r2}$ is an invariant transformation relating the relative positions of the robot end effectors as they grasp a common, rigid object.

2. The robots are described by forward kinematic equations:

$$\mathbf{Ti}_0^6 = f(\Theta \mathbf{i}) \quad , \text{i=1,2} \tag{3.4}$$

where $\mathbf{\Theta i} = (\theta i_1, \ldots, \theta i_{n_i})$ represents robot i's joint configuration, $n_i \geq 6$ is the number of degrees of freedom (dof) of robot i.

3. The robots are described by inverse kinematic equations with *at most one* solution:

$$\mathbf{\Theta i} = f(\mathbf{Ti}_0^6, \mathbf{\Theta i'}, Ci_{j_i}) \quad \text{, i=1,2} \tag{3.5}$$

where $\mathbf{\Theta i'} = (\theta i_1, \ldots, \theta i_{n_i-6})$, and Ci_{j_i} represents one of j_i possible robot configurations for robot i, and j_i is finite and known.

4. Geometric models of the robots, the payload, and workspace obstacles.

5. Start and goal joint configurations of $\mathbf{\Theta i}_S$ and $\mathbf{\Theta i}_g$, respectively, where $i = 1, 2$.

Determine:

A closely spaced sequence of k paired joint space knot points $((\mathbf{\Theta 1}_1, \mathbf{\Theta 2}_1), \ldots, (\mathbf{\Theta 1}_k, \mathbf{\Theta 2}_k))$, where $\mathbf{\Theta i}_1 = \mathbf{\Theta i}_S$ and $\mathbf{\Theta i}_k = \mathbf{\Theta i}_g$, which represent a discretization of a feasible, continuous, and collision free path connecting $(\mathbf{\Theta 1}_S, \mathbf{\Theta 2}_S)$ and $(\mathbf{\Theta 1}_g, \mathbf{\Theta 2}_g)$. Each paired knot point $(\mathbf{\Theta 1}_j, \mathbf{\Theta 2}_j)$ shall satisfy the closure constraint, Equation 3.3. Also, the discretization shall be sufficiently fine so that a trajectory planner may ignore the nonlinearities of the closure constraint between knot points.

CHAPTER 4
Divide-and-Conquer C-Space Traversal Heuristic

This chapter presents the configuration space traversal heuristic which is the heart of the path planning strategy presented in this thesis. This chapter merely presents the heuristic. The utilization of the heuristic is discussed in subsequent chapters. This chapter is organized into eight main sections:

- Motivation for a New Approach
- Conceptual Description of Heuristic
- Vector Description of Heuristic
- Computing Search Directions
- Prioritizing Search Directions
- Comparison of the Heuristic to the Literature

Section 4.1 discusses the motivation for a new path planning technique for cooperating robots. Sections 4.2 and 4.3 present conceptual and vector descriptions of the c-space traversal heuristic, respectively. Computation and prioritization of search directions used by the heuristic are discussed in Sections 4.4 and 4.5, respectively. Finally, a comparison of the heuristic to published path planning algorithms and heuristics is presented in Section 4.6.

4.1 Motivation for a New Approach

This section attempts to make a case that there is sufficient motivation for this new research in the area of path planning for cooperating robots. First, recall from Section 2.2 that path planning approaches in the literature for cooperating robots

are generally limited with regard either to the number of degrees of freedom (dof) of the robots or to the apparent difficulty of problems which they are capable of solving. Thus, there appears to be sufficient motivation for this research.

Due to the fact that researchers' interest in cooperating robotic systems is relatively young compared to the much longer history of interest in single robots, thorough consideration should be given to the application of methods developed for single robot path planning when searching for a solution to the cooperating robot path planning problem. There are, however, some unique elements to the general cooperating robot path planning problem that make it unlikely that any of the single arm path planning methodologies discussed in Chapter 2 could be successfully applied to cooperating robots without significant modifications. These differences were presented earlier in Table 2.1. Some of these special elements of the cooperating arm problem and the way in which they impact the solution process are discussed in this section.

Consider, for instance, two cooperating six degree of freedom manipulators. The effective number of degrees of freedom for the closed kinematic chain is six (from Equation 1.1). Hence, the problem is essentially six dimensional (almost as if it were a single arm problem) but possesses the added closure constraint. This restriction does not affect the dimensionality of the space in which a graph search algorithm must perform, but does affect the validity of some of the assumptions typically used to reduce the system to one of a lower dimensionality. For example, a common assumption for single arm planning is to neglect orientation for large, gross moves through space. This assumption would not likely prove effective for two cooperating robots since the orientation of the load will usually be crucial to the maintenance of configurations reachable by both robots.

The added difficulty induced by the closure constraint would also make it extremely difficult to implement a planner based on task space heuristics. One

of the difficulties with task space based heuristics for single robot path planning problems is that they often produce collisions with one obstacle while trying to avoid another. Such difficulties could only be more severe for a closed kinematic chain such as results during robotic cooperation. An additional difficulty which would be magnified by the reduction in free space during cooperation is the fact that the avoidance strategy suggested by a task space heuristic may not always be feasible to achieve.

Although the potential fields method should, in theory, be applicable to the cooperating robot motion planning problem, much difficulty in achieving a reliable implementation would be anticipated. Much thought would be required to attempt to develop potential field functions that would be well behaved for the closed kinematic chain which results during cooperation. Also, the practice of selecting a grid of trial points and perturbing them or rerouting the path through a different set of via points would be significantly more difficult for cooperating robots than for a single robot. The basis for the preceding statement is that a far more restricted safe space results for cooperating arms. As a result, the practice of determining safe trial points more closely spaced than the overall global problem would be more difficult. Also, there would be increased likelihood that some intermediate trial points would lie in unreachable regions of safe space. Results in the literature seem to support the premise that achieving a practical and reliable potential fields based planner for cooperating robots would be difficult (see Section 2.2).

The human assisted approaches still maintain their advantage of capitalizing on the natural ability of humans to solve complicated geometric problems. In fact it is the human assisted approach by which most non-trivial collision free robot motion planning is currently accomplished. However, the level of insight which the user would be required to supply would clearly be much greater for two cooperating

arms than for a single arm. This increase in difficulty may make an already potentially undesirable task for a human prohibitively tedious, frustrating, and difficult. In addition, while the human assisted approach offers the best chance for nearly immediate results, it is contrary to our longer term goals of creating more autonomous robotic systems capable of complete task planning and execution from a task level command.

The path planning procedure being presented herein is of the graph search type and, in a fashion similar to Dupont's approach to path planning for a single redundant manipulator (see Section 2.1.1), the procedure involves selective mapping of c-space on an as needed basis to reduce computational burden. Because of the added difficulty of the cooperating arm problem, an improved heuristic was sought to guide the mapping of c-space in a manner directed towards finding a solution with a minimal amount of mapping. This resulted in the development of the "divide-and-conquer" c-space traversal heuristic presented below.

4.2 Conceptual Description of Heuristic

In this section, a novel "divide-and-conquer" style heuristic is presented for traversing an n-dimensional space consisting of safe and unsafe regions. For purposes of robot path planning, the space to be traversed is c-space. The heuristic is general in nature and, while our intended application is to solve the robot path planning problem, this technique could be used to attempt to traverse any space consisting of regions of safe and of unsafe points. An example of another possible application is the "piano movers' problem." Because of the impracticality of mapping the space exhaustively for dimensionality greater than perhaps three, the heuristic was formulated to be compatible with selective mapping of c-space with no computationally expensive precomputations. The c-space traversal heuristic is the

"backbone" of the path planning technique being presented in this thesis. Discussion of the application of the c-space traversal heuristic to the robot path planning problem is deferred until the next Chapter.

This section describes the heuristic conceptually using several simple 2D and one 3D illustrative examples. A vector description of the heuristic is given in Section 4.3. Although the pictorial examples herein are mainly 2D for simplicity of illustration, the approach suffers no loss of generality regardless of problem dimensionality (although the complexity of the searches increases with problem dimensionality). The vector description presented later is applicable to a space of arbitrary dimension.

To illustrate the heuristic, consider the 2D path planning problem illustrated in Figure 4.1a, where Θ_s and Θ_g are the start and goal positions, respectively. The following note is important:

> In this example and subsequent examples herein the boundary of the unsafe c-space is defined in the figure as though the c-space obstacle has been mapped out. This is not the case, but the entire unsafe region is shown a priori to provide better understanding of the subsequent steps.

First, the n-dimensional direction vector from the start point to the goal point is calculated and an attempt is made to traverse along that vector until the first unsafe point is found. This involves discretizing the path from the start to the goal and mapping each successive step along that path until the first unsafe point is found. In the example, the progression from Θ_s is safe through point Θ_a (Figure 4.1b). Points safely mapped are indicated by the solid circles in the Figure.

Next, the progression along the straight line path from start to goal is continued through the unsafe region until the first safe point is found. In the example, this first safe point is labeled Θ_b in Figure 4.1b. Unsafe points mapped in this process are indicated by the open circles. Although in this example Θ_b lies in the

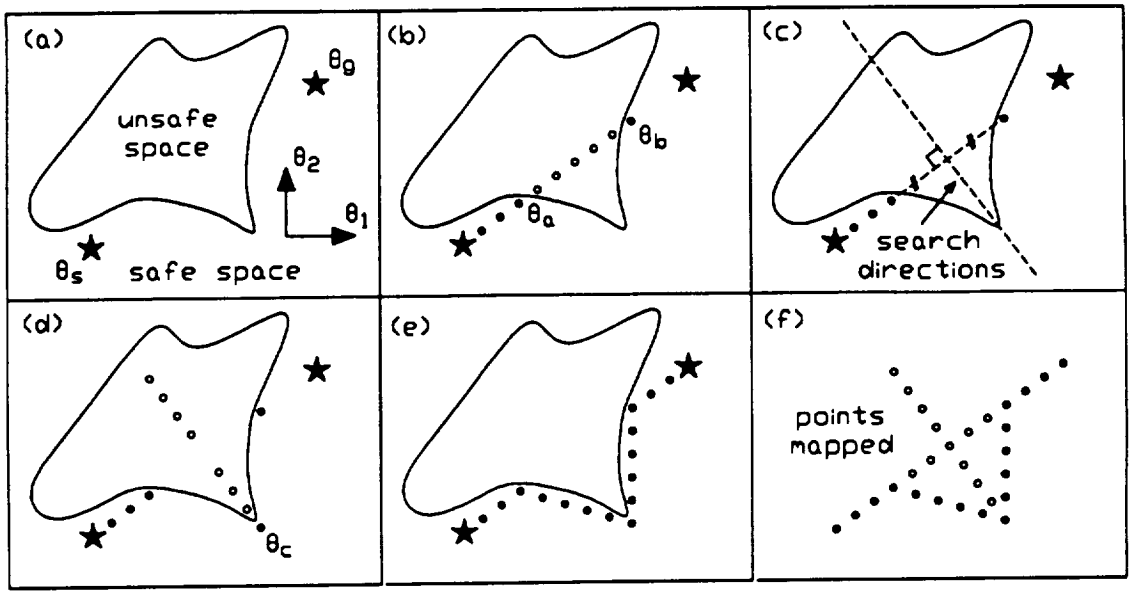


Figure 4.1: 2D Example of C-Space Traversal Heuristic

same connected region of safe space as the start and the goal points, this will not be true in general. Next, the intent is to find a safe point in the $n-1$ dimensional space orthogonal to and bisecting the vector between the last safe point (Θ_a in the example) and the first safe point on the other side of the homogeneously unsafe region (Θ_b in the example). It is apparent that such a safe point must exist if the problem at hand is solvable. In this example, this reduces to searching the 1D line shown in Figure 4.1c. The search methodology depends upon whether this is an initial search or a subsequent search:

- For an *initial* search, the search space is effectively searched for the safe point nearest to the midpoint of the unsafe line segment which was mapped previously. This is done by radiating out equal amounts in all search directions until a safe point is found.

- For a *subsequent* search, the search directions are prioritized and searched non-uniformly per the methodology discussed in Section 4.3. In 2D, a prioritized search would first search in the search direction which has a component in the direction of the previously successful search direction. If no safe point can be found in that direction, the opposite direction is searched.

Since this is the initial search in the example, the line is searched discretely and in both directions equally from the midpoint until safe point Θ_c is found (see Figure 4.1d). Next, an attempt is made to traverse to the safe point from the last safe point initially found in trying to go directly from the start to the goal (that point being Θ_a in the example). The following steps depend upon the result of that attempted traversal, as detailed by the following two cases:

Case 1: The Traversal to the New Safe Point is Entirely Safe

In this case it is attempted to traverse to any previously determined *guide* points, where guide points are previously determined safe points such as those found at the other side of the homogeneously unsafe region or intermediate goal points found in any prior searches. The sequence for considering the guide points is the opposite of the order in which they were found with the global goal point to be considered as a final guide point. When progression to a particular guide point is not entirely safe, that guide point is permanently dismissed and progression is attempted toward the next guide point in the specified sequence. It is in this manner that productive use may be made of safe points which could be in unreachable regions of safe space. As a result, intermediate guide points may or may not be part of the final path. The attempted progressions continue until an attempt has been made to progress to the global goal point. If progression can be made to the global goal point the entire path planning problem has been solved. Otherwise, the last safe point progressed to becomes the new start point and the entire heuristic is repeated until the global goal point has been safely progressed to.

Note that only points which have been safely progressed to from the start point are mandatorily included as part of the final path but those which may be in unreachable regions are used to help guide the overall process. All points actually comprising part of the path will, of course, be in the same connected region of safe space as the start point.

The 2D example of Figure 4.1 invoked this case since the attempt to traverse from Θ_a to Θ_c can be seen to be successful (Figure 4.1e), after which progression is made to guide points Θ_b and Θ_g thereby completing this simple 2D path planning problem with the resulting path shown in Figure 4.1e. The c-space points which required mapping during the process are shown in Figure 4.1f. Note how relatively few points were mapped by this technique.

Case 2: The Traversal to the New Safe Point is Not Entirely Safe

In this case the heuristic is recursively applied taking the last point safely progressed to as the start point and the safe point found in the last search as the goal point.

4.2.1 More 2D Examples

Another 2D illustration of the heuristic is given in Figure 4.2. The solution sequence in this example is similar to that in the previous example except in this case, following the safe traversal to the safe point Θ_c, no progression can be made toward Θ_b. Thus Θ_b is disregarded, progression is attempted toward the second guide point Θ_g resulting in the solution shown in Figure 4.2e. Note that the disregarded point did not necessarily have to lie in the same region of safe space as the start and goal positions (although it did in this example).

An example of a 2D task which would result in a c-space having an unreachable safe region is shown in Figure 4.3. A 2D illustration of the c-space traversal heuristic for a problem with two disjoint regions of safe space is illustrated in Figure 4.4. This

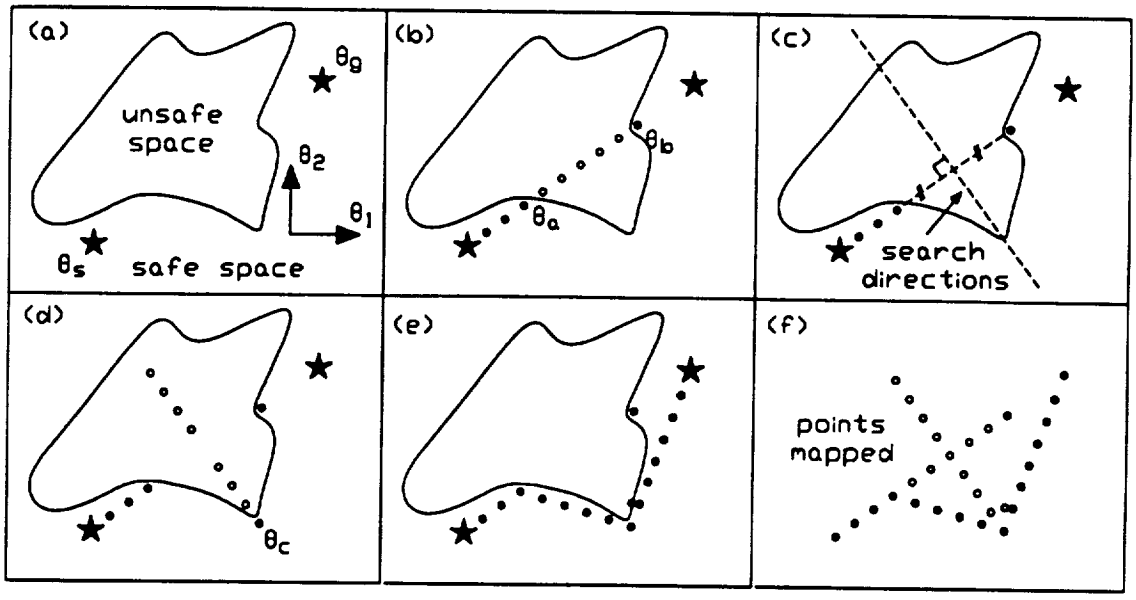


Figure 4.2: Example Which Dismisses an Intermediate Point

illustration also demonstrates the inherent reversal nature of the heuristic when a joint limit problem is encountered (the second search hits a joint limit in the preferred direction after which reversal occurs). This example also illustrates the heuristic for a problem requiring multiple searches.

4.2.2 A 3D example

An example of the c-space traversal heuristic applied to a 3D problem is illustrated in Figure 4.5. In the 3D case, the search space is 2D (planar). For this example eight evenly distributed search directions were considered with the search directions prioritized into two groups (prioritization is discussed below).

4.2.3 Philosophy Behind the Heuristic

The basic idea behind the "divide-and-conquer" c-space traversal heuristic is that better local decisions at the beginning of the trouble region may be made if a

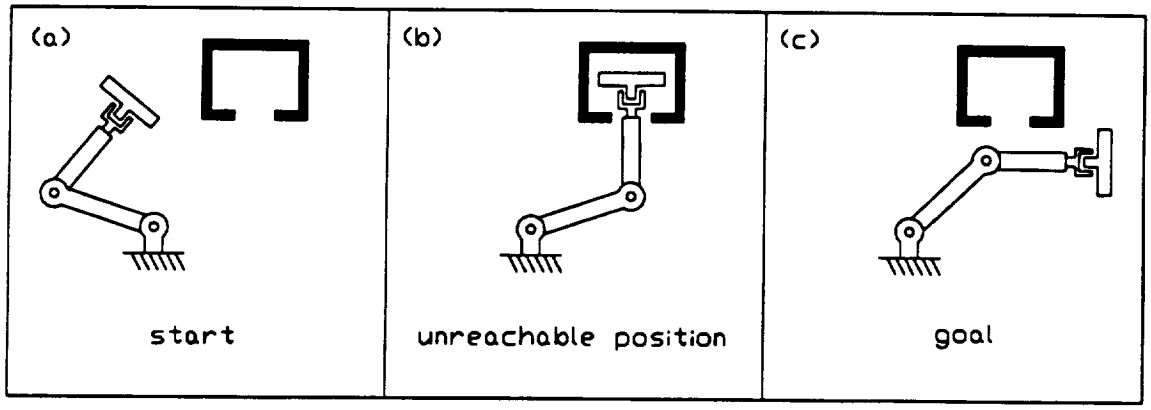


Figure 4.3: Scenario Which Would Result in Non-Disjoint C-Space

possible way around the "center" of the trouble region is known. Thus, rather than attempting paths which look promising locally (at the beginning of a trouble region) but which may not yield overall results, the heuristic attempts local strategies that appear to have a possible overall solution around the trouble region. A comparison of how this heuristic relates to the literature is given later in Section 4.6.

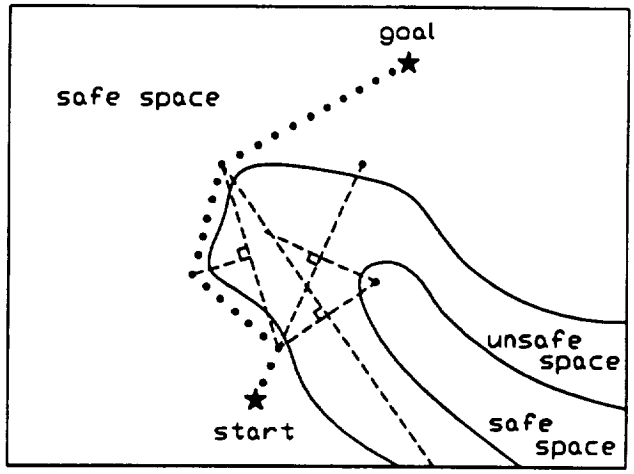


Figure 4.4: Example with Non-Disjoint Safe Space and Multiple Searches

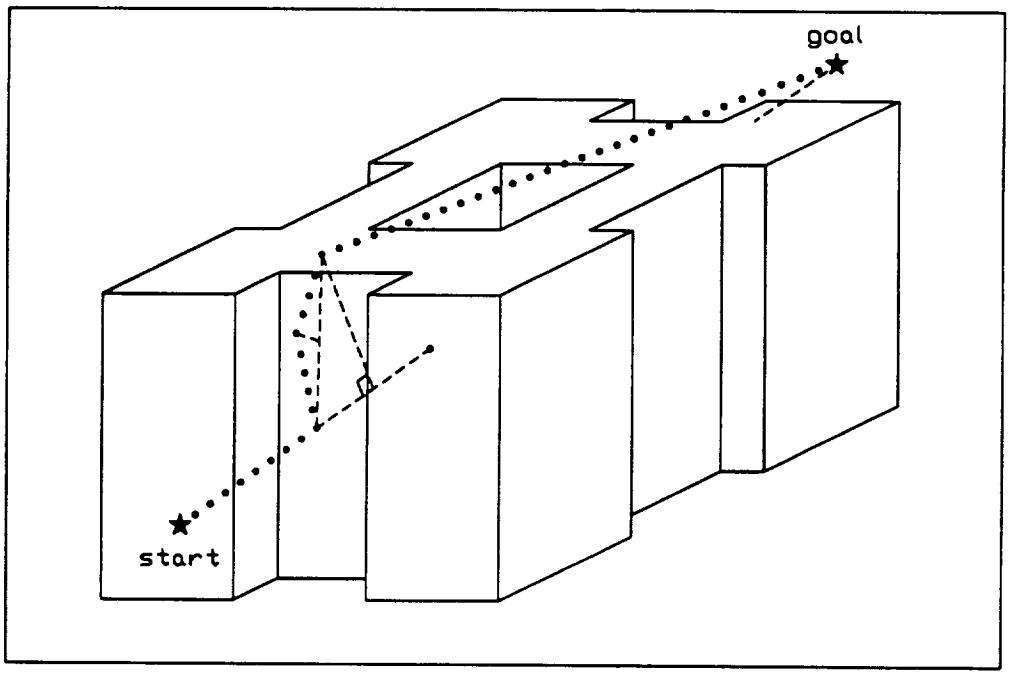


Figure 4.5: 3D Example of C-Space Traversal Heuristic

4.3 Vector Description of Heuristic

Given Θ_s and Θ_g, the start and goal positions in n-dimensional space, respectively, the heuristic may be described in vector notation by the following ten step procedure:

Step 1

Compute the direction vector from start to goal and normalize:

$$\mathbf{D} = \frac{\Theta_g - \Theta_s}{||\Theta_g - \Theta_s||}$$

Step 2

Compute the number of discrete steps along D from start to goal:

$$n = c \,||\, \Theta_g - \Theta_s \,||$$

where c = constant which determines discretization size

Step 3

Discretize from Θ_s to Θ_g in the direction of $\mathbf{D}$ until the first unsafe point is found. Call the last safe point Θ_a:

$$\Theta_a = \Theta_s + j\frac{\mathbf{D}}{c}$$

where j = last integer in 1,2,...,n before an unsafe point is found

Step 4

Continue the discretization through the unsafe region until the next safe point is found. Call that point Θ_b:

$$\Theta_b = \Theta_s + k\frac{\mathbf{D}}{c}$$

where k = first integer in j+2, j+3,...,n which yields a safe point

Step 5

Establish a set of n_{SD} normalized search directions, Θ_{SD_i}, orthogonal to **D**:

$$\Theta_{SD_i} \cdot \mathbf{D} = 0$$

where i=1,2,...,n_{SD} and $\cdot$ represents the dot product operator.

Calculation of search directions is discussed in Section 4.4.

Step 6

If this is a subsequent search, prioritize the search directions by grouping them according to their dot product with the last successful search direction. A technique for so prioritizing the search directions is described in Section 4.5. The number of groups used will affect the emphasis given to continuing searches in the previously successful direction. The purpose of the prioritization is to favor search directions based on their component in the direction of the last successful search direction.

Step 7

Search from the midpoint of the unsafe region, $(\Theta_a + \Theta_b)/2$, in the (possibly prioritized) search directions until a safe point, designated as Θ_c, is found. The search technique shall depend upon whether this is the initial search or a subsequent search.

If this is the initial search, search the entire set of search directions for the safe point nearest to the center of the trouble region by radiating out equal discrete steps in each search direction until a safe point is found or until all directions exceed a joint limit and no safe point has been found.

If this is a subsequent search, search the highest priority group by radiating out equal discrete steps in each search direction in that group until a safe point is found or until it is determined that no safe point can be found in any of those directions (such as a joint limit has been reached in each direction). If no safe point

is found in the highest priority group then repeat for the next highest priority group. Repeat until a safe point is found or until all groupings of search directions have been exhausted and no safe point has been found.

If no safe point could be found, reinitialize the global problem as from the last point safely progressed to the global goal point and restart the entire procedure.

Step 8

Discretize along Θ_a to Θ_c and traverse as far along this segment as is safe. If this entire segment is safely traversed goto Step 9. Otherwise goto Step 10.

Step 9

Progress toward all previous guide points in the opposite order in which they were found, where guide points include not only previous intermediate goal points but also the safe points found on the goal end of each unsafe region which invoked a search. The global goal point is added as a final guide point. When progression to a particular guide point is not entirely safe, that point is permanently dismissed and progression is attempted toward the next guide point in the specified sequence. The progression continues until an attempt has been made to progress to the global goal point. If progression to the global goal point is safe, the global path planning problem has been solved. Otherwise, redefine Θ_s as the last safe point in that progression, Θ_g as the global goal point, and go to Step 1.

Step 10

Set Θ_s equal to the last safe point, and Θ_g equal to Θ_c, and go to Step 1.

4.3.1 Failure Condition

The heuristic fails when a call is made to Step 1 above with identical values of Θ_s and Θ_g as a previous call. This can occur by one of the following two failure modes:

1. Cycling occurs

2. The first search following reinitialization fails to locate a safe point.

A 2D example which results in the first failure mode is shown in Figure 4.6. In spite of the possibility that the heuristic will fail, the results presented later in this

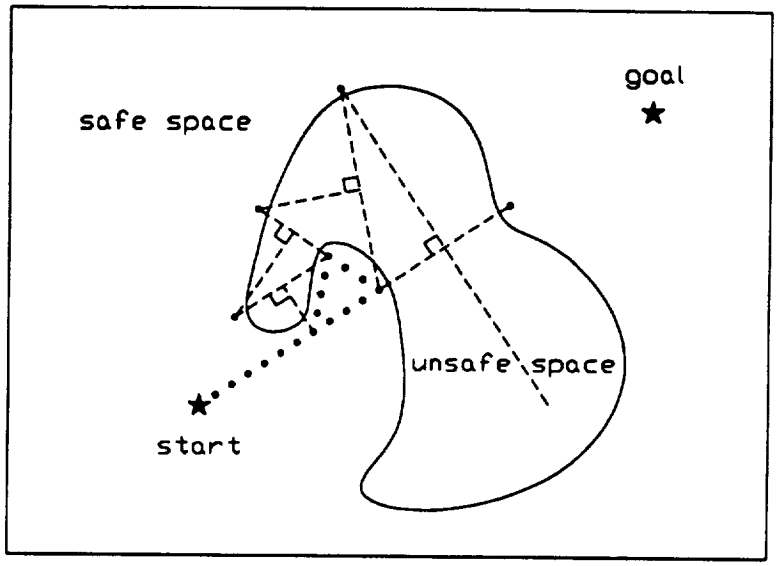


Figure 4.6: 2D Example for which Heuristic Fails by Cycling

thesis seem to indicate that the heuristic provides the capability to solve realistic and potentially difficult path planning problems. The example shown in Figure 4.6 does involve a concave obstacle. The heuristic does appear to perform better with convex obstacles however the complexity and nonlinearity of the task space to c-space mapping makes it unlikely that even simple problems will result in a c-space with strictly convex obstacles. In addition, the ability to attack the problem from either direction (see discussion following Assumption 5 in Section 3.2) would mean that a problem would have to induce cycling if approached from either direction in order to result in inability to find a solution. As the dimensionality of the space increases, the likelihood of *actual, practical* robot path planning problems possessing

deep concave cavities of safe c-space in both directions (start toward goal and vice versa) would intuitively seem to decrease. Such a c-space shape would probably not occur for practical problems.

The cyclic failure mode is not sufficient to rule out the existence of a solution since this mode can occur for a problem in which the search directions on the first search following reinitialization happens to miss all available safe space in the search hyperplane.

4.4 Computing Search Directions

This section discusses methods for computing search directions as required for Section 4.3 Step 5.

Recall from above that the c-space traversal heuristic involves searching the space orthogonal to and bisecting the unsafe region encountered in an attempted traversal. For an n-dimensional space $\mathbf{\Theta} = (\theta_0, \ldots, \theta_n)$, the n-1 dimensional hyperplane to be searched shall be orthogonal to direction vector $\mathbf{D} = (d_0, \ldots, d_n)$ and shall include point $\mathbf{\Theta}_C = (\theta_{C_0}, \ldots, \theta_{C_n})$, where $\mathbf{\Theta}_C$ is the center point of the unsafe segment. Thus, points to be considered in the search shall satisfy:

$$d_0(\theta_0 - \theta_{C_0}) + d_1(\theta_1 - \theta_{C_1}) + \ldots + d_n(\theta_n - \theta_{C_n}) = \sum_{i=1}^{n} d_i(\theta_i - \theta_{C_i}) = 0 \quad (4.1)$$

From Equation 4.1, it can be seen that the search directions $\mathbf{S} = (\theta - \theta_C)$ must be orthogonal to $\mathbf{D}$:

$$d_0 s_0 + d_1 s_1 + \ldots + d_n s_n = \sum_{i=1}^{n} d_i s_i = \mathbf{S} \cdot \mathbf{D} = 0 \quad (4.2)$$

Four procedures for determining search directions which satisfy Equation 4.2 were considered:

1. Searching a uniform grid

2. Radiating out along orthogonal basis vectors and their negatives.

3. Radiating out along a set of vectors made up of combinations of the n-1 free variables in Equation 4.2.

4. Radiating out along uniformly distributed vectors made up of combinations of orthogonal basis vectors.

These four procedures are discussed below. Selecting amongst the procedures for implementation is then addressed in Section 4.4.1.

Procedure 1 *Searching a uniform grid.*

Searching a uniform grid would involve discretizing uniformly in the $n-1$ dimensional search space defined by Equation 4.2. Such an approach would clearly produce a very effective search from the standpoint that it would ensure finding a safe point if one exists (within discretization limitations). However, this approach can be quickly dismissed due to its computational complexity. For example, an n dof problem discretized 100 points per axis (approximately every three degrees for a typical revolute joint) would produce a grid containing $100^{(n-1)}$ points. For a nine dof problem, this would result in 10^{16} points. Even if one million points could be mapped every second (far from achievable today) it would take more than 300 years to exhaustively perform one search of such a uniform grid!

Procedure 2 *Radiating out along orthogonal basis vectors and their negatives.*

A set of $n-1$ n-dimensional linearly independent and orthogonal unit vectors satisfying Equation 4.2 can be computed. Such a set of vectors would constitute a basis for the search space, i.e., each possible search direction could be represented as a linear combination of the basis vectors. A set of orthogonal basis vectors will be uniformly distributed in the space. Referring to the i^{th} basis vector as $\mathbf{B}_i = (b_{i_1}, \ldots, b_{i_n})$,

the basis vectors must satisfy:

$$\mathbf{D} \cdot \mathbf{B}_i = \sum_{k=1}^{n} d_k b_{i_k} = 0 \qquad i = 1, \ldots, n-1 \tag{4.3}$$

$$\mathbf{B}_i \cdot \mathbf{B}_j = \sum_{k=1}^{n} b_{i_k} b_{j_k} = 0 \qquad i, j = 1, \ldots, n-1 \text{ and } i \neq j \tag{4.4}$$

$$|| \mathbf{B}_i || = 1 \qquad i = 1, \ldots, n-1 \tag{4.5}$$

where $\mathbf{D}$ is the normal vector to the search space as per Equation 4.2, Equation 4.3 ensures that the basis vectors lie in the search hyperplane, and Equations 4.4 and 4.5 require all the basis vectors to be mutually orthogonal unit vectors.

There are, of course, an infinite number of orthogonal bases. Calculation of search directions requires only one. The following set of vectors could be calculated in the sequence shown and then normalized to yield one such orthogonal basis:

$$\begin{aligned}
\mathbf{B}_1 &= (1, h_1, 0, \ldots, 0) \\
\mathbf{B}_2 &= (b_{1_1}, p_2, h_2, 0 \ldots, 0) \\
\mathbf{B}_3 &= (b_{2_1}, b_{2_2}, p_3, h_3, 0 \ldots, 0) \\
\vdots \quad & \quad \vdots \qquad\qquad\qquad \vdots \\
\mathbf{B}_{n-1} &= (b_{n-2_1}, b_{n-2_2}, \ldots, b_{n-2_{n-2}}, p_{n-1}, h_{n-1})
\end{aligned} \tag{4.6}$$

where the p_i are chosen so that the $\mathbf{B}_i$ and $\mathbf{B}_{i-1}$ satisfy Equation 4.4 and then the h_i are chosen so that the $\mathbf{B}_i$ satisfy Equation 4.3.

Radiating out along the orthogonal basis vectors and their negatives would amount to considering search directions of the form $\pm\mathbf{B}_i$. This approach would yield $2(n-1)$ search directions for an n dof problem (16 for a nine dof problem). Thus, the number of search directions using this procedure would increase linearly with the number of dof, i.e., the complexity of searching with search directions based on this procedure would be $O(n)$. While this is an attractive feature it could be expected to perform poorly for cooperating robot path planning problems since such a reduced set of search vectors might miss the relatively little safe space available.

This expectation was verified when search directions based on Procedure 2 were found to be ineffective even for very simple robot path planning problems. The reason for discussion of this procedure is to illustrate that attempts were made to utilize as small a set of search directions as possible.

Procedure 3 *Radiating out along a set of vectors made up of combinations of the n-1 free variables in Equation 4.2.*

The third approach attempts to bridge the gap between the intractability of Procedure 1 and the oversimplification of Procedure 2. This procedure involves allowing the n-1 independent variables to take on all combinations of $\pm sd_i$ and solving for the dependent variable using Equation 4.2, where the sd_i may be chosen for each joint i as desired to vary the amount of motion being prescribed for joint i.

This approach will yield 2^{n-1} search directions for an n dof problem. While this procedure results in tractable numbers of search directions (256 for a nine dof problem), better performance may be possible using still more search directions.

A more extensive set of search directions could be computed by allowing the $n-1$ independent variables to take on all combinations of $\pm sd_i$ and 0 (except all zeros) and solving for the dependent variable using Equation 4.2, where the sd_i may again be chosen for each joint i. This will result in $3^{(n-1)}-1$ search directions for an n dof problem (6560 for a 9 dof problem).

This procedure for computing search directions is equivalent to considering all combinations of $\pm sd_i$ (and 0 for the more extensive set) times the following $n-1$ vectors:

$$\begin{aligned} \mathbf{V}_1 &= (sd_1, 0, \ldots, 0, \frac{d_1\ sd_1}{d_n}) \\ \mathbf{V}_2 &= (0, sd_2, 0, \ldots, 0, \frac{d_2\ sd_2}{d_n}) \\ &\vdots \qquad\quad \vdots \qquad\qquad \vdots \\ \mathbf{V}_{n-1} &= (0, \ldots, 0, sd_{n-1}, \frac{d_{n-1}\ sd_{n-1}}{d_n}) \end{aligned} \tag{4.7}$$

The potential disadvantage of this procedure is that the search directions will not, in general, be uniformly distributed in the search space. The degree to which coverage of the search space is non-uniform will depend upon the coefficients in Equation 4.2. Uniform distribution will occur only in the special case where $d_n \gg d_i$, for all $i \neq n$.

Procedure 4 *Radiating out along uniformly distributed vectors made up of combinations of orthogonal basis vectors.*

The final approach for computing search directions, radiating out along uniformly distributed vectors made up of combinations of orthogonal basis vectors, eliminates the non-uniformity which results using using Procedure 3. A uniformly distributed set of search directions could be computed by considering all combinations of ± 1 times the basis vectors. The basis vectors may be calculated per Equation 4.7. This approach will yield 2^{n-1} search directions for an n dof problem. Note that these search directions each involve a component along *all* of the orthogonal basis vectors.

An even more extensive set of search directions could be computed by considering all combinations ± 1 and 0 (except all zeros) times the basis vectors. This will yield $3^{(n-1)} - 1$ search directions for an n dof problem (6560 for a nine dof problem).

4.4.1 Selecting a Procedure

As mentioned above, Procedures 1 and 2 were eliminated from further consideration due to their computational complexity and apparent inadequacy, respectively.

Procedures 3 and 4 are similar in that they result in tractable numbers of search directions and in that the search density will automatically decrease with increasing distance from the center of the trouble region. Since it is impractical to have a uniform grid, it would seem desirable to decrease search resolution with distance from the center of the unsafe region since it is generally more desirable to find a point closer to the center of that region in order to attempt an efficient circumvention strategy. In other words, given a choice between failing to find a safe point near the center of the unsafe region and failing to find a safe point far from the center of the trouble region, one would choose the latter.

The differences between Procedures 3 and 4 are:

- Procedure 3 produces a non-uniformly distributed set of search directions whereas Procedure 4 guarantees uniform distribution.

- Procedure 3 allows for easy computation of search directions which favor certain joints whereas it is difficult to achieve such joint favoring using Procedure 4 since the basis vectors will, in general, have components in all joint directions.

The following example illustrates the uniform versus non-uniform distribution effect. Consider a three dimensional problem (so the search space will be planar) and let $\mathbf{D} = (2, 2, 1)$. The search directions that would be produced in the search plane using Procedures 3 and 4 are shown in Figure 4.7, where $sd_1 = sd_2$ for Procedure 3. Figure 4.7 shows that Procedure 4 consistently produces uniformly distributed search directions while Procedure 3 does not.

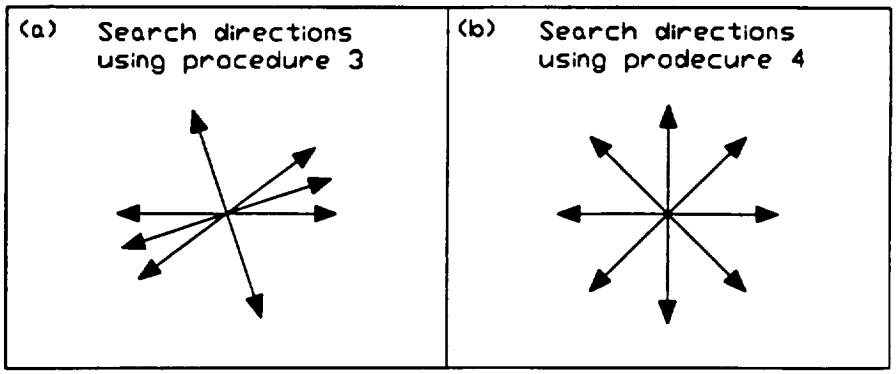


Figure 4.7: Procedure 3 vs Procedure 4

Experimentation was done with Procedures 3 and 4 for the cases implemented in Chapter 6. In all four scenarios considered (single 6 dof, single 9 dof, cooperating 6 dof, and cooperating 9 dof) both procedures were successful in solving a variety of problems. For more difficult problems, however, Procedure 4 produced noticeably better results, often with fewer search directions. This was true in spite of the ability to favor certain joints using Procedure 3.

As discussed in Chapter 6, search directions computed from the more extensive set based on combinations of $\pm sd_i$ and 0 times the basis vectors proved to be practical and effective for six dof problems. For 12 dof problems (such as cooperating nine dof robots), however, this procedure would produce 177146 search directions and thus could potentially result in very long execution times. In the 12 dof case, search directions computed from the smaller set based on combinations of $\pm sd_i$ times the basis vectors (which yields 2048 for a 12 dof problem) proved to be a good compromise between practicality and effectiveness.

4.5 Prioritizing Search Directions

This section discusses methods for prioritizing search directions as required for Step 6 of Section 4.3.

Recall from above that the search directions are to be prioritized based on their dot product with the previously successful search directions. Recall also that the searches are conducted by looking at successively prioritized groups of search directions. Two methods were considered for achieving this prioritization:

- Sorting the search directions
- Grouping the search directions into bins

The first method would simply involve sorting the entire list of search directions based on their dot products with the previously successful search direction. Following the sorting, the search directions will be divided into groups of search directions having similar priority. This type of sorting was found to be computationally burdensome, unacceptably so for cases with several thousand search directions.

Grouping the search directions into bins involves much less computation than sorting the entire list and would seem to provide similar performance to sorting since the treatment of each search direction within a particular group differs only in the order in which they are considered (and not in the relative depths considered in each direction). Sorting into bins can be easily accomplished. If the dot product of the i^{th} search direction, $\mathbf{S}_i$, with the previously successful (or reference) search direction, $\mathbf{S}_{ref}$, is dp_i, and the maximum and minimum dot products are $dpmax$ and dp_{min}, respectively, then a set of search directions can be grouped into g equal breadth groups (bins) by the following rule:

$$\mathbf{S}_i \in bin(j) \;\; if \;\; \frac{j-1}{g} \leq \frac{dp_i - dp_{min}}{dpmax - dp_{min}} \leq \frac{j}{g} \tag{4.8}$$

It is this technique of bin sorting which is implemented in Chapter 6.

Another variation on the prioritization method is to consider the *past history* of successful search directions rather than simply considering the previous successful search direction. This can be accomplished by computing dot products with the

following reference search direction computed following a successful search:

$$\mathbf{S}_{ref} = \lambda \, \mathbf{S}_{ref'} + (1 - \lambda) \, \mathbf{S}_s \tag{4.9}$$

where $\mathbf{S}_{ref'}$ is the previous reference search direction, $\mathbf{S}_s$ is the most recent successful search direction, and $\lambda \in [0, 1)$ represents a *forgetting factor* which may be used to vary the emphasis on the past history. With $\lambda = 0$, the method results in prioritizing exclusively based on the last successful search direction. The case $\lambda = 1$ is disallowed since $\mathbf{S}_{ref}$ would be invariant in that case.

Since in cooperating robot cases the role between leading robot and tracking robot may change (as discussed in the next Chapter), an effective reference search direction must be calculated for the tracking robot after each successful search. This effective reference search direction is the search direction which would have yielded the safe point found had the search been based on the tracking robot rather than the leading robot.

4.6 Comparison of the Heuristic to the Literature

This heuristic is somewhat similar to many of the c-space graph search techniques in that it is based around *selective* rather than *exhaustive* mapping of c-space. Aside from that broad similarity, this heuristic is fundamentally and significantly different from any of the approaches discussed in Chapter 2, with the most significant difference stemming from the process used to guide the selective mapping process. Nonetheless, it bares some some resemblance to Dupont's selective mapping [5], Glavina's goal directed sliding [46], and Warren's vector based approach [58] (see Section 2.1.1). Specific similarities and differences are discussed below.

The heuristic is similar to Dupont's approach in that both attempt to initially follow a c-space vector from start to goal and employ heuristics to attempt to minimize the amount of mapping required to circumvent unsafe portions of the path. The key difference is the type of heuristic used to attempt to traverse the trouble

regions. Interested only in single (redundant) robots, Dupont successfully used task space heuristics to build paths from each end of the trouble region until a feasible solution was found. The approach being presented here utilizes the c-space traversal heuristic described above to guide the selective mapping process.

The resemblance to Glavina's approach is that both perform a search in the n-1 dimensional hyperplane containing a point which was unsafe in the straight traversal between two points. Glavina's approach, however, performs those searches at the beginning of the trouble region and is therefore subject to blindly following strategies which look locally promising at the beginning of the trouble region but which may not lead to traversal around that region. Glavina's approach does, however, have the advantage over the heuristic being presented in this thesis in that it does not introduce intermediate points which may be in unreachable regions of free space. It is felt that that advantage does not outweigh the inherent inability of a completely local strategy to adopt a promising global course. It is expected that Glavina's approach would become excessively computationally intensive for problems with six or more dof even if the safe c-space possessed only relatively shallow concavities.

The resemblance to Warren's approach is that both are graph search type and "divide-and-conquer" in nature in that they attempt to identify an intermediate *via* point by searching outward from the center of the trouble region. The resemblance ends, however, when comparing the means used to identify a safe intermediate point. As discussed in Section 2.1.1, Warren's approach projects a vector from the centroid of the obstacle through the center of the unsafe region whereas the heuristic presented in this thesis utilizes structured searches of the hyperplane bisecting the trouble region. Warren's approach, while relatively new and still under development, has some potential difficulties:

- Obstacle centroids must be known *in the space being considered* (typically c-space). This is computationally intractable for more than a few dof.

- If the centroid lies on or near the unsafe vector the resulting intermediate point will lie at or near a previously found point thereby providing no new information.

- The case for which no safe point is found along the vector is not considered. As the dimensionality of the problem increases, the likelihood of finding a safe point along one particular vector would decrease rapidly.

- The case of obtaining an intermediate point in an unreachable region of space is not addressed.

These potential difficulties are all either addressed or eliminated by the approach being presented in this thesis.

Some of the potential fields approaches also adopt a "divide-and-conquer" style solution to attempt to circumvent local minima difficulties. Some techniques used in conjunction with potential fields approaches to locate intermediate trial points (via points) include task space heuristics, uniform grids, randomized motions, and use of potential functions (see Section 2.1.2). The heuristic presented herein does not resemble any of these approaches beyond the fact that each involve a divide-and-conquer style strategy.

CHAPTER 5
Utilizing the Heuristic for Robot Path Planning

This chapter explains how the divide-and-conquer c-space traversal heuristic presented in the preceding chapter may be utilized to solve single and cooperating robot path planning problems. This chapter is organized into three main sections:

- Single Robot Path Planning
- Cooperating Robot Path Planning
- String Tightening
- Handling Constrained Motions

Sections 5.1 and 5.2 discuss the utilization of the heuristic for single and cooperating robot path planning problems, respectively. A "string tightening" method to improve the efficiency of a path found by the planner is presented in Section 5.3. The implementation of the path planning strategy for particular single and cooperating robots is deferred until the following chapter.

5.1 Single Robot Path Planning

The single n dof robot path planning problem as defined in Section 3.4 can be addressed by direct application of the heuristic presented in Chapter 4 where the n dimensional space to be traversed is simply the configuration space of the n dof robot. C-space points are mapped only as needed by updating the geometric models of the robot links and the payload and performing interference detections as required to determine whether or not the specified joint variables correspond to a collision free configuration.

In all cases, the parameter c which determines step size (see Step 2 of Section 4.3) should be established for each task such that the largest possible step is many times smaller than the step size necessary for thinnest part of the robot/payload to step through the thinnest obstacle in one step.

Some potential issues which arise are:

- Handling robots with mixed joint types
- Joint limit problems
- Choosing λ
- Choosing number of bins
- Multiple robot configurations
- Singularity concerns

These issues are addressed below.

5.1.1 Handling Robots with Mixed Joint Types

Mixed unit concerns for robots with mixed joint types (some prismatic and some revolute) may be eliminated by linearly mapping each joint's actual range onto the interval $[0, 1]$, i.e.:

$$\theta = \frac{q_a - q_{min}}{q_{max} - q_{min}} \tag{5.1}$$

where q_a, q_{min}, and q_{max} represent the actual joint value, the lower joint limit, and the upper joint limit, respectively, all in identical units for each joint. Robots with revolute joints having no joint limits may be treated by replacing the denominator on the right hand side of Equation 5.1 with 360 degrees (2π radians). No multiple rotations are permitted.

5.1.2 Joint Limit Problems

The joint limit problem is handled inherently since any point which would violate a joint limit is simply mapped as unsafe. In addition, the prioritization of search directions allows a reversal to take place when a potential joint limit is encountered. The prioritization strategy will then favor the direction away from the joint limit even after the immediate danger of hitting a joint limit is avoided. This reversal tendency is more global than the technique often employed with potential fields methods whereby a joint is repelled if it is in proximity to a joint limit.

5.1.3 Choosing λ

Recall from Equation 4.9 that prioritization of search directions utilizes a parameter denoted as λ. Experimentation with the test cases in Chapter 6 indicates that small λ (near or equal to 0) provides the most robust path planner from the standpoint of finding a path for difficult problems, particularly for single robot problems. Path efficiency, however, appears to decrease with decreasing λ. In addition, small λ does not perform particularly well for cooperating robot cases. This is likely partially due to the swapping of roles between the leading and tracking robot. The values used for λ for the cases implemented will be presented in Chapter 6.

5.1.4 Choosing Number of Bins

Recall from Equation 4.8 that prioritized searches consider search directions grouped into bins. For the wide variety of problems considered, either 5 or 10 bins proved successful. In most cases, any number of bins in the 5 to 10 range would yield a solution although the path efficiency may decrease with an increase in the number of bins. Fewer than 5 bins did not provide robust performance and more than about 20 bins led to very inefficient paths (if a solution could even be found).

5.1.5 Multiple Robot Configurations

Recall from the problem definition in Chapter 3 that the start and goal joint angles are given. In addition, note that the path planner operates exclusively in c-space. As a result, multiple robot configurations which achieve identical end effector position/orientation need not be explicitly considered by the path planner.

5.1.6 Singularity Concerns

There are no singularity concerns using this approach for single robot path planning since singularities are a task space phenomena whereas the path planning approach is strictly configuration space based.

5.2 Cooperating Robot Path Planning

The two cooperating arm path planning problem is essentially equivalent to the single arm problem with the addition of the closure constraint, Equation 3.3. The closure constraint requires that, in order for a point in the configuration space of the one robot to be considered safe, it must correspond to a reachable and collision free configuration of the second robot. Thus, the basic concept for attacking the cooperating robot path planning problem is to apply the c-space traversal heuristic to one of the robots, referred to herein as the lead robot, with the other robot, referred to herein as the tracking robot, acting as a constraint. For example, the straight line path in c-space is determined for the lead robot and an attempt is made to traverse from the start position towards the goal position. If this attempted traversal is not entirely safe a search is conducted in the c-space of the lead robot with due consideration to the tracking robot. When the lead robot reaches the global goal position the entire path planning problem will have been solved. Mapping a particular point in the c-space of the lead robot involves verifying that the closure constraint can be met, updating geometric models of the robot links and payload,

and performing the required interference detection calculations.

The above rather simplistic conceptual explanation of applying the c-space traversal algorithm to two cooperating robots neglects the following potential issues:

- Handling robots with mixed joint types
- Joint limit problems
- Choosing a lead robot
- Handling cooperating redundant robots
- Multiple robot configurations
- Singularity concerns

The first two of these issues are identical for the cooperating robot case as for the single robot case discussed in Section 5.1. The remainder of these issues are discussed below.

5.2.1 Choosing a Lead Robot

The simplest way to choose a lead robot would be to always choose the same robot. This simple approach can be dismissed for the following reasons:

- A small change in the configuration of the lead robot might correspond to a much larger change in the configuration of the tracking robot thereby making it difficult to discretize the path to ensure that it is collision free. In an extreme case, it is possible that the lead robot may have the same start and goal positions for radically different start and goal configurations of the tracking robot (such as an arm configuration change).
- It would not allow the tracking robot to easily change configuration since this would typically involve passing the tracking robot through a singularity. It

is highly unlikely that the traversal heuristic would happen to prescribe lead robot positions which would allow the tracking robot to change configuration.

These difficulties may be eliminated by choosing the lead robot for each call to the heuristic based on relative distances (in c-space) between start and goal positions of each of the robots. This approach can be represented as follows:

$$\text{if } ||\Theta 1_s - \Theta 1_g|| < r\,||\Theta 2_s - \Theta 2_g|| \quad \text{then robot 1 leads} \quad \text{otherwise robot 2 leads} \tag{5.2}$$

where $r \geq 1$ represents a relative weighting between the two robots. Setting $r = 1$ would result in simply choosing the lead robot as the one with the greatest distance to travel. Equation 5.2 is evaluated to select the lead robot for each segment of the path where the s and g subscripts represent not the global start and goal positions but rather the start and goal positions for the particular segment of the path being addressed.

Experimentation with the cases in Chapter 6 revealed that oscillation tends to occur using this method for $r = 1$. These oscillations resembled a tug-of-war between the two robots.

Better path planner performance was achieved by choosing the lead robot based on relative c-space distances with consideration to past history. This approach favors the robot which led the previous segment unless the other robot has some multiple r further to go, i.e.:

if robot i had led robot j and

$$\text{if } ||\Theta j_s - \Theta j_g|| < r\,||\Theta i_s - \Theta i_g|| \quad \text{then robot i leads} \quad \text{otherwise robot j leads} \tag{5.3}$$

where $r > 1$ represents a relative weighting by which the distance for the formerly tracking robot must exceed the distance for the formerly leading robot before the

roles are reversed. Essentially, this method incorporates some hysteresis into the determination of the leading robot.

This approach was used to select the lead robot for the cases implemented in Chapter 6.

5.2.2 Handling Cooperating Redundant Robots

The implementation of the c-space traversal heuristic for cooperating robots as described in Section 5.2 requires that the closure constraint be checked for the tracking robot. Since each point in the c-space of the lead robot defines a position of the end effector of the tracking robot, inverse kinematics must be applied to determine if and how the tracking robot can reach a prescribed position/orientation. For cooperating non-redundant robots, the reachability of the second robot can be easily determined using inverse kinematics which are one-to-one. Checking the closure constraint for cooperating redundant robots, however, can be potentially difficult since the inverse kinematics are not one-to-one. Two possible methods of addressing the cooperating redundant robot path planning problem are:

- Applying the heuristic directly to one of the robots
- Applying the heuristic to a composite c-space with dimensionality equal to total number of degrees of freedom for the cooperating system

These two approaches are discussed below.

5.2.2.1 Applying the Heuristic Directly to One of tne Robots

Application of the procedure directly to one of the robots would require some means for performing inverse kinematics on the redundant tracking robot. This inverse kinematics problem could be handled either by iterative testing of a number of prescribed positions for all but six of the joints or by utilization of a potential fields

based inverse kinematics solution. Iterative testing would likely prove very computationally expensive. A potential fields based inverse kinematic solution would be computationally tractable. Such an approach, however, has an intuitive disadvantage, namely that it does not treat all the free variables of the path planning problem in the same fashion. In implementation terms, this means that the treatment of the tracking robot would not contribute significantly to the overall strategy for solving the global cooperating robot path planning problem.

In attempt to further clarify this point, consider an example for which a potential fields inverse kinematics solution is used for the tracking robot. The inverse kinematics applied to each point prescribed by the lead robot must consider the position of the tracking robot at the previous point. This is necessary to avoid a discontinuous path for the tracking robot. The difficulty arises when the lead robot prescribes a point in the progression for which the inverse kinematics fail for the tracking robot. That failure of the inverse kinematics is contingent upon the path of the tracking robot up to the point before failure. Since no global path planning strategy was incorporated into the inverse kinematics of the tracking robot, it seems likely that better results might be obtained using a different strategy for selecting the configuration of the tracking robot.

5.2.2.2 Applying the Heuristic to a Composite C-Space

This technique for considering cooperating redundant robots was developed to enable the heuristic to be applied to a space with dimensionality equal to the effective number of dof for a cooperating system of robots. To illustrate this method, consider an n_1 dof robot (Robot 1) working cooperatively with an n_2 dof robot (Robot 2), $n_i > 6$. The mobility of the cooperating system is $m = n_1 + n_2 - 6$ per Equation 1.1. The two robots can be conceptually replaced with an m dof lead robot and a six dof tracking robot by treating $n_2 - 6$ links of Robot 2 as if they belong to Robot 1.

In this manner, the c-space traversal heuristic can be applied to the mD c-space of the composite lead robot while one-to-one inverse kinematics can be applied to determine if the tracking robot can satisfy the closure constraint.

The main concern regarding this approach is that it results in increased dimensionality of the space which must be searched when implementing the c-space traversal heuristic. This increased dimensionality does, however, accurately reflect the problem complexity and is therefore considered reasonable. It also seems reasonable to expect that the traversal heuristic would handle the extra dof in a more logical fashion than considering them in the inverse kinematics of the tracking robot.

Application of this procedure to cooperating nine dof robots would amount to considering a twelve dof composite robot being tracked by a six dof robot. The heuristic would then be applied to the 12D c-space of the composite robot. Results presented in Chapter 6 illustrate that this technique is a practical and effective way to address the path planning problem for cooperating nine dof robots.

A similar approach could be applied to cooperating robots with less than six degrees of freedom. For example, consider two five dof manipulators. Since the inverse kinematics for the five dof robot would be overdetermined (i.e., not every position and orientation would have a solution), it would appear more effective to plan based on, for example, the first four joints of a lead robot. The lead robot's remaining joint and the five joints of the tracking robot would effectively result in a six degree of freedom robot with one-to-one inverse kinematics. In this case, such an approach would actually reduce the dimensionality of the search space (from five to four) as compared to direct application of the heuristic to one of the robots. Once again, the heuristic is applied to a space with dimensionality equal to the actual mobility of the cooperating system.

5.2.3 Multiple Robot Configurations

In general, a six dof robot will possess a finite number of distinct robot configurations which achieve identical end effector position/orientation (such as *elbow up* or *elbow down* for a Puma). This situation is represented mathematically in Equation 3.5. Multiple configurations are handled inherently for the lead robot just as in the single robot case. However, special consideration is required to address this issue for the tracking robot. The following set of rules address this issue:

1. Configurations must be defined such that, for the robot in any one configuration, an infinitesimal change in end effector position/orientation will always correspond to an infinitesimal change in the corresponding joint angles.

2. During progressions forward through safe space (Steps 3 and 9 of Section 4.3), the tracking robot shall maintain the same configuration as it had at the start of that segment of the path.

3. While mapping through unsafe space in search of a safe point (Step 4 of Section 4.3), only the configuration of the tracking robot at the goal position of the current segment of the path shall be considered.

4. While conducting searches (Step 7 of Section 4.3), all possible configurations of the tracking robot shall be considered.

These rules will enable full use of all available configurations while prohibiting discontinuous motions of the tracking robot for smooth motions of the leading robot.

5.2.4 Singularity Concerns

Robot arm degeneracy at singularities is handled inherently by the path planning method. For the lead robot, only singularity-free c-space is considered. For the tracking robot, any region prescribed by the lead robot which cannot be tracked

by the other robot is mapped out as an unsafe region. This combined with the ability to swap roles between the leading and tracking robots results in a planner which inherently handles singularity concerns for cooperating robots. This means of handling singularities does not attempt to physically avoid singular configurations but rather allows either robot to pass through singularities as necessary when attempting to solve the path planning problem.

5.3 String Tightening

The path planning procedure presented thus far has a principle objective of finding a feasible solution. As a result, the paths found will typically be sub-optimal in some sense and it should be possible to modify a feasible path found by the planner to produce a better one. This process of path modification may be referred to as string tightening. This section presents a brief history of approaches used for string tightening and then presents an approach which can be utilized for string tightening paths found for two cooperating robots.

5.3.1 History of Smoothing

Once a collision free path has been found by a robot path planner, it can be further optimized by numerical methods. A commonly used cost function aims to minimize the length of the path while incorporating safety clearances from obstacles. The resulting performance index to be minimized can be expressed as:

$$J = \int_{\Theta_s}^{\Theta_g} (1 + \frac{w}{D(\Theta)}) d\Theta \tag{5.4}$$

where $D(\Theta)$ is the minimum distance between the robot and obstacles, w is a weighting factor, and the integral is taken over all configurations connecting Θ_s and Θ_g. Polytope methods seem to be the current state of the art for computing robot to obstacle distances. Bryson and Ho [96] note that several numerical methods

may be used to find a path with minimum J using any feasible path as an initial guess. Simple gradient methods perform reasonably well for this purpose. The resulting path, however, is only optimal in the vicinity of the initial guess.

An alternate technique for path smoothing which also attempts to shorten a path while maintaining due safety clearances is Thorpe's [97] path relaxation technique. This process begins with a mobile robot path consisting of straight line connections between a sequence of nodes. The relaxation involves moving one node at a time in either direction perpendicular to the line connecting the preceding and following nodes in order to minimize the cost of traversing between the three nodes. The cost function is similar to Equation 5.4 since it includes length of path segment with a penalty for proximity to obstacles or unmapped (unknown) regions. Since moving a node may affect its neighbors, the process is repeated until no nodes move more than some small tolerance.

Another technique which can be used to smooth paths, avoid collisions, and move paths away from objects is based on potential fields. Krogh [74] presents one such approach. Krogh uses sensory measurements of obstacles as feedback during execution of paths planned with another algorithm. This feedback can help to smooth jagged paths and to steer the path away from obstacles.

5.3.2 String Tightening Algorithm

This section presents a method for improving upon a path produced by the cooperating robot path planner. Recall from Chapter 3 that the path planner output consists of a sequence of closely spaced knot points for both robots along a feasible and collision free path.

5.3.2.1 Measure of Goodness

A variety of possible criterion may be used to evaluate the quality of a path. For string tightening purposes, the goodness of a path may be measured by the sum of the lengths of the joint space trajectories for the two cooperating robots. Since the path planner produces discretized paths for both robots, the objective during string tightening is to reduce the following cost function:

$$L_1^N = \sum_{r=1}^{2} \sum_{i=1}^{N} \sqrt{\sum_{j=1}^{n_r} (\theta r_j(i+1) - \theta r_j(i))^2} \tag{5.5}$$

where:

L_1^N = the sum of the joint space trajectory lengths

N = number of knot points in path

r = robot identifier

n_r = number of dof for robot r

$\theta r_j(i)$ = i^{th} knot point for robot r joint j

If the original path is considered to be a string passing through the knot points in the joint space of each of the robots, then the objective for improving upon the path is to shorten the sum of the string lengths while maintaining the same endpoints. Hence the name string tightening as suggested by Dupont [5].

The tightening algorithm which was implemented involves examining each sequence of three adjacent knot points and performing whichever of the three options below produces the most desirable effect on L_1^N:

1. Make no changes to the knot points.

2. Modify the second knot point for robot 1 so that the three knot points are straight in the joint space of robot 1 (if not already so).

3. Modify the second knot point for robot 2 so that the three knot points are straight in the joint space of robot 2 (if not already so).

The feasibility of options 2 and 3 must be determined with consideration to closure and collisions. The procedure described in Section 5.2 can again be used to simplify the question of closure for cooperating redundant robots. The incremental effect which each of the above options will have on L_1^N can be assessed using Equation 5.5 over the appropriate three knot point segment.

These local adjustments are continued until no significant improvement can be obtained from further adjustments.

A conceptual illustration of the string tightening algorithm for cooperating robots is shown in Figure 5.1. An initial three knot point segment for the two robots is shown in Figure 5.1a. These three knot points are a portion of a much longer many knot point path. Figure 5.1b and c show the effect of options 2 and 3, above. In this example, option 2 (moving the second knot point of robot 1 in line with its neighbors) produces the most significant reduction in path length. Thus, this iteration would move each robot's second knot point to their positions in Figure 5.1b.

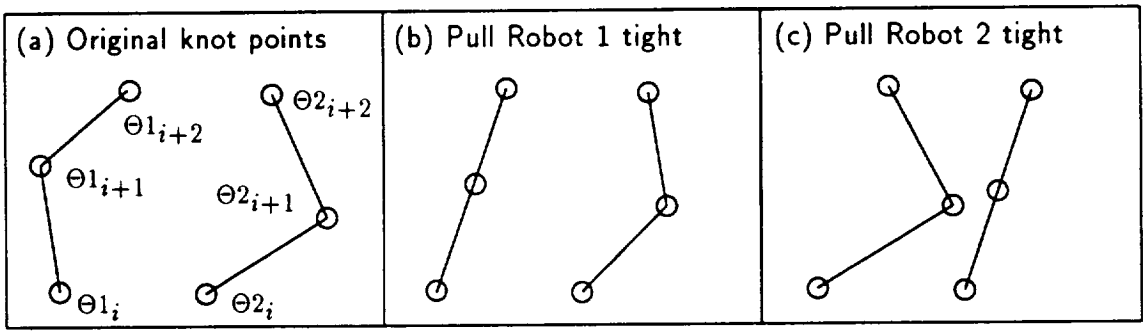


Figure 5.1: Local Effect During String Tightening

For single robot problems, Equation 5.5 need only be evaluated for one robot and the options are reduced to two:

1. Make no changes to the knot points.

2. Modify the second knot point so that the three knot points are straight in the robot's joint space (if not already so).

5.3.2.2 Limitations of the String Tightening Algorithm

Because this string tightening method involves a discretized approximation to continuous deformation, the tightened path may still be far from optimal. For example, consider Figure 5.2. A safe path may be found as shown in Figure 5.2a. A shorter path found by continuous deformation of the original path is shown in Figure 5.2b. However, this path is suboptimal as shown by Figure 5.2c.

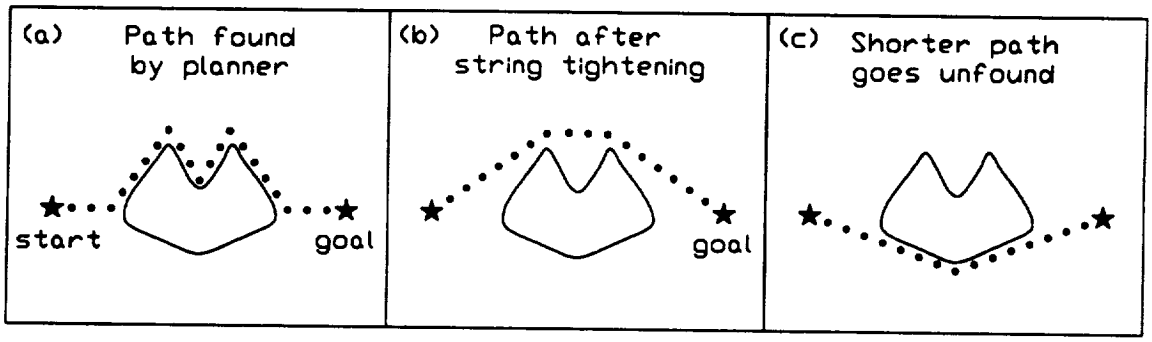


Figure 5.2: String Tightening May Not Produce Optimal Path

One disadvantage of the approach is that the shortened paths tend to provide very little obstacle clearance. This property is generally more acceptable for manipulators than for mobile robots because the manipulator environment is generally accurately known and the manipulator control is typically precise. Possible means for addressing this limitation are discussed in Section 8.2.1.

This string tightening algorithm is also unable to find any paths which would require temporary lengthening of the path in order to ultimately achieve a better path.

5.3.3 Comparison to Other Path Smoothing Approaches

This approach is very similar to Thorpe's approach discussed in Section 5.3.1 where the differences are as follows:

- Cooperating robots are considered.
- The cost function is c-space distance only, whereas Thorpe includes distance from obstacles in the cost function.
- The sequences of points are closely spaced knot points, whereas Thorpe's node points may be far apart.

5.4 Handling Constrained Motions

Earlier, it was assumed that the end effector motion between the start and goal positions may be arbitrary. Though this is a valid assumption for the typical robot path planning problem in free space, there are cases where contact between the payload and an obstacle may lead to constrained rather than arbitrary end effector motion. For example, the payload may come into planar contact with a table surface. As such, the end effector motion is confined to 3 dof (two translations and a rotation) as opposed to 6 dof. Although such cases are not considered in this thesis, the heuristic could be utilized to solve such problems by applying the heuristic in the task space defined by the reduced degrees of freedom rather than in the joint space of the robot. The robot must be away from singularities in order for such an approach to be effective.

CHAPTER 6
Implementation and Results

This chapter presents the implementation details and results of applying the path planning method described in the preceding chapters to the following single and cooperating robot scenarios:

- The CIRSSE Testbed (single 6 dof, single 9 dof, cooperating 6 dof, and cooperating 9 dof cases)
- The Automated Structure Assembly Lab at NASA Langley (6 dof case)
- Cooperating Pumas Assemble a Truss Structure

The specifics of each of these implementations and sample results are presented in sections which follow. First, some points common to all of these implementations are presented in the next section.

6.1 Characteristics Common to All Implementations

All of the implementations that will be discussed in this chapter have the following common characteristics:

- Heuristic is applied generically
- Geometric modeling is done with polytopes.
- A hierarchical interference detection scheme is used.
- Paths may be visually simulated using CimStation.
- The programs are written in C.

These characteristics are discussed below.

6.1.1 Heuristic is Applied Generically

All of the cases invoke the c-space traversal heuristic in its completely general form. In other words, in no case are task or hardware specific assumptions or modifications utilized. Search direction computation is always done strictly mathematically. The ability to directly apply the heuristic generically to a variety of problems suggests that the planning methodology presented herein could be quickly and effectively applied to hardware or tasks not addressed herein.

6.1.2 Geometric Modeling with Polytopes

The geometric modeling scheme implemented to enable interference detection utilizes polytope models of the robot links, payload, and obstacles in the workspace. Details of the modeling may be found in [6]. A polytope is a set of points whose convex hull (the smallest volume which encloses all points) describes the object being represented. The polytope representation incorporates a radius which can be used to achieve a safety margin. A few simple 2D polytopes are shown in Figure 6.1. In 3D, a two vertice polytope would correspond to a cylinder with hemispherical end caps, where the radius of the cylinder and of the end caps is specified by the polytope radius. A 3D block can be made using eight vertices and a radius of zero.

The polytope representation scheme was chosen because it permits accurate modeling of the robots and typical obstacles in the workcell while enabling relatively fast interference checking. Although each polytope represents a convex object, concave objects may be easily modeled as several distinct convex polytopes.

6.1.3 Hierarchical Interference Detection

Collision checking is currently being done in a two level hierarchy. First, spherical approximations for each pair of potentially colliding objects are examined. If the spherical approximations do not intersect then there is no possibility of collision

PAGE _____ INTENTIONALLY BLANK

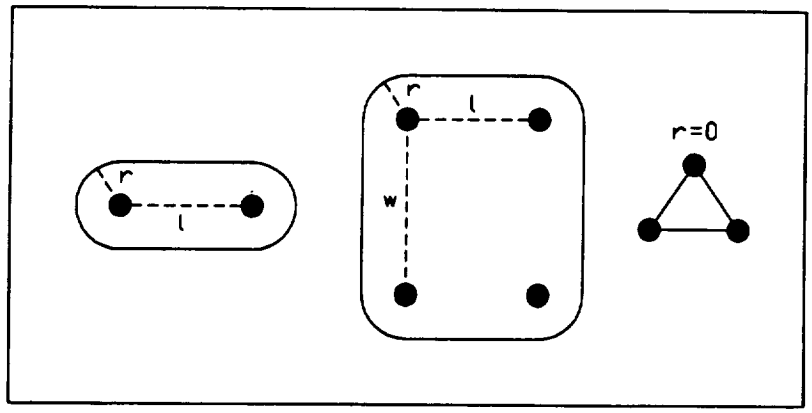


Figure 6.1: Some 2D Polytopes

between the pair of objects under consideration. If the spherical approximations do intersect then a polytope distance calculation routine is invoked to determine whether or not the two objects intersect (collide). The polytope routines being used were provided by Hamlin and Kelley [98, 99]. The reason for the spherical approximation level of the hierarchy is to reduce the number of computationally expensive calls to the polytope distance calculation routine.

Mapping a point in c-space thus reduces to the following steps:

1. Verifying the closure constraint and determining the configuration of the tracking robot (not necessary in single robot cases).

2. Updating the coordinates of the sphere centers and polytope vertices based on the joint angles of the point being mapped.

3. Performing interference detection per the hierarchy discussed above.

The interference detection routine for the path planner simply needs to determine a *yes* or a *no* regarding collision. This enables use of faster routines than would be required if the path planner needed to know distances and directions between pairs of objects.

6.1.4 Animation of Paths

Paths found by the path planner may be visually animated using any suitable robot simulation package. We use CimStation, a commercially available package, for path animation purposes. The interface between the path planning programs and CimStation is a file storing the sequence of knot points determined by the path planner. CimStation then replays the sequence to animate the path found by the planner. The CimStation workcell model must, of course, be consistent with the world model given to the path planner. The CimStation model of the CIRSSE testbed used for this work was developed by Hron [100]. The CimStation model of NASA Langley's ASAL lab was provided to us by NASA Langley. The model used for the cooperating Pumas assembling a truss is an edited version of the model of the CIRSSE testbed.

For CIRSSE testbed cases, path planning program output may also be run on the actual hardware by first applying a trajectory generation routine to the planner output and then running the resulting trajectory in the typical fashion. For cooperating robot cases, path execution in this manner requires use of active compliance on one of the two end effectors at any given time to maintain acceptable internal forces on the payload. Further work to be done in the area of integrating the path planner into the CIRSSE testbed is discussed in Section 8.2.2.

CimStation was also found to be very useful in defining the start and goal joint angles for path planning problems, particularly in the cooperating robot case. Due to the tremendous loss of workspace due to the closure constraint, it is easy to inadvertently define start and goal positions of the robot which are feasible but which probably have no path which can connect them. CimStation may be used to view different robot configurations and to quickly determine the feasibility of a robot reaching a particular pose. The various robot configurations may be tried as input to the path planner until a solution is found.

6.1.5 Description of Programs

All of the path planning programs were implemented in the C programming language. Portions of the program utilize code developed by Schima [6]. The path planning programs are similar for all cases considered. A sample flowchart is shown in Figure 6.2.

Program inputs and outputs are per the problem statements in Chapter 3. Additional output is included to document and evaluate the performance of the path planner. This output includes the following:

N_p = Number of knot points in path.

N_s = Number of searches required.

$\Delta L/L$ = Percent reduction in path length due to string tightening.

L_f = Final path length after string tightening (Eqn. 5.5).
Note that this is dimensionless since joints are
scaled using Equation 5.1

N_{sph} = Calls to spherical interference check function.

N_{poly} = Calls to polytope interference detection function.

t_{path} = Time to find safe path.

t_{tight} = Time to string tighten.

t_{cc} = Time spent collision checking (both phases).

t_{poly} = Time spent in polytope phase of collision detection.

t_{tot} = Total time.

The condition used to terminate string tightening is that a knot point will be moved only if doing so will reduce the distance over the three knot point segment centered at that point by at least 0.5 percent.

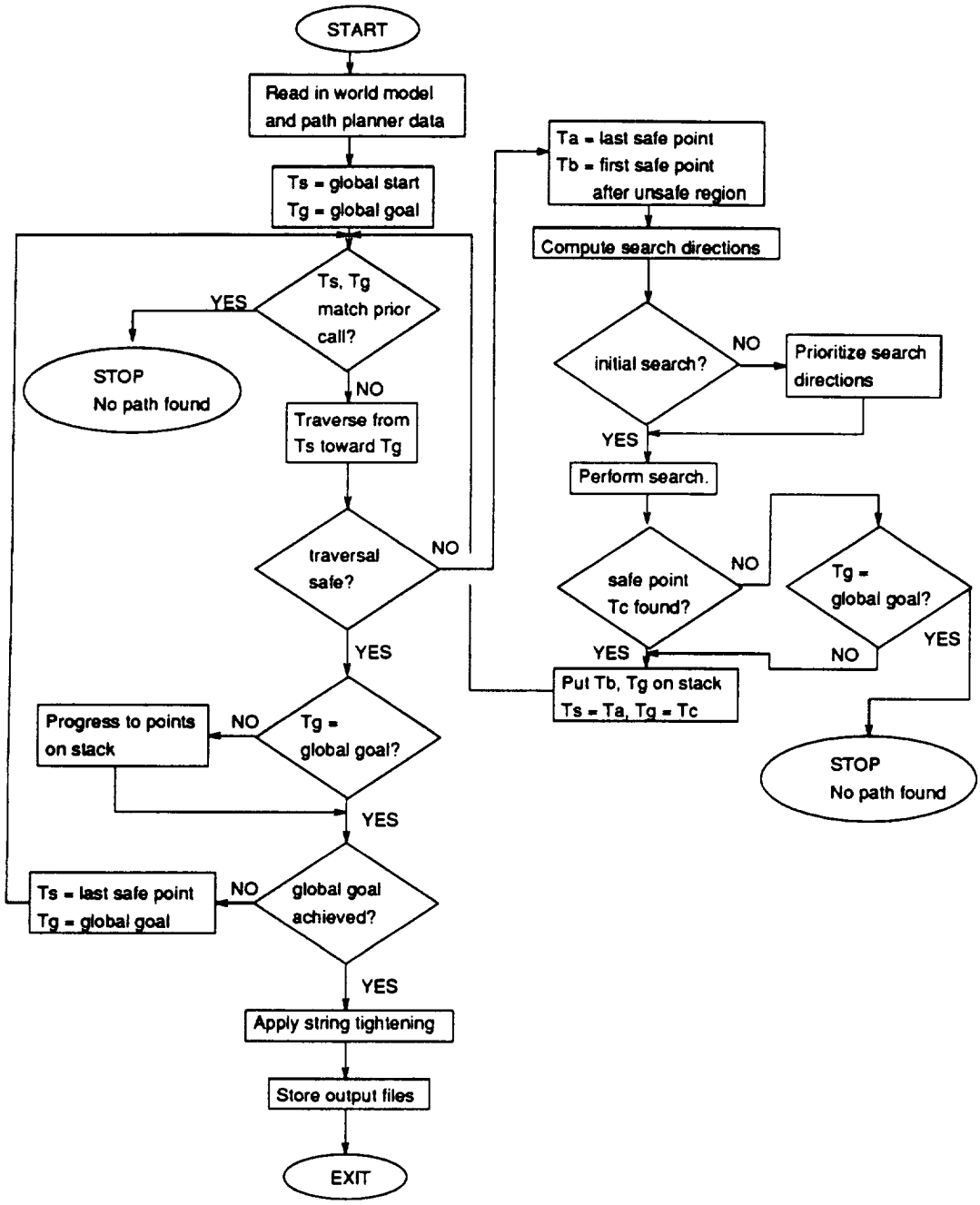


Figure 6.2: Flowchart of Path Planning Program

6.2 CIRSSE Testbed

The path planning method described herein has been implemented for the robotic testbed system of the Center for Intelligent Robotic Systems for Space Exploration (CIRSSE) at Rensselaer Polytechnic Institute (RPI). The CIRSSE testbed, shown earlier in Figure 1.2, consists of two 6R Puma 560's, each of which rides on a separate Aronson 1P-2S platform. The kinematic parameters including joint limits may be found in [101] and in Appendix A.

The methods described in this thesis have been implemented for four different CIRSSE testbed scenarios:

- Single Puma
- Single 9 dof robot (platform plus Puma)
- Cooperating Pumas
- Cooperating 9 dof robots

Numerous path planning problems were contrived for these different scenarios in attempt to illustrate the effectiveness of the path planner for various potentially difficult path planning problems. Implementation details and sample results for each of the scenarios are presented below. Applications of the path planner to more practical path planning problems are discussed later in Sections 6.3 and 6.4. Except as noted for the case specifically illustrating the effect of string tightening, all paths illustrated herein are the final path obtained after string tightening. Start and goal joint angles and obstacle definitions for the included CIRSSE testbed examples (Examples 1 through 4) are provided in Appendix B.

6.2.1 Single Puma 560

The path planner was implemented for such a single Puma path planning problem. The specific parameters of the implementation are as follows:

$$
\begin{aligned}
c &= \frac{1}{200} \text{ step size (See Step 2 of Section 4.3)} \\
N_{SD} &= 242 \text{ search directions per Procedure 4} \\
g &= 10 \text{ bins (see Equation 4.8)} \\
\lambda &= 0.5 \text{ forgetting factor (See Equation 4.9)}
\end{aligned}
$$

6.2.1.1 Example 1

A sample path found by the single Puma path planner is shown in Figure 6.3. Figure 6.4 provides a top and side view of the start configuration. A trace of the path followed by the payload is shown in Figure 6.5.

The results for this example are summarized in Table 6.1. The variables in the Table are as defined in Section 6.1. As shown in the table, the total time required to find a path and perform string tightening was just over three minutes. Approximately 60% of the total time involved finding a safe path with the remaining time utilized for string tightening.

The payload for this example is a 0.7 meter long strut, a scale version of the type which might be used to construct space structures such as Space Station Freedom. A long, thin payload such as this highlights the need for consideration to rotational as well as translational degrees of freedom. This path planning problem is potentially difficult because limits on joint 1 prohibit a simple counterclockwise rotation (viewed from above) which would move the payload from start to goal. As a result, the prominent motion is clockwise and escaping from the box-like obstacle near the start requires some backtracking to remove the strut from within the box. Once the strut is out of the box there is also potential for allowing the strut back

	Single Puma (Example 1)	Coop. Pumas (Example 2)	Single 9 DOF (Example 3)	Coop. 9 DOF (Example 4)
N_p	717	524	1124	1307
N_s	112	154	42	78
$\Delta L/L$	20.2	16.4	8.4	14.2
L_f	3.04	2.14	9.35	14.27
N_{sph}	745K	1.72M	1.60M	3.34M
N_{poly}	128K	390K	94.5K	189.5K
t_{path}	114 sec	560	158	167
t_{tight}	71	38	185	384
t_{cc}	101	283	118	247
t_{poly}	69	201	52	115
t_{tot}	185	598	343	551

Table 6.1: Summary of Results for CIRSSE Testbed Examples (times in seconds)

into the box. Similarly, achieving the goal position requires passing the triangular obstacle, aligning the strut for insertion between the sides of the triangle, and performing that insertion. This example also illustrates the fact that concave objects such as the box and the triangle may be easily modeled as several distinct convex polytopes whose combined effect defines a concave task space object.

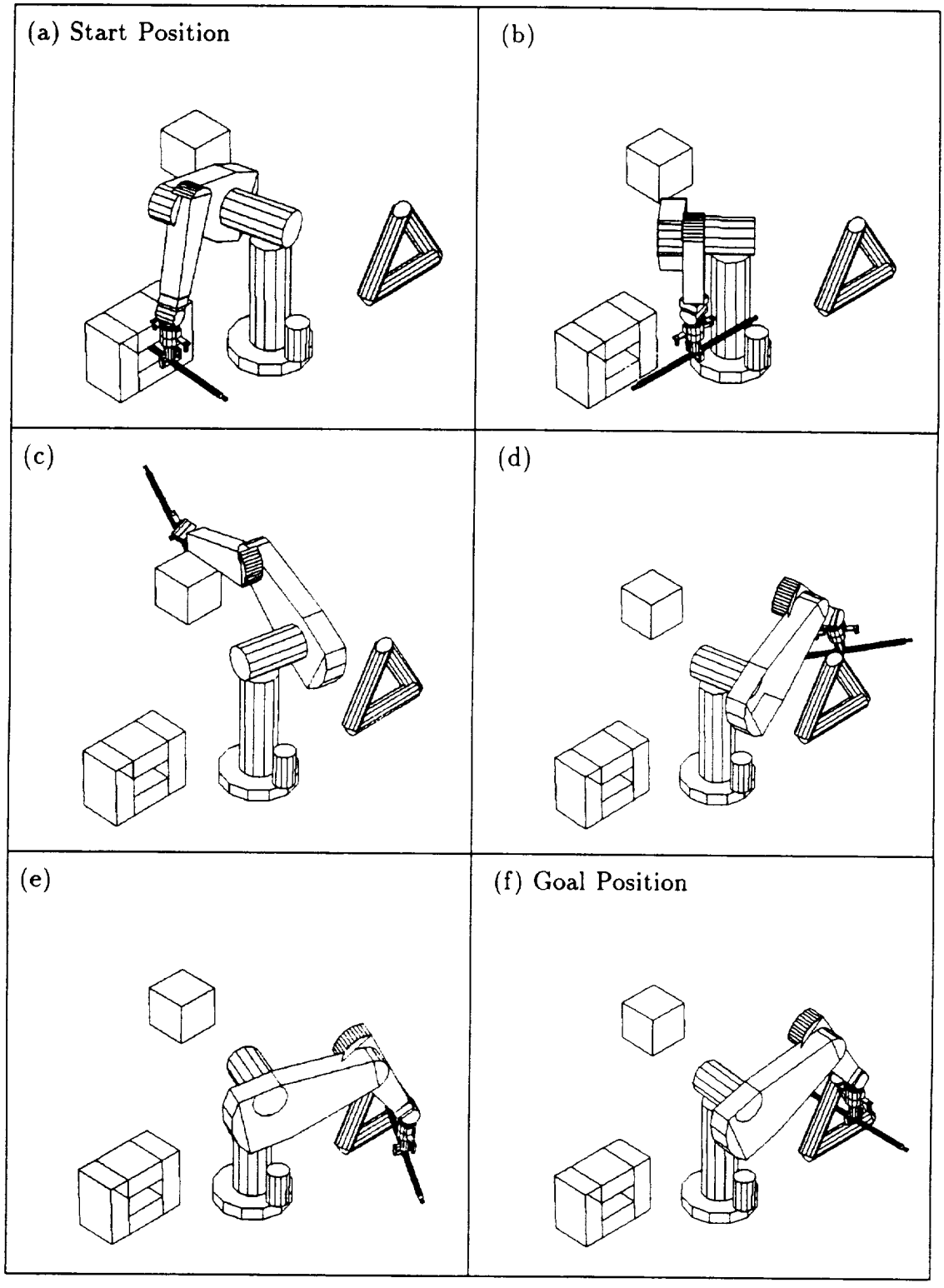


Figure 6.3: Sample Results for Single Puma (Example 1)

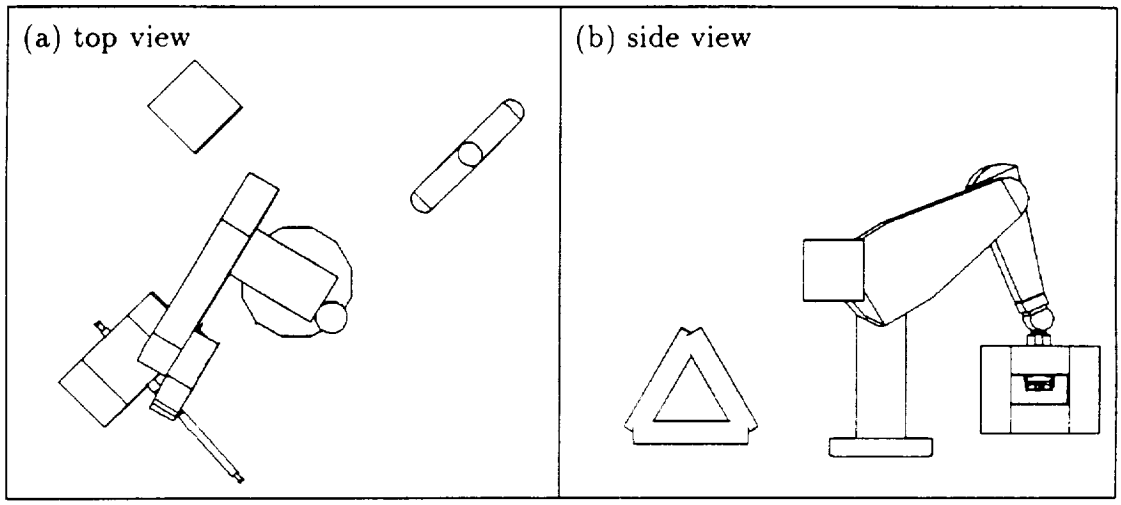


Figure 6.4: Start Configuration for Example 1

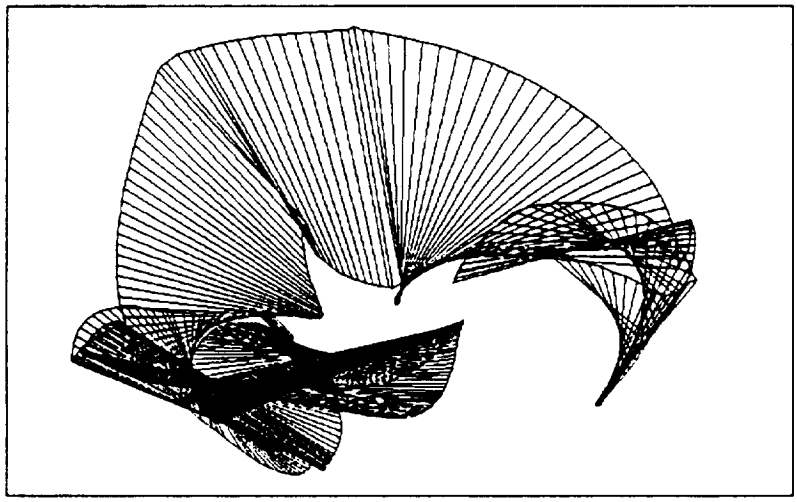


Figure 6.5: Trace of Payload Path for Example 1

6.2.2 Single 9 DOF Robot

The path planner was implemented for a single nine dof robot consisting of one of the testbed's platforms plus the attached Puma. The specific parameters of the implementation are identical to those presented in Section 6.2.1 for a single Puma except for the number of search directions. In the single 9 dof case, the number of search directions is:

$$N_{SD} = 6560 \text{ search directions}$$

6.2.2.1 Example 2

A sample path found by the single 9 dof path planner is shown in Figure 6.6. This problem is identical to the problem in Example 1 except that the extra three dof of the platform may be utilized. The path found by the planner uses the platform translation and tilt capabilities to aid in obstacle avoidance.

The results for this example are summarized in Table 6.1. As shown in the table, the total time required to find a path and perform string tightening was just under ten minutes. The results also show that the redundancy was utilized to produce a path which was approximately 50% shorter than the path obtained for the Puma alone. Over 90% of the total time involved finding a safe path with the remainder of the time utilized for string tightening.

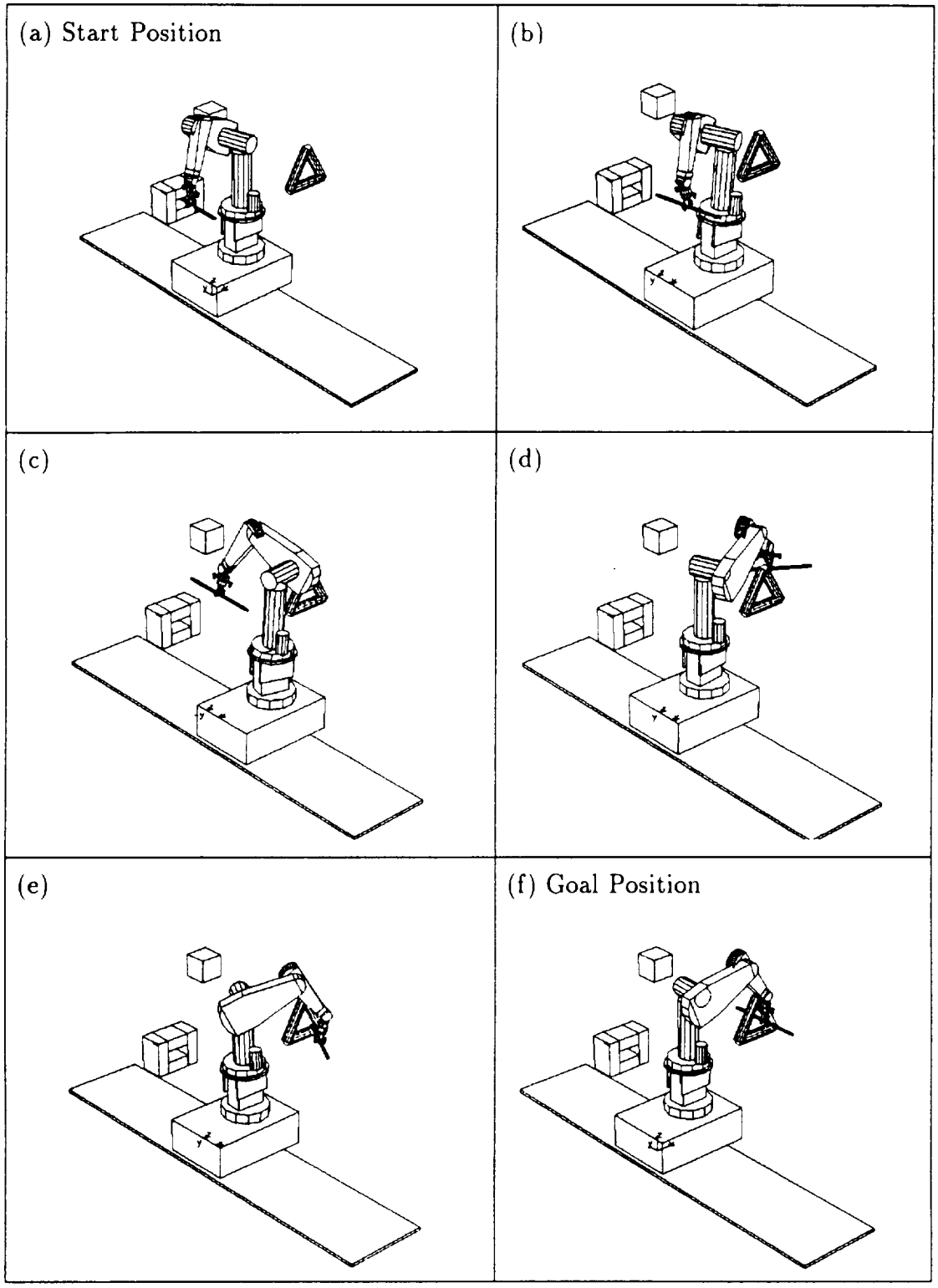


Figure 6.6: Sample Results for Single 9 DOF Robot (Example 2)

6.2.3 Cooperating Puma 560's

Addressed in this implementation is the path planning problem for the two CIRSSE testbed Pumas working cooperatively. Thus, the platforms may be used to initially position the two Pumas but are stationary throughout the planning problem. The specific parameters of the implementation are as follows:

$$\begin{aligned} c &= \frac{1}{300} \text{ step size} \\ N_{SD} &= 242 \text{ search directions} \\ g &= 5 \text{ bins} \\ \lambda &= 0.5 \text{ forgetting factor} \\ r &= 4 \text{ (See Equation 5.3)} \end{aligned}$$

6.2.3.1 Example 3

This example involves a space containing six obstacles arranged in a maze-like fashion. The path planner successfully finds the path shown in Figure 6.7 which traverses from start to goal with no collisions. The payload is a 3cm x 3cm x 110 cm box. The clearance between the long horizontal obstacles is 15cm. Figure 6.8 provides a top and side view of the start configuration. Similarly, Figure 6.9 provides a top and side view of the goal configuration.

The results for this example are summarized in Table 6.1. As shown in the table, the total time required to find a path and perform string tightening was under six minutes. Approximately 46% of the total time involved finding a safe path with the remaining time utilized for string tightening.

This example seems to reflect the maximum level of difficulty which the cooperating Puma path planner as presently implemented can solve within a reasonable amount of time. For example, if the obstacle near the goal end of the passageway between the two long obstacles is lengthened downward by 0.1 meters the planner

fails to find a path (when allowed to try for over an hour). This failure to find a solution occurs even though it is apparent to the user that a solution does exist. Example 3 is also a path planning problem which the planner cannot solve if the start and goal positions are interchanged. In that case the planner begins by going below the open passageway between the long obstacles and then fails to find a path which can circumvent the lowest obstacle. Once in this position, it seems likely that a planner would need to resort to an impractical exhaustive mapping of a huge concavity before determining that significant backtracking would need to take place to find the opening to the passageway.

When difficulty was experienced with the cooperating robot path planner (cooperating 6 and cooperating 9 dof case), the source of the difficulty virtually always turned out to be in the choice of the start and goal robot configurations (i.e., the start and goal joint angles). Such difficulties appear difficult to address intuitively but are easily addressed brute force by trying all combinations of feasible Puma configurations at the start and goal positions. This typically resulted in at least one suitable problem definition for which the planner was successful.

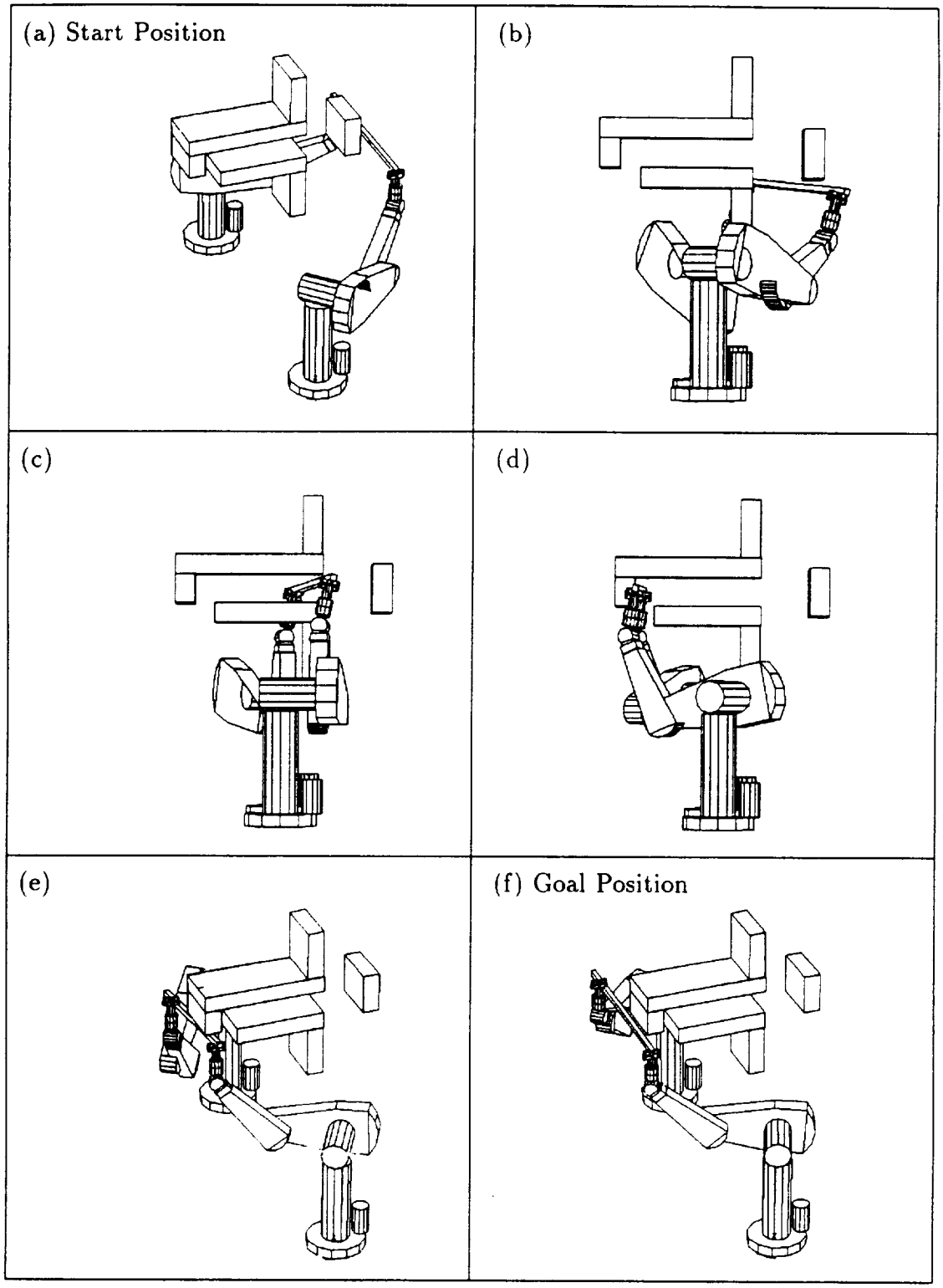


Figure 6.7: Sample Results for Cooperating Pumas (Example 3)

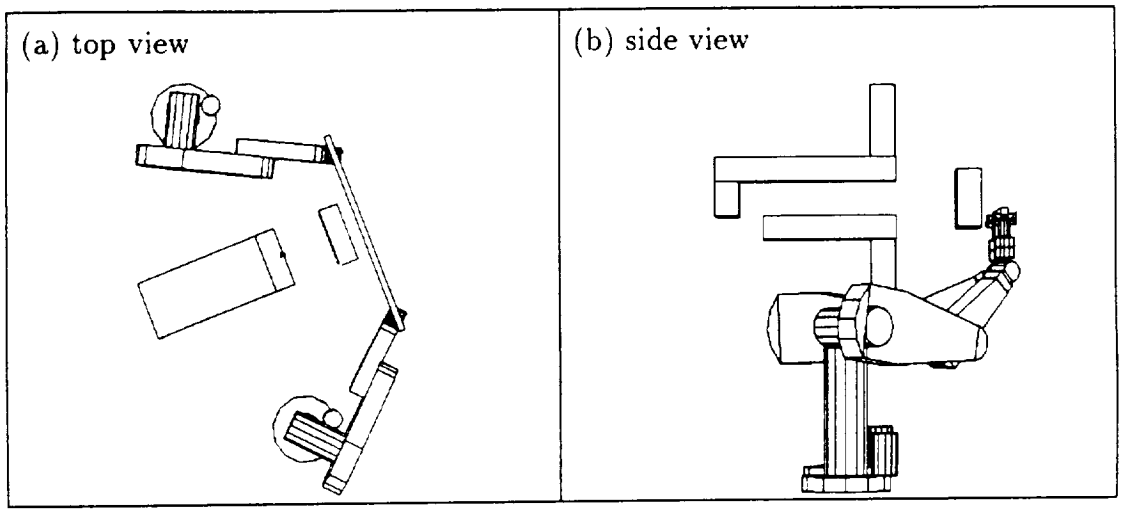


Figure 6.8: Start Configuration for Example 3

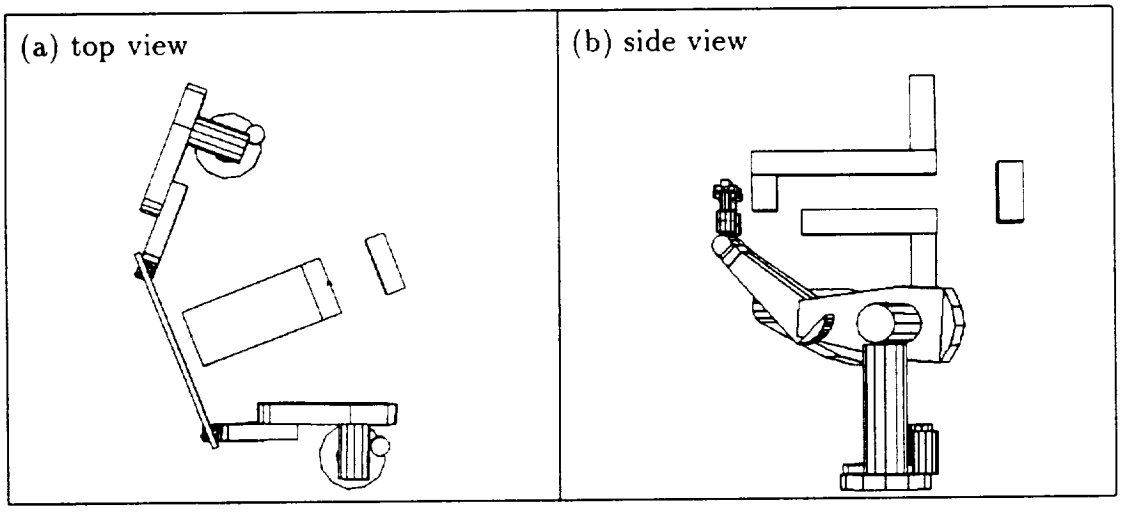


Figure 6.9: Goal Configuration for Example 3

6.2.4 Cooperating 9 DOF Robots

Addressed in this implementation is the path planning problem for the two 9 DOF CIRSSE testbed robots working cooperatively. The specific parameters of the implementation are as follows:

$$\begin{aligned} c &= \frac{1}{150} \text{ step size} \\ N_{SD} &= 2048 \text{ search directions} \\ g &= 10 \text{ bins} \\ \lambda &= 0.5 \text{ forgetting factor} \\ r &= 10 \end{aligned}$$

Recall from Chapter 5 that the c-space traversal heuristic is applied in a 12D space for cooperating 9 dof path planning problems. As a result, the complexity of the cooperating 9 dof robot path planning problem is immensely higher than the complexity of the cooperating 6 dof. This increased complexity would result in $3^{11}-1$ or 177146 search directions using Procedure 4. Since such a number of search directions would be computationally impractical, this implementation utilized the reduced set considering all combination of ± 1 times the basis vectors. This results in 2^{11} or 2048 search directions.

6.2.4.1 Example 4

A sample path found by the cooperating 9 dof robot path planner is shown in Figure 6.10. The start and goal positions appear in the upper left and lower right, respectively. Figures 6.11 and 6.12 provide top and side views of the start and goal configurations, respectively.

The results for this example are summarized in Table 6.1. As shown in the table, the total time required to find a path and perform string tightening was just

under 10 minutes for this example. Approximately 30% of the total time involved finding a safe path with the remaining time utilized for string tightening.

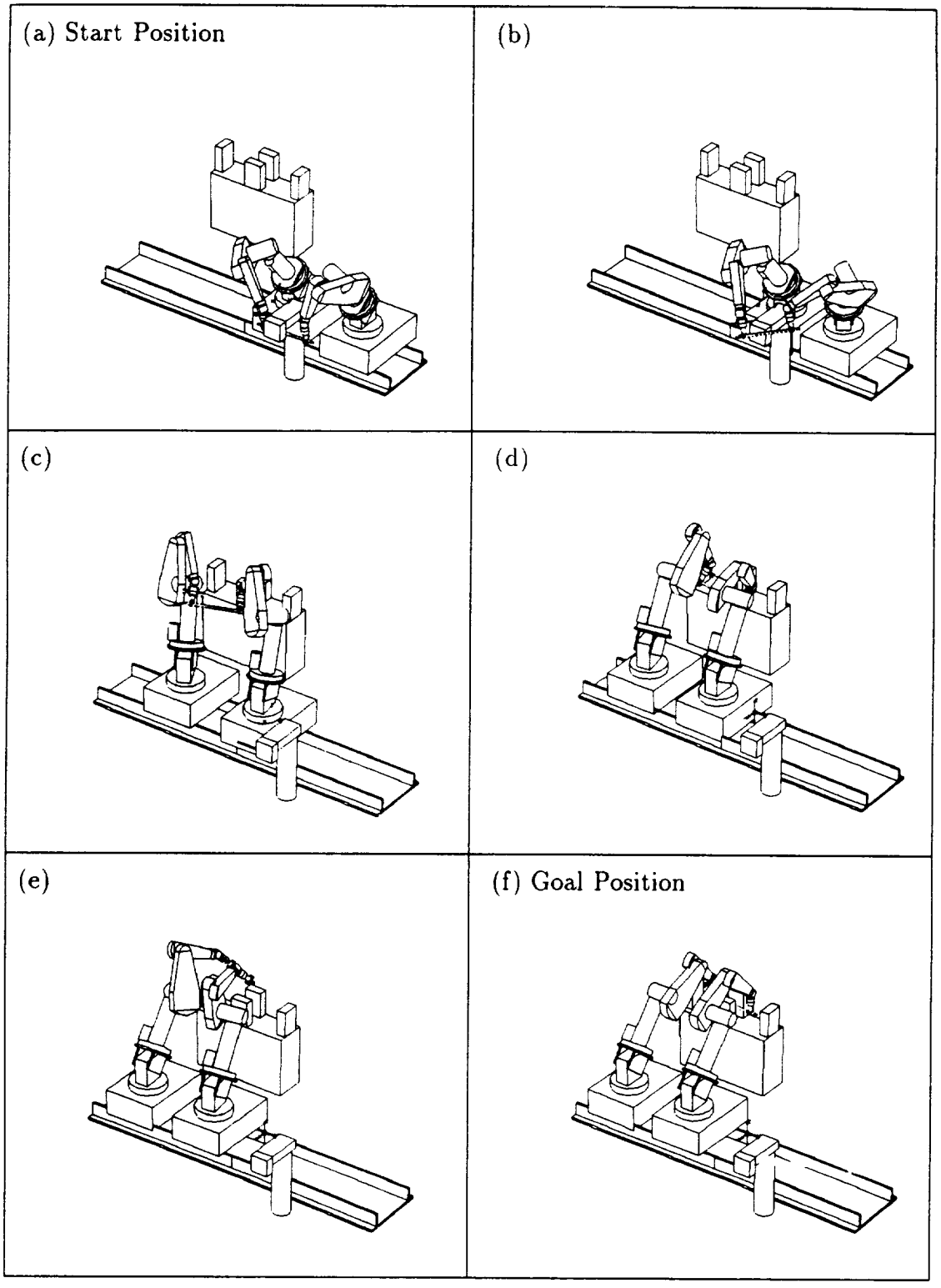


Figure 6.10: Sample Results for Cooperating 9 DOF (Example 4)

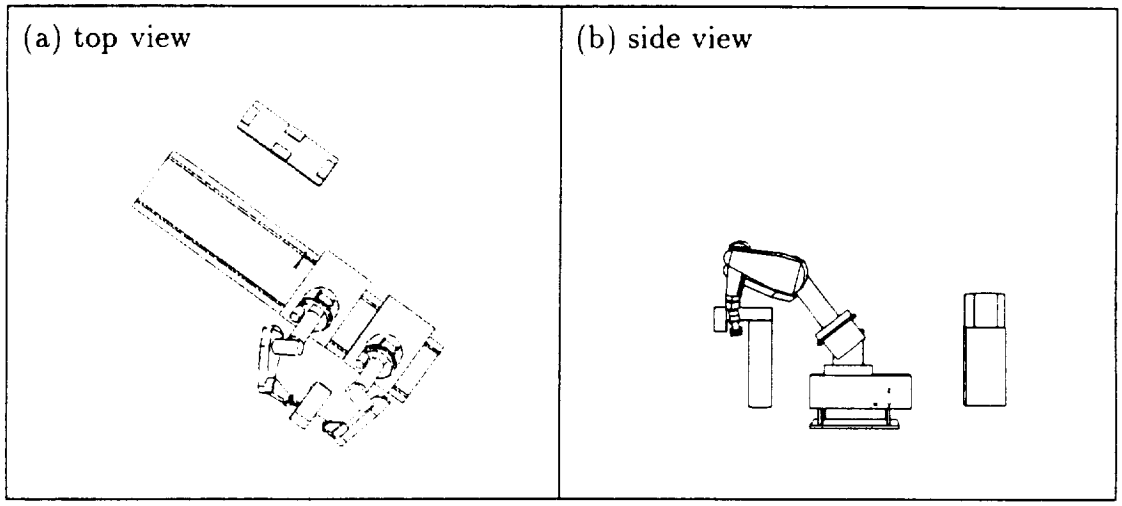


Figure 6.11: Start Configuration for Example 4

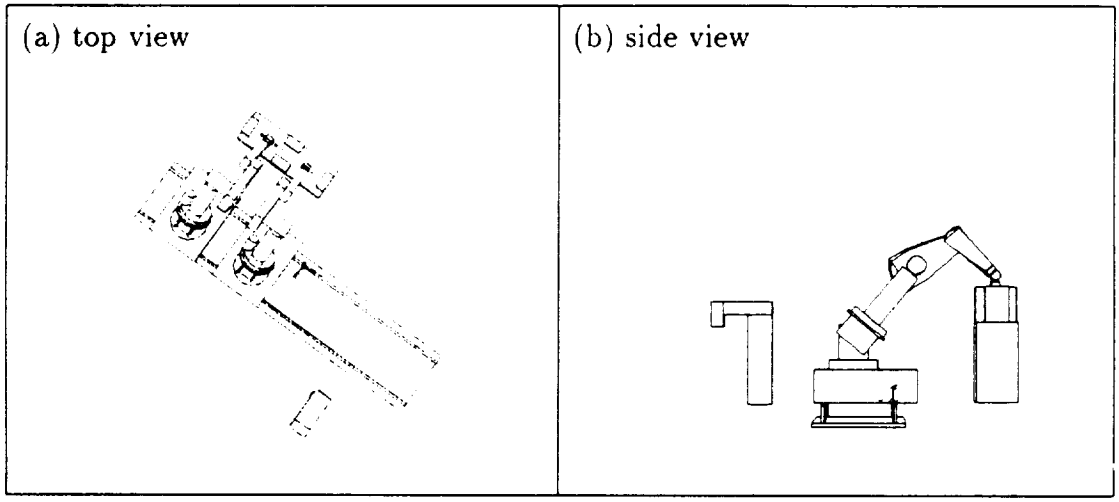


Figure 6.12: Goal Configuration for Example 4

6.2.5 Effect of String Tightening

An example of the effect of string tightening on the payload path for a cooperating nine dof robot path planning problem is shown in Figure 6.13. Parts (a) and (b) of the figure show traces of load positions along the path before and after string tightening, respectively. The string tightening phase required approximately 30 minutes computation time and resulted in a 37% reduction in path length.

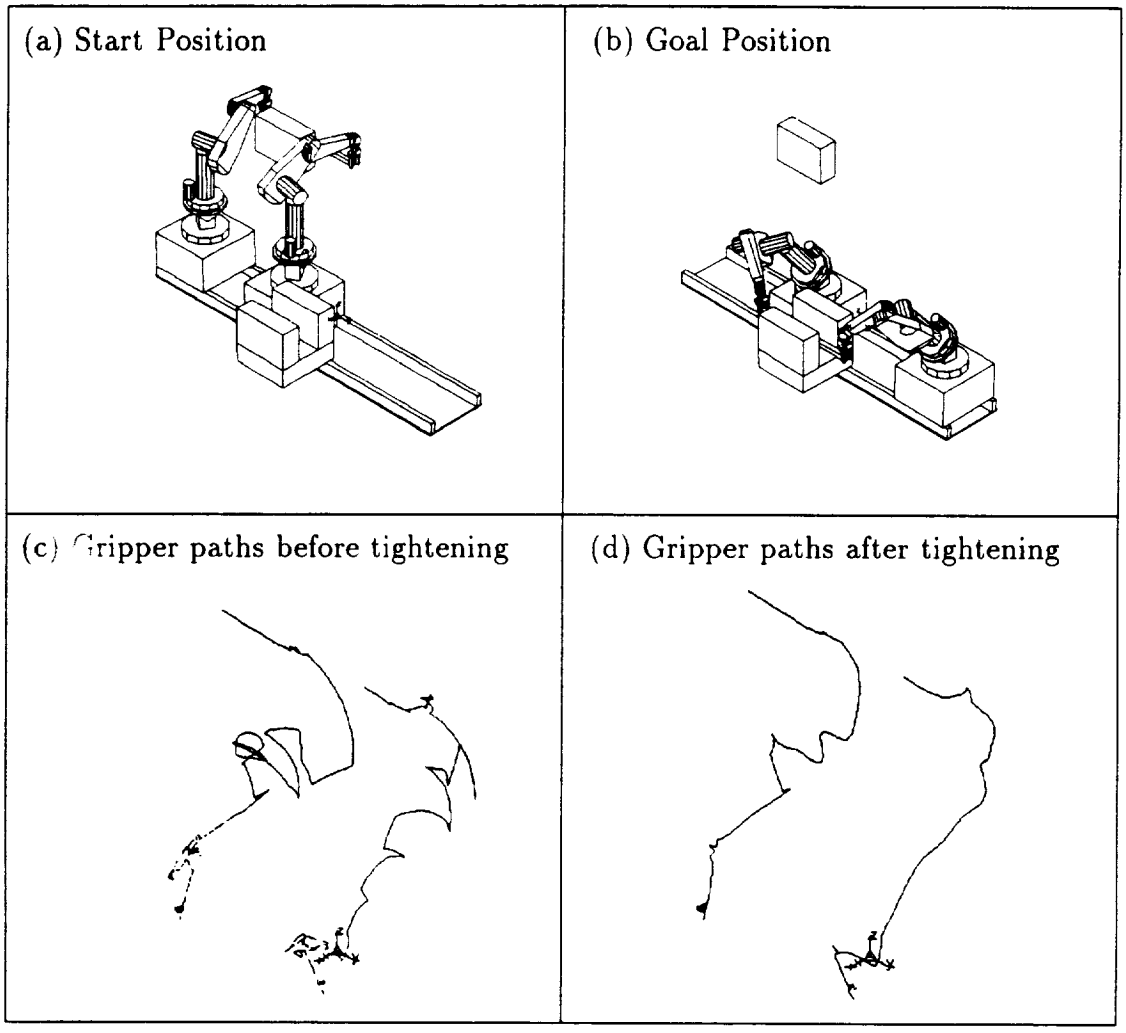


Figure 6.13: String Tightening a Path for Cooperating Nine DOF Robots

6.3 NASA Langley's Automated Structure Assembly Lab

A CimStation model of NASA Langley's Automated Structure Assembly Lab (ASAL) is shown in Figure 6.14. The system consists of a 6 dof Merlin robot, shown in Figure 6.15, mounted to a xy-positioning table (referred to as the carriage), and a turntable. The turntable includes a triangular platform which can rotate around a vertical axis through its center. The Merlin robot is kinematically similar to a Puma. The objective of the ASAL is to assemble truss structures consisting of 102 2 meter long struts. Such a truss is illustrated in Figure 6.16. The truss is assembled upon the turntable of the ASAL by positioning the carriage and the turntable such that the Merlin may take each strut from a canister near the base of the Merlin and install it in its final position in the assembly.

A single arm path planner was implemented for the ASAL environment. The implementation parameters are as follows:

$$\begin{aligned} c &= \frac{1}{400} \text{ step size} \\ N_{SD} &= 242 \text{ search directions} \\ g &= 5 \text{ bins} \\ \lambda &= 0.0 \text{ forgetting factor} \end{aligned}$$

The assembly sequence considered was provided by NASA. The path planner quickly found paths for the first 21 struts since there is little possible interference at that stage. Due to symmetry, the assembly of the remaining 81 struts can be accomplished using only 21 unique trajectories for the Merlin with the appropriate carriage and turntable positions for each strut. The path planner was able to find feasible paths for all 102 struts with solution times ranging from 1 to 30 minutes, with the vast majority of solution times in the 2 to 5 minute range. Since the final approach must be in a specified direction, the goal positions used were 10 cm from the final strut position with the end effector oriented to allow the final insertion to

be performed by a straight task space move.

This implementation of the path planner for the ASAL assembly task illustrates the potential usefulness of the path planning technique presented in this thesis for solving practical, potentially very difficult real-world path planning problems. Some particular comments regarding this implementation follow:

- The path planner has no trouble with goal positions placing the load or robot in very close proximity to obstacles.

- The path planner performs well even with a large number of obstacles. For example, the final few struts of the assembly involve over 100 workspace obstacles. The additional collision checks required near the end of the assembly seem to increase execution time by a factor of approximately two.

- The paths found typically include segments which are obstacle boundary tracing. Because of the close tolerances involved, it is not practical to simply model the objects larger than actual size to provide a safety margin since so doing may result in an unsolvable problem. This shortcoming was noted earlier in Section 5.3.2.2 and possible remedies are addressed in Section 8.2.1.

- The nodes to connect the struts were not modeled. As a result, some of the paths might collide with the nodes if the paths were used in an actual assembly. This could be remedied simply by modeling the nodes and including them in the collision checking routine. Due to the small size of the nodes it is expected that including them would have little impact on the difficulty of the path planning problems.

- In a few cases the path planner was not always able to solve the problem quickly in the forward direction but could quickly solve the problem in the opposite direction. Although a very confined goal position makes it likely that

solving in reverse may prove easier, trial and error was the only sure way to decide which direction would yield better performance.

- Return paths for the robot after inserting a strut were not planned.

6.4 Cooperating Pumas Assemble a Truss

This section describes the implementation of the path planner to a task whereby two Pumas work cooperatively to assemble a 24 strut truss. The workcell for this implementation with the completed truss is shown in Figure 6.17. The pumas are in their start position in Figure 6.17. The workcell includes two Puma 560's which are 500 cm apart and mounted to a carriage. The carriage can translate toward or away from a turntable upon which the truss is assembled. The carriage and turntable are used to position the Pumas and the partially completed truss structure such that the Pumas may insert each strut without concurrent motion of the carriage or turntable. The struts are 133 cm long. The robot end effectors are 100 cm apart when grasping a strut. The parameters for this path planning implementation are as described in Section 6.2.3 for the CIRSSE Pumas.

The planner successfully planned paths for all 24 struts with solution times per strut ranging from less than one minute to approximately 10 minutes. Some points regarding this implementation are as follows:

- Many of the paths found involve multiple arm configurations for one or both Pumas. As a result, the robots pass through many task space singularities.
- There is significant potential for collision between the robots due to their proximity.
- Although the start positions were identical and all the goal task space positions were known, trial and error was typically necessary in order to determine suitable goal Puma configurations which would enable a solution to be found.

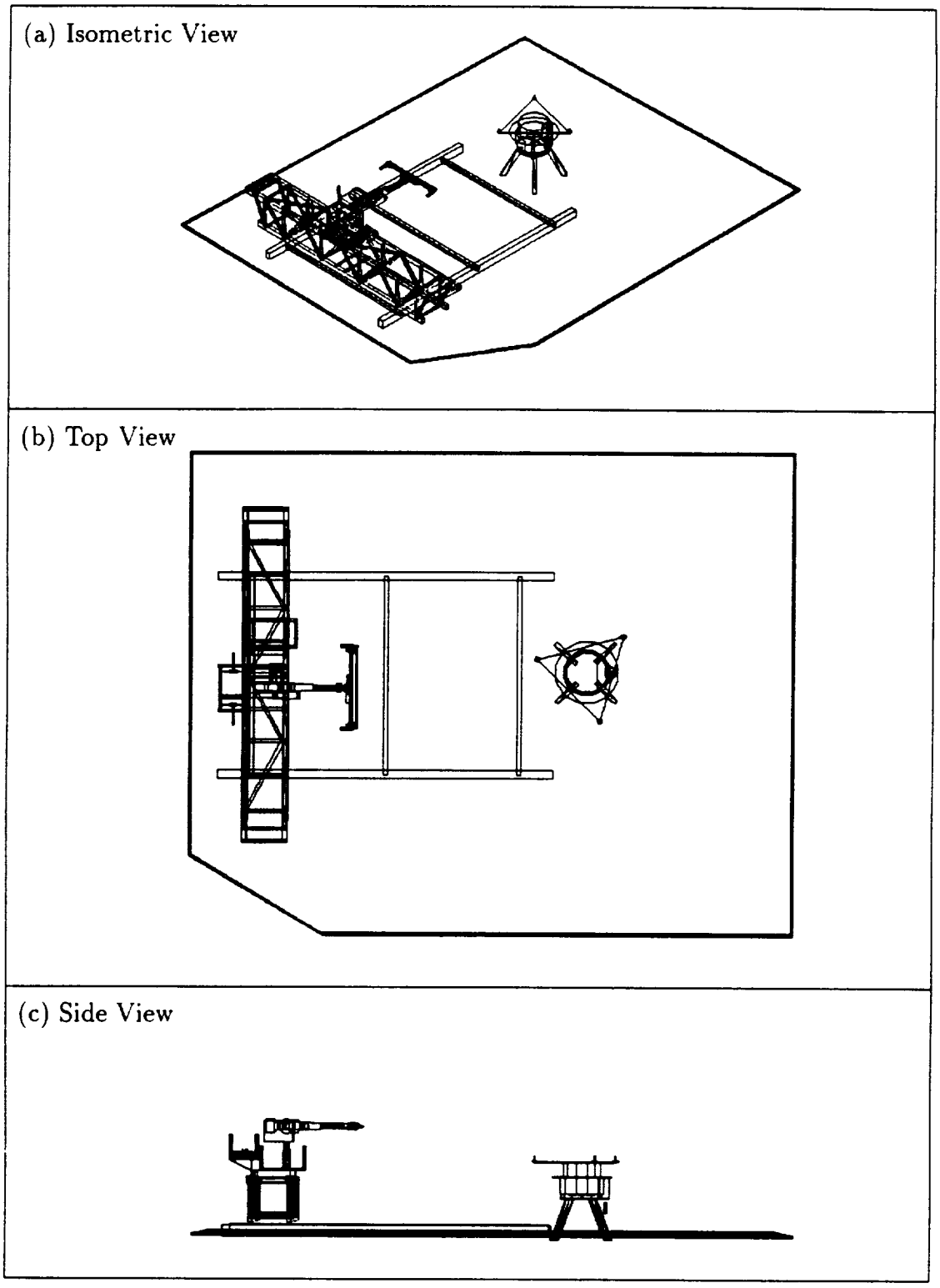


Figure 6.14: NASA Langley's Automated Structure Assembly Lab

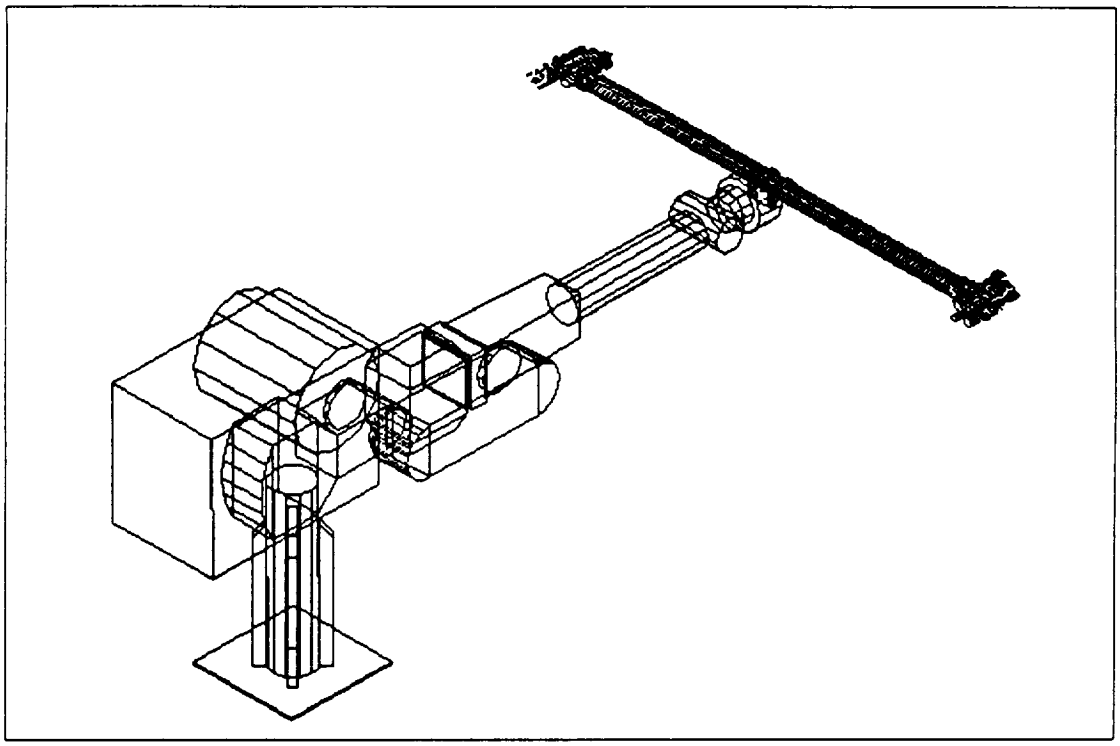


Figure 6.15: 6 DOF Merlin Robot with End Effector for Truss Assembly

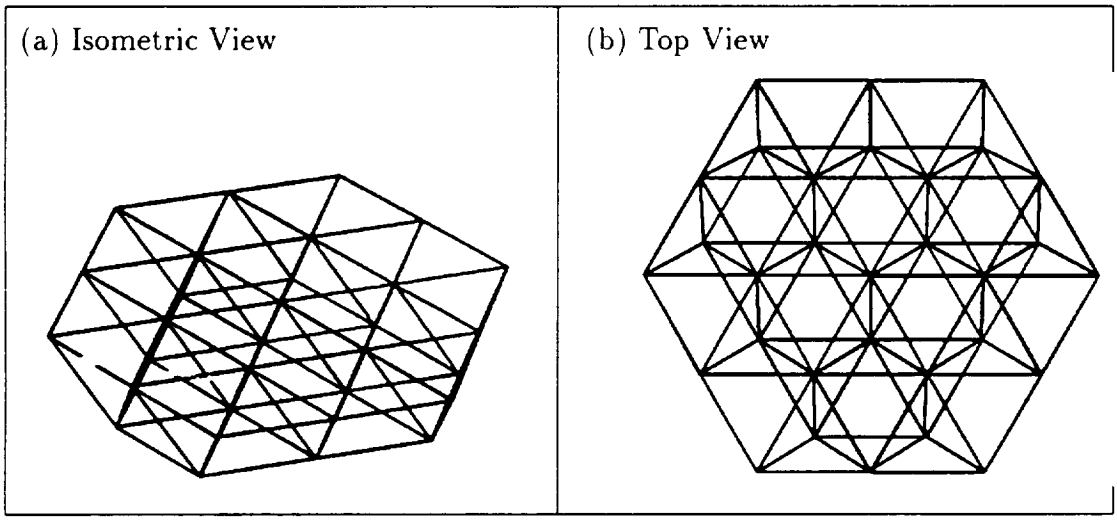


Figure 6.16: 102 Strut Truss Structure

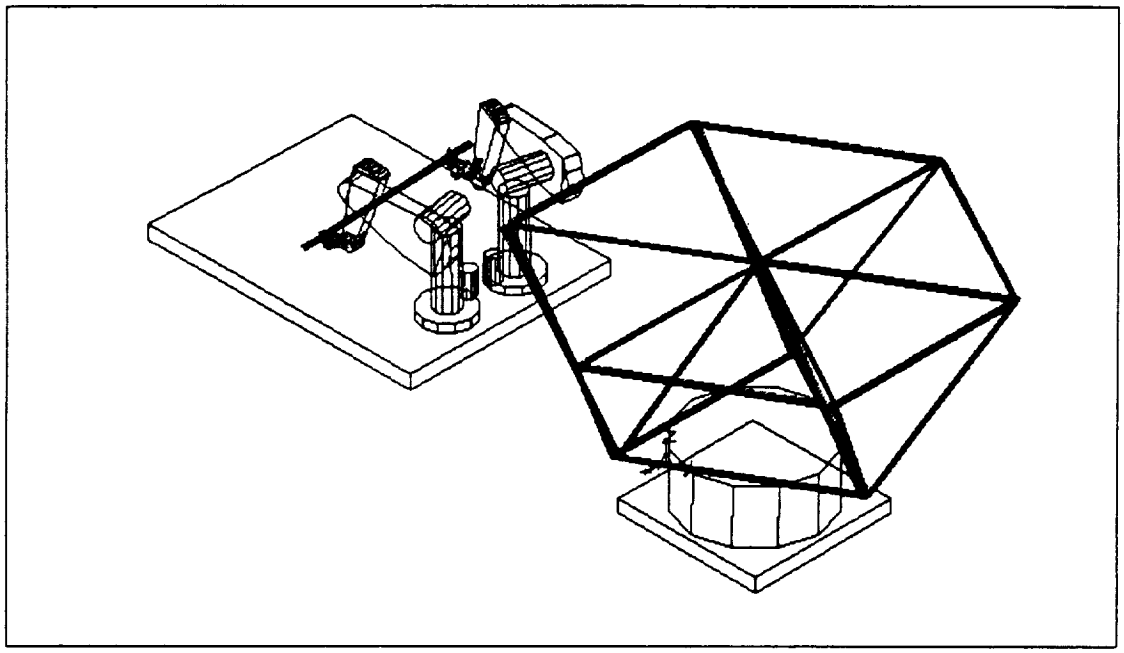


Figure 6.17: Workcell for Cooperating Pumas Assembling Truss

CHAPTER 7
Discussion of the Path Planning Strategy

This chapter discusses the path planning strategy presented in this thesis. This chapter is organized into three main sections:

- Completeness
- Computational Complexity
- Overall Effectiveness

Completeness and computational complexity are discussed in Sections 7.1 and 7.2, respectively. Section 7.3 attempts to judge the overall effectiveness of the strategy.

7.1 Completeness

Unfortunately, the path planning approach is not complete. In other words, the approach does not guarantee that a solution will be found or determine that a solution does not exist. Based on the literature (see Chapter 2), it appears to be difficult to achieve both completeness and practicality for reasonably difficult yet practical path planning problems with more than a few degrees of freedom. Since our emphasis was toward achieving a potentially useful path planner for cooperating robots with at least 6 dof each, we sacrificed completeness in exchange for the possibility (with no guarantees) of solving some practical problems within a reasonable amount of computation time.

This lack of completeness was discussed earlier in Section 4.3.1 where it was shown that the c-space traversal heuristic around which the path planner is based can fail to find a solution even if one may exist due to one of the following scenarios:

- Cycling occurs.
- No safe point is found by the limited set of search directions.

Modifying the heuristic to guarantee finding a safe point if one exists (such as by continually increasing the search resolution) would still not ensure completeness since cycling might still occur. In addition, it was shown in Section 4.4 that performing even one thorough search can be computationally intractable.

Many path planning algorithms such as those based on randomized searches are *probabilistically complete*, meaning that given sufficient computation time they will guarantee finding a solution if one exists. However, such algorithms offer little practical value since they inevitably take a very long time to run for reasonably difficult problems.

7.2 Computational Complexity

Computational complexity of this work can be analyzed by giving an upper or a lower bound on the number of elementary computations or the size of memory required to solve a problem. Recall from Chapter 2 that the n dof robot path planning problem is PSPACE-hard with an upper bound complexity of $O(n^n)$.

This section investigates the computational complexity of the planner in order to determine how an increase in system dof would be expected to affect solution time. The computational complexity of the planner can be addressed in three parts:

- Precomputations
- Mapping a c-space point.
- Performing searches
- Overall Complexity

These parts are discussed below.

The path planning method presented in this thesis requires no precomputations.

Consider a workspace involving an n link robot and m obstacles. Mapping a c-space point involves the following operations:

- Updating the link model
- Checking for joint limit violations
- Checking for collisions

Both updating the link model and checking for joint limit violations have an upper bound complexity $O(n)$. Checking for collisions has a higher upper bound complexity $O(nm)$. Thus, c-space mapping computations grow linearly with both increasing dof and number of obstacles.

The worst case complexity for performing searches will be a linear function of the number of search directions used. For search directions computed as described by Procedure 4 in Chapter 4, an upper bound on search complexity for an n dof problem is $O(k^{n-1})$, where $k < n$. For our implementation, $k = 3$ for problems with a mobility $m \leq 9$ and $k = 2$ for problems with mobility $m = 12$.

An overall upper limit on computational complexity can be taken to be the worst case complexity of the above three operations. Thus, the path planner presented in this thesis has an upper bound on complexity of $O(k^{n-1})$, where $k < n$.

7.2.1 Possible Benefits of Parallel Processing

When mapping along a prescribed vector, parallel processing could be used to map each discretized point along that vector simultaneously. Even more significantly, the various possible search directions and even the different depths in those

directions could be examined simultaneously. Parallel processing could also greatly speed the interference checking by performing multiple checks simultaneously.

A massively parallel machine, such as the *Connection Machine* which has 2^{16} (or 65536) 1-bit processors, could radically decrease the execution time of the path planner presented in this thesis.

7.3 Overall Effectiveness

Relatively few other approaches have appeared in the literature for solving the cooperating robot path planning problem for robots with six dof each. The path planning strategy presented in this thesis appears to be capable of solving more difficult problems than those approaches. In addition, this thesis illustrates that the strategy presented can be practically applied to cooperating nine dof robots. Results in the literature for cooperating redundant robots appear to be limited to planar manipulators. A single arm version of the planner has demonstrated the ability to solve some practical yet potentially very difficult path planning problems in a reasonable amount of time. Some general statements regarding the effectiveness of the path planner follow:

- Solution times are reasonable for off-line programming (typically under 30 minutes).
- Potential problems with joint limits and multiple arm configurations are inherently handled.
- The planner performs well and in reasonable time even with over 100 obstacles.
- The planner is effective even for start and/or goal positions involving little safety clearance.

CHAPTER 8
Conclusions and Future Work

This Chapter presents some conclusions on the subject of this thesis, Section 8.1, and discusses some areas for future work, Section 8.2.

8.1 Conclusions

The general problem of planning collision free paths for an n dof robotic systems has an upper bound on complexity of $O(n^n)$. As a result, exact solutions to the robot path planning problem will likely remain excessively computationally intensive for some time. As a result, any implementation of autonomous robotic path planning which is likely to prove successful in the near future will probably involve some simplifying assumptions, shortcuts, or heuristics. While any inexact solution may fail for some cases, the advantage of this type of approach is that a solution may be found for many practical yet potentially difficult path planning problems with a reasonable amount of computation.

This thesis addressed the problem of planning feasible and obstacle-avoiding paths for two spatial robots working cooperatively in a known static environment. Because of the apparent impracticality of developing a general and complete path planning strategy, the main emphasis of this work involved developing a heuristic based path planner for cooperating robots which sacrifices completeness in exchange for a hope of finding a solution in a reasonable amount of time. The path planning approach presented in this thesis is configuration space (c-space) based and performs selective rather than exhaustive c-space mapping. A novel, divide-and-conquer type of heuristic is used to guide the selective mapping process. Also, a configuration space based algorithm was presented to modify any feasible path found by the planner into a more efficient one.

Although the path planner cannot guarantee finding a solution even if one exists, and in spite of its $O(k^{n-1})$ complexity for n degree of freedom problems (where $k = 2$ or 3 as implemented), it has demonstrated the ability to solve a variety of practical yet potentially difficult path planning problems with a reasonable amount of computation. This thesis presented the implementation details and illustrated sample results for the following four cases: single six dof (6R) robot, single nine dof (1P-8R) robot, cooperating six dof (6R) robots, and cooperating nine dof (1P-8R) robots. The path planning program typically requires under 10 minutes to execute for cooperating six dof robots and under 30 minutes to execute for cooperating nine dof robots. The planner appears to perform better than other cooperating robot path planners in the literature.

Some specific advantages and disadvantages of the path planning technique presented in this thesis are discussed below.

8.1.1 Advantages

1. The planner utilizes selective (non-exhaustive) mapping of c-space thus making it possible to get solutions in a reasonable amount of time.

2. The planner is global in nature but has provision for local navigation around obstacles.

3. The approach is completely general and would, in theory, be applicable to any system of arbitrary dimension. The approach is also independent of the type of geometric representation employed, so long as the chosen representation enables mapping of c-space points on an as-needed basis.

4. Unsafe space is handled in the same manor regardless of the reason for it being unsafe (such as various possible collisions or joint limit violations).

5. The approach could be applied to either single robot or cooperating robot path planning problems.

6. Robot degeneracy is not a concern for single arm problems and is inherently handled for the cooperating arm scenario (see Chapter 5).

7. While the resulting path is generally sub-optimal, it should be feasible to "tighten up" on any safe path to obtain a shorter one (Chapter 5.3).

8. The potential speed of the collision detection is enhanced by the fact that the method simply needs a *yes* or a *no* regarding collisions and does not require distance or direction information.

9. Cooperating redundant robot path planning problems may be handled without requiring use of inverse kinematics for a redundant robot.

10. The bulk of the calculations are such that they could be done in parallel (see Section 7.2).

11. Implementation of the path planner is relatively straightforward and easy.

8.1.2 Disadvantages

1. The planner is heuristic in nature and is not complete, i.e., it cannot guarantee finding a solution even if one may exist. Other approaches which are complete are computationally impractical for reasonably difficult yet practical problems for more than a few dof.

2. Joint angles at the start and goal configurations are required to be specified.

3. There is presently no means to determine that a solution exists other than to find one.

4. The number of strategy directions required to achieve an effective search increase exponentially with dimensionality. This effect may be partially offset by the fact that there may be more acceptable solutions to systems of higher dimensionality making it easier to find one of them.

5. The resulting path may be longer than necessary even after being shortened.

6. The planner cannot be directly applied to cases with dynamic obstacles.

8.2 Future Work

Some potential areas of future work include:

- Improvement to String Tightening Process
- Integration with the CIRSSE Geometric State Manager
- Utilization of Parallel Processing
- Guaranteeing Completeness

These areas of potential future work are discussed below.

8.2.1 Improvement to String Tightening Process

As discussed in Section 5.3.2.2, the string tightening algorithm presented herein has the disadvantage of yielding paths which very nearly involve collision. This issue could be addressed as part of future work by one of the following means:

- Expanding the obstacles so that paths with very little clearance in the model actually provide sufficient clearance. This is not a feasible option when the only safe path involves tight clearances.
- Modifying the cost function (Equation 5.5) to include a penalty for proximity to obstacles and considering knot point movement in any direction orthogonal to the segment between the preceding and following knot points.

- Implementing an alternate approach to string tightening, such as a potential fields approach similar to that discussed in Section 5.3.1. This is a very promising approach since the local minima problem can be effectively eliminated since the path planner provides the potential fields based path smoother with a feasible solution to the global path planning problem.

8.2.2 Integration with the CIRSSE Geometric State Manager

The path planner could be integrated with the CIRSSE Geometric State Manager (GSM) [102]. The purpose of the GSM is to maintain a time-varying geometric model of the CIRSSE robots and their environment. Once the path planner is integrated with the GSM, the planner could automatically obtain the current robot and obstacle information from the GSM when a testbed task determines the need to utilize the path planner.

8.2.3 Utilization of Parallel Processing

The path planning programs are currently implemented using serial coding. As such, the path planning program typically requires under 5 minutes to execute for cooperating six dof robots and under 30 minutes to execute for cooperating nine dof robots. The algorithm being used is ideally suited for parallel processing since each search involves a large number of independent calculations. Implementing the path planning program in parallel could drastically reduce the path planning program execution time.

8.2.4 Guaranteeing Completeness

As discussed earlier, a complete solution to the cooperating spatial robot path planning problem appear to be impractical at this time. Nonetheless, it might be possible to modify the c-space heuristic in such a way as to guarantee completeness.

At present, the usefulness of such an modification is at best questionable. However, advances in both the path planning and computer fields might warrant a second look at the completeness issue sometime in the future.

8.2.5 Decidability

At this time, there does not appear to be an easy answer to the question as to the existence of a solution to a given general path planning problem. Future research advances may make it possible to quickly determine whether or not a solution will exist.

LITERATURE CITED

[1] Reif, John H. Complexity of the Mover's Problem and Generalizations Extended Abstract. In *Proceedings of the 20th Annual IEEE Conference on Foundations of Computer Science*, pages 421–427, 1979.

[2] Schwartz, J.T. and M. Sharir. On the 'Piano Movers' Problem II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds. *Computer Science Technical Report No. 41*, February 1982. Courant Institute, New York University.

[3] Canny, J.F. The complexity of robot motion planning. MIT Press, 1988.

[4] Dooley, J.R. and J.M. McCarthy. Parameterized Descriptions of the Joint Space Obstacles for a 5R Closed Chain Robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1536–1541, 1990. Vol. 3.

[5] Dupont, Pierre E. *Planning Collision-Free Paths for Kinematically Redundant Robots by Selectively Mapping Configuration Space.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1988.

[6] Schima, Francis J. Two Arm Robot Path Planning in a Static Environment Using Polytopes and String Stretching. Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, 1990.

[7] CimStation User's Manual, CimStation 4.3. Silma Inc., Cupertion, CA, 1992.

[8] Akman, Varol. *Shortest Paths Avoiding Polyhedral Obstacles in 3-Dimensional Euclidian Space.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, June 1985.

[9] Andresen, F.P. Visual Algorithms for Autonomous Navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 856–861, St. Louis, MO, March 1985.

[10] Brooks, Rodney A. and Tomas Lozano-Perez. A Subdivision Algorithm in Configuration Space for Findpath with Rotation. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 224–233, March/April 1985. Vol. SMC-15, No. 2.

[11] Brooks, Rodney A. Solving the Findpath Problem by Good Representation of Free Space. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 190–197, March/April 1983. Vol. SMC-13, No. 3.

[12] Brooks, Rodney A. Planning Collision-Free Motions for Pick-and-Place Operations. *International Journal of Robotics Research*, 1983, Vol. 2, No. 4, pp 19–44. Winter.

[13] Canny, J.F. and M.C. Lin. An opportunistic global path planner. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1554–1559, 1990.

[14] R.T. Chien, Ling Zhang and Bo Zhang. Planning Collision-Free Paths for Robotic Arm Among Obstacles. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, January 1984. Vol. PAMI-6, No. 1.

[15] Donald, Bruce R. Hypothesizing Channels Through Free-Space in Solving the Findpath Problem. In *MIT A.I. Memo No. 736*, June 1983.

[16] Donald, Bruce R. On Motion Planning With Six Degrees of Freedom: Solving the Intersection Problem in Configuration Space. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 536–541, St. Louis, MO, March 1985.

[17] Faverjon, Bernard. Object Level Programming of Industrial Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1403–1411, 1986. Vol. 3.

[18] Faverjon, Bernard. Obstacle Avoidance Using an Octree in the Configuration Space of the Manipulator. In *Proceedings of International Conference on Robotics*, pages 504–512, Atlanta, GA, March 1984.

[19] Gouzenes, Laurent. Strategies for Solving Collision-Free Trajectories Problems for Mobile and Manipulator Robots. *International Journal of Robotics Research*, 1984, Vol. 3, No. 4, pp 51–65. Winter.

[20] Hasegawa, Tsutomu. Collision Avoidance Using Characterized Description of Free Space. *'85 ICAR*, 1985, pages 69–76.

[21] Kambhampati, S. and L.S. Davis. Multi-Resolution Path Planning for Mobile Robots. *IEEE Journal of Robotics and Automation*, September 1986, Vol. RA-2, No. 3, pp 135–145.

[22] D.T. Kuan, J.C. Zamiska and R.A. Brooks. Natural Decomposition of Free Space for Path Planning. In *IEEE International Conference on Robotics and Automation*, pages 168–173, St. Louis, MO, March 1985.

[23] Laugier, C. and F. Germain. An Adaptive Collision-Free Trajectory Planner. *'85 ICAR*, 1985, pages 33–41.

[24] Lozano-Perez, T. Spatial Reasoning in the Planning of Robot Motions. *Proceedings of the 1981 Joint Automatic Control Conference*, June 1981, pages WP-2D.

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

[26] Lozano-Perez, Tomas. A Simple Motion Planning Algorithm for General Robot Manipulators. *IEEE Journal of Robotics and Automation*, June 1987, Vol. RA-3, No. 3, pp 224–238.

[27] Park, W.T. State Space Representations for Coordination of Multiple Manipulators. *Proceedings 14th International Symposium on Industrial Robots, 7th International Conference on Industrial Robot Technology*, October 1984, pages 397–405.

[28] Red, W.E. Configuration Maps for Robot Task Planning in 3-D. *Computers in Engineering 1984*, 1984, pages 115–124.

[29] Udupa, S. Collision Detection and Avoidance in Computer Controlled Manipulators. Ph.D. Dissertation, Department of Electrical Engineering, California Institute of Technology, 1977.

[30] Wong, E.K. and K.S. Fu. A Heirarchical-Orthogonal Space Approach to Collision-Free Path Planning. *Proceedings of the IEEE International Conference on Robotics and Automation*, March 1985, pages 506–511. St. Louis, MO.

[31] Dittenberger, Kurt. *Graph Decomposition and Retraction: An Approach to Collision-Free Path Planning.* PhD thesis, Rensselaer Polytechnic Institute, 1990.

[32] Sharir, Micha. Algorithmic Motion Planning in Robotics. *IEEE Symposium on Robotics and Automation*, 1989, pages 9–19.

[33] Lozano-Perez, T. and M. Wesley. An Algorithm for Planning Collision Free Paths Among Polyhedral Objects. *Comm. ACM*, 1979, Vol. 22, pp 560–570.

[34] T.H. Cormen, C.E. Leiserson and R.I. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, New York, New York, 1990.

[35] Branicky, M.S. and W.S. Newman. Rapid Computation of Configuration Obstacles. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 304–310, 1990.

[36] Paden, B., A. Mees, and M. Fisher. Path planning using a Jacobian-based freespace generation algorithm. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1732–1737, 1989.

[37] Kondo, K. Motion planning with six degrees of freedom by multstrategic bidirectional heuristic free-space enumeration. In *IEEE Transactions on Robotics and Automation*, pages 267–277, June 1991. Vol. 7, No. 3.

[38] Chen, Pang C. and Yong K. Hwang. SANDROS: A motion planner with performance proportional to task difficulty. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2346–2353, Nice, France, May 1992.

[39] Herman, Martin. Fast, Three-Dimensional Collision-Free Motion Planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1056–1063, 1986. Vol. 2.

[40] Herman, Martin. Fast Path Planning in Unstructured, Dynamic, 3-D Worlds. unpublished manuscript, Robot Systems Division, National Bureau of Standards, January, 1986.

[41] Lee, B.H. and Y.P. Chien. Time-Varying Obstacle Avoidance for Robot Manipulators Approaches and Difficulties. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1610–1615, 1987. Vol. 3.

[42] Lee, B.H. and Y.P. Chien. Time-Varying Obstacle Avoidance for Robot Manipulators: Approaches and Difficulties. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1987, Vol. 3, pp 1610–1615.

[43] Gupta, Kamal Kant. Fast Collision Avoidance for Manipulator Arms: A Sequential Search Strategy. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1724–1729, 1990.

[44] Lewis, R.A. Autonomous Manipulation of a Robot: Summary of Manipulator Software Functions. Jet Propulsion Laboratory Technical Memorandum 33-679, March 15 1974.

[45] Pieper, D. *The Kinematics of Manipulators Under Computer Control.* PhD thesis, Stanford University, 1969.

[46] Glavina, Bernhard. Solving findpath by combination of goal-directed and randomized search. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1718–1723, 1990.

[47] Yap, Chee-Keng. How to Move a Chair Through a Door. *IEEE Journal of Robotics and Automation*, June 1987, Vol. RA-3, No. 3, pp 172–181.

[48] Rovetta, Alberto and Remo Sala. Robot motion planning with parallel systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2224–2229, Nice, France, May 1992.

[49] Brooks, R.A. Solving the find-path problem by good representation of free space. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 190–197, March/April 1983. Vol. SMC-13, No. 2.

[50] Canny, John. A Voronoi Method for the Piano Movers Problem. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 530–535, St. Louis, MO, March 1985.

[51] Donald, Bruce R. Motion Planning with six degrees of freedom. In *MIT A.I. Memo No. 791*, 1984.

[52] Lumelsky, Vladimir J. and K. Sun. Gross Motion Planning for a Simple 3-D Articulated Robot Arm Moving Amidst Unknown and Arbitrarily Shaped Objects. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1987, Vol. 3, pp 1929–1934.

[53] Lumelsky, Vladimir J. and A. Stepanov. Path Planning Strategies for a Traveling Automaton in an Environment with Uncertainty. Center for Systems Science Technical Report No. 8504, Electrical Engineering, Yale University, April 1985.

[54] Lumelsky, Vladimir J. On Dynamic Path Planning for a Planar Robot Arm. Center for Systems Science Technical Report No. 8505, Electrical Engineering, Yale University, April 1985.

[55] Lumelsky, Vladimir J. Continuos Motion Planning in Unknown Environment for a 3-D Cartesian Robot. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986, Vol. 3, pp 1050–1055.

[56] Lumelsky, Vladimir J. Effect of Kinematics on Motion Planning for Planar Robot Arms moving Amidst Unknown Obstacles. *IEEE Journal of Robotics and Automation*, June 1987, Vol. RA-3, No. 3, pp 207–223.

[57] Petrov, A.A. and I.M. Sirota. Obstacle Avoidance by a Robot Manipulator Under Limited Information About the Environment. *Automatic Remote Control*, April 1983, Vol. 44, No. 4, pp 431–440.

[58] Warren, C.W. Visual Algorithms for Autonomous Navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1021–1026, Sacramento, CA, April 1991.

[59] Lee, C.T. and P.C.Y.Sheu. A Divide-and-Conquer Approach with Heuristics of Motion Planning for a Cartesian Manipulator. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 929–944, September/October 1992. Vol. SMC-15, No. 2.

[60] Koichi Kondo, et al. Motion Planning in Plant CAD Systems. Toshiba Corp. ME R&D Center, 4-1, Kanagawa Pref. 210, Japan.

[61] Khatib, O. and J.F. Lemaitre. Dynamic Control of Manipulators Operating in a Complex Environment. *3d CISM IFToMM Symposium on Theory and Practice of Robot Manipulators*, September 1978.

[62] Hogan, N. Impedance Control: An approach to Manipulation. In *ASME Transactions on Dynamic Systems, Measurement, and Control*, volume 107, pages 1–24, March 1985.

[63] Khosla, P. and R. Volpe. Superquadric artificial potentials for obstacle avoidance and approach. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1778–1784, 1988.

[64] Okutomi, M. and M. Mori. Decision of robot movement by means of a potential field. In *Advanced Robotics*, volume 1, pages 131–141, 1986.

[65] Warren, Charles W. Global Path Planning Using Artificial Potential Fields. In *IEEE International Conference on Robotics and Automation*, pages 316–321, 1989.

[66] Hirukawa, H. and S. Kitamura. A Collision Avoidance Algorithm for Robot Manipulators Using the Potential Method and Safety First Graph. In *Japan-U.S.A. Symposium on Flexible Automation*, pages 99–102.

[67] Aerospace, Martin Marietta Denver. Phase I - Intelligent Task Automation. Air Force Wright Aeronautical Laboratories, Technical Report AFWAL-TR-85-4062, Vol. 3, pp. 194-208, 214-215, April 1986.

[68] Meyers, J.K and G.J. Agin. A Supervisory Collision Avoidance System for Robot Controllers. *Robotics Research and Advanced Applications*, 1983, pages 225–232. W.J. Book, editor, ASME, New York, NY.

[69] Myers, J.K. Multi-Arm Collision Avoidance Using a Potential Field Approach. SRI International, Menlo Park, CA, 1983.

[70] Munger, Rolfe. Path Planning for Assembly of Strut-Based Structures. Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, May 1991.

[71] Warren, C.W. et al. An approach to manipulator path planning. *International Journal of Robotics Research*, October 1989, Vol. 8, No. 5, pp 87–95.

[72] Warren, C.W. A vector based approach to robot path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, April 1991. Sacramento, CA.

[73] Kim, Jin-Oh. Real-Time Obstacle Avoidance Using Harmonic Potential Functions. In *IEEE Transactions on Robotics and Automation*, pages 338–349, June 1992. Vol. 8, No. 3.

[74] Krogh, B.H. A generalized potential field approach to obstacle avoidance control. In *Proceedings SME Conference on Robotics Research*, Bethlehem, PA, August 1984.

[75] Burns, C.I. Connolly J.B. and R. Weiss. Path lanning using Laplace's equation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2102–2106, Cincinatti, OH, May 1990.

[76] Rimon, E. and D.E. Koditschek. Exact robot navigation using artificial potential fields. In *IEEE Transactions on Robotics and Automation*, pages 501–518, October 1992. Vol. 8, No. 5.

[77] Faverjon, B. and P. Tournassoud. A Local Approach for Path Planning of Manipulators with a High Number of Degrees of Freedom. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1152–1159, 1987.

[78] Barraquand, Jerome and Jean-Claude Latombe. A Monte-Carlo Algorithm for Path Planning With Many Degrees of Freedom. In *Proceedings of the IEEE International Conference on Robotics and Automation*, page 1712, 1990. Vol. 3.

[79] Lozano-Perez, T. et al. Task-Level Planning of Pick-and-Place Robot Motions. *Computer*, 1989, Vol. 22, No. 3,.

[80] Derby, Stephen J. *Kinematic Elasto-Dynamic Analysis and Computer Graphics Simulation of General Purpose Manipulators.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, 1982.

[81] Hornick, M.L. and B. Ravani. Computer-Aided Off-Line Programming of Robot Motion. *International Journal of Robotics Research*, 1986, Vol. 4, No. 4,. Winter.

[82] Stobart, R.K. Geometric Tools for the Off-Line Programming of Robots. *Robotica*, 1987, Vol. 5, pp 273–280.

[83] Han, D. et al. Computer-aided off-line planning of robot motion. *Robotics and Autonomous Systems*, 1991, Vol. 7, pp 67–72.

[84] Weisbin, C.R. and M.D. Montemerlo. NASA's telerobotics research program. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2653–2666, Nice, France, May 1992.

[85] Chien, Yung-Ping and Qing Xue. Path planning for two planar robots moving in unknown environment. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 307–317, March/April 1992. Vol. SMC-22, No. 2.

[86] Koga, Yoshihito and Jean-Claude Latombe. Experiments in Dual-Arm manipulation planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2238–2245, Nice, France, May 1992. Vol. 3.

[87] Seereeram, Sanjeev and John T. Wen. A global approach to path planning for redundant manipulators. In *Proceedings of the 1992 Regional Control Conference*, pages 101–104, Brooklyn, NY, 1992.

[88] Lim, Joonhong and Dong H. Chyung. Admissible Trajectory Determination for Two Cooperating Robot Arms. *Robotica*, 1988, Vol. 6, pp 107–113.

[89] Hu, Yan-Ru and Andrew A. Goldenberg. Dynamic control of multple coordinated redundant robots. In *IEEE Transactions on Systems, Man, and Cybernetics*, pages 568–574, May/June 1992. Vol. SMC-22, No. 3.

[90] Bodduluri, Radhika Mohan. *Design and Planned Movement of Multi-Degree of Freedom Spatial Mechanisms.* PhD thesis, University of California, Irvine, 1990.

[91] Chen, Jau-Liang and Joseph Duffy. Path Generation for Two Cooperative Puma Robots. In *Robotics, Spatial Mechanisms, and Mechanical Systems, ASME*, volume DE-45, pages 195–201, 1992.

[92] McKerrow, P.J. *Introduction to Robotics.* Addison–Wesley, Reading, MA, 1991.

[93] Hwang, Y.K. and Narendra Ahuja. Gross Motion Planning - A Aurvey. *ACM Computing Surveys*, September 1992, Vol. 24, No. 3, pp 219–291.

[94] Fu, K. S., R.C. Gonzalez, and C. S. G. Lee. *Robotics: Control, Sensing, Vision, and Intelligence.* McGraw-Hill Book Company, New York, New York, 1987.

[95] J.E. Bobrow, S. Dubowsky and J.S. Gibson. Time-Optimal Control of Robotic Manipulators Along Specified Paths. *International Journal of Robotics Research*, 1985, Vol. 4, No. 3, pp 3–17. Fall.

[96] Bryson, A.E. Jr. and Y.C. Ho. *Applied Optimal Control.* Hemisphere Publishing, Washington, D.C., 1975.

[97] Thorpe, C.E. Path Relaxation: Path Planning for a Mobile Robot. CMU-RI, TR-84-5, 1984.

[98] A Representation Scheme for Rapid 3-D Collision Detection. CIRSSE Document No. 9, 1988.

[99] Hamlin, G.J. and R.B. Kelley. Efficient Distance Calculation using the Spherically-Extended Polytope (S-tope) Model. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2502–2507, Nice, France, May 1992. Vol. 3.

[100] Hron, Anna B. Graphical Interface Between the Cirsse Testbed and Cimstation with MCS/CTOS. Master's thesis, Rensselaer Polytechnic Institute, Troy, NY, 1992.

[101] Testbed Kinematic Frames and Routines. CIRSSE Technical Memo No. 1, March 1991.

[102] The Geometric State Manager. CIRSSE Technical Memo No. 21, December 1992.

APPENDIX A
CIRSSE Testbed Kinematic Frames

This appendix describes the CIRSSE Testbed kinematic frames and the joint limits.

The first section describes how the coordinate frames are assigned and numbered. Section 2 defines the pose names. For reading ease, angular data presented in this appendix is given in units of degrees.

A.1 Coordinate Frames

This section describes the conventions related to the coordinate frame assignments for the 18-DOF Testbed. This section includes a set of standard labels for the coordinate frames and numbers for the joints. The joint ranges implied by the coordinate frame assignment are also given.

A.1.1 Assignment/Labeling of Frames

The consistent numbering of the joints in the Testbed results in a convention for referring to the joints by a standard set of labels. The designed convention specifies one uniform assignment of the coordinate frames, whereby each frame is associated with a single joint and each joint is associated with a single frame, (i.e., there are no redundant frames). Although the frame assignments and their association with the joints are unique, there are two different ways to number each frame/joint. This results in two different sets of frame/joint labels: one to account for an 18-DOF experiment, and one to account for a 9-DOF experiment.

The assignment of frame 0, i.e., the global origin, is made on top of the back platform rail in the middle of its length. The positive x-axis of this frame points towards the other platform rail, the positive y-axis points to the right of the Testbed, and the positive z-axis points towards the ceiling. Scribe marks will be placed on

the back rail to indicate this coordinate frame's origin, positive x-axis, and positive y-axis.

The coordinate frame numbering starts with the left cart, continues through the left PUMA, and then includes the right cart and right PUMA. The coordinate frames associated with the PUMA joints are ordered in the standard way. During an 18-DOF experiment, the frame/joint labels G_1 through G_{18} are used sequentially in the manner just described (G indicates global). During a 6- or 9-DOF experiment, the frame, joint labels are L_1 through L_9, or R_1 through R_9, depending on whether the left or right PUMA+cart is used, respectively. Note, there is no reduced classification of the frame labels beyond those for a single PUMA+cart. Thus, a PUMA only experiment will use joints numbered L_4 through L_9 or R_4 through R_9. The following table summarizes the numbering and labeling of the coordinate frames, and gives the hardware joint limits for the PUMA (rounded to the nearest degree).

frame number	name of associated axis	global label	local label	physical limit associated joint
0	n/a	G_0	L_0, R_0	n/a
1	left cart linear	G_1	L_1	(−1.3716, 0.6096) m
2	left cart rotate	G_2	L_2	(−150, 150) degs
3	left cart tilt	G_3	L_3	(−45, 45) degs
4	left PUMA shoulder	G_4	L_4	(−256, 79) degs
5	left PUMA upper-arm	G_5	L_5	(−221*, 40*) degs
6	left PUMA fore-arm	G_6	L_6	(−60, 246) degs
7	left PUMA wrist	G_7	L_7	(−126, 150*) degs
8	left PUMA flange tilt	G_8	L_8	(−100, 100) degs
9	left PUMA flange rotate	G_9	L_9	(−290*, 290*) degs
10	right cart linear	G_{10}	R_1	(−0.6096, 1.3716) m
11	right cart rotate	G_{11}	R_2	(−150, 150) degs
12	right cart tilt	G_{12}	R_3	(−45, 45) degs
13	right PUMA shoulder	G_{13}	R_4	(−253, 83) degs
14	right PUMA upper-arm	G_{14}	R_5	(−221*, 43*) degs
15	right PUMA fore-arm	G_{15}	R_6	(−60, 243) degs
16	right PUMA wrist	G_{16}	R_7	(−134, 153*) degs
17	right PUMA flange tilt	G_{17}	R_8	(−100, 100) degs
18	right PUMA flange rotate	G_{18}	R_9	(−290*, 290*) degs

The numbers marked with * indicate those limits which are **not** the mechanical limits of the joint but the **encoder** limits. In either case, a hardware limit has been reached. Beyond an encoder limit, the encoder count exceeds the storage capacity of a 'C' short, causing a sign change in the encoder value. This would have serious repercussions for any real-time control code.

The coordinate frame assignment follows a Modified Denavit-Hartenberg formulation, whereby the i^{th} frame is attached to the i^{th} link and has its origin on the i^{th} joint axis, (ref., Craig, J. J., *"Introduction to Robotics Mechanics and Control,"* Addison-Wesley, 1986, Chapter 3). Note that motion of a given joint throughout its entire range does not guarantee lack of collisions; this is particularly true with the linear joints of the carts.

Two figures attached to the end of this memorandum illustrate the coordinate frame assignment. Figure A.1 shows all 18 coordinate frames and joints for the carts and PUMAs. Figure A.2 shows a closer view of the coordinate frames for the left PUMA+cart.

The kinematic parameters for one of the PUMA+cart pairs are given in the following table. Entries preceded by an asterisk indicate the currently accepted approximate values which may change at a later date.

frame number, i	α_{i-1}	a_{i-1} (m)	d_i (m)	θ_i
1	-90^o	*0.32000	q_1	0^o
2	90^o	0.00000	*0.54400	q_2
3	-90^o	0.00000	0.00000	q_3
4	90^o	0.00000	*0.82800	q_4
5	-90^o	0.00000	0.24300	q_5
6	0^o	0.43182	-0.09391	q_6
7	90^o	-0.02031	0.43300	q_7
8	-90^o	0.00000	0.00000	q_8
9	90^o	0.00000	0.00000	q_9

Note that frames 7, 8, and 9 have co-located origins. Specifically, the last frame is *not* located at the flange of the PUMA's wrist. Numerical detail for the

transformation from frame 9 to the gripper frame have not as yet been determined. **HOME** positions have been defined for the MCS for the PUMAs. This position corresponds to all zero joint values, and is shown in Figures A.1 and A.2. This position, because of the alignment of two the wrist joint axes, is singular.

A.2 Software Joint Limits for the PUMAs

While the hardware joint limits describe the range of motion physically permitted, it is not possible to utilize this entire range. For example, path planners may require additional restrictions to provide safe motion. The following table lists the recommended joint limits for the testbed. These values are based on the hardware joint limits with consideration given to the link size and range, and a safety region (nominally 5 degrees, except it is 6 degrees for a joint able to reach its encoder limit).

frame number	name of associated axis	global label	local label	software range of associated joint
0	n/a	G_0	L_0, R_0	n/a
1	left cart linear	G_1	L_1	(−1.3716, 0.6096) m
2	left cart rotate	G_2	L_2	(−150, 150) degs
3	left cart tilt	G_3	L_3	(−45, 45) degs
4	left PUMA shoulder	G_4	L_4	(−251, 74) degs
5	left PUMA upper-arm	G_5	L_5	(−215, 34) degs
6	left PUMA fore-arm	G_6	L_6	(−55, 241) degs
7	left PUMA wrist	G_7	L_7	(−121, 144) degs
8	left PUMA flange tilt	G_8	L_8	(−95, 95) degs
9	left PUMA flange rotate	G_9	L_9	(−284, 284) degs
10	right cart linear	G_{10}	R_1	(−0.6096, 1.3716) m
11	right cart rotate	G_{11}	R_2	(−150, 150) degs
12	right cart tilt	G_{12}	R_3	(−45, 45) degs
13	right PUMA shoulder	G_{13}	R_4	(−248, 78) degs
14	right PUMA upper-arm	G_{14}	R_5	(−215, 37) degs
15	right PUMA fore-arm	G_{15}	R_6	(−55, 238) degs
16	right PUMA wrist	G_{16}	R_7	(−129, 148) degs
17	right PUMA flange tilt	G_{17}	R_8	(−95, 95) degs
18	right PUMA flange rotate	G_{18}	R_9	(−284, 284) degs

The information in the joint limit tables should be used in the following manner:

- Trajectory generators, controllers, path planners, etc, should use the software joint limits for specifying the manipulator motion ranges.

- The low level protection code in the robo channel drive ould use hardware joint limits.

This usage permits a consistent specification of manipulator motions and provides two levels of protection against reaching the joint limits: the channel drivers will disable a joint only when the physical limit is threatened; higher level software will never request a joint move to these limits. It is expected that the channel drivers will also include a 3 degree limit on these values to ensure safety.

A.3 Po Names

In g eral, thr ose variables, each with two values, are needed to select the desired solution fr n the eight possible solutions of a PUMA inverse kinematic problem. Selection of the pose definitions was a trade-off between easy visualization of the pose by human analogy and ease of computation. The labels to be used for the PUMA poses and their definitions are summarized in the table below—joint variables referenced are those for the left PUMA.

pose name	joint range
right	$f_{\text{thres}}(q_4, q_5, q_6) < 0$
left	$f_{\text{thres}}(q_4, q_5, q_6) > 0$
flex	$q_6 < 92.6864^o$
noflex	$q_6 > 92.6864^o$
flip	$q_8 < 0^o$
noflip	$q_8 > 0^o$

Standing on the PUM base and looking straight at its wrist, the shoulder link of the PUMA will be o her the left or right side of your body, corresponding to the **left** or **right** config on, respectively. It is important to only consider the

location of the PUMA's wrist coordinate frame, and *not* the flange of its last joint or any tool that might be attached to the wrist. The computation involves joints 4, 5, and 6. The PUMA is in the **left** configuration when it is in the **HOME** position, (as shown in Figures A.1 and A.2). With the other PUMA joints remaining stationary, this configuration variable changes when either q_5 or q_6 move to cause the wrist to pass over the "head" of the PUMA. When the wrist is directly above the PUMA, the robot is neither in the **left** or **right** configuration.

Consider, now, that the PUMA is in the **left** configuration. When the value of the elbow angle, i.e., q_6, is 92.6864^o, the fore-arm and upper-arm align to make the PUMA stretched. In this position, the PUMA is neither in the **flex** or **noflex** configuration. As the fore-arm is drawn towards the top of the upper-arm by changing the elbow angle, i.e., the motion achievable with the *unbroken* human arm, the PUMA enters the **flex** configuration (so named since this motion mimics a human flexing his/her arm). Conversely, the PUMA is in the **noflex** configuration if the elbow angle is changed in the other direction. This analogy is *reversed* when the PUMA is in the **right** configuration. In this case, the orientation of the fore-arm and upper-arm unlikely for humans is the **flex** configuration.

The last pose label deals with the PUMA's wrist orientation. Because of the construction of the PUMA wrist, there is no human analogy to this redundancy. A piece of tape will be placed on the PUMA's wrist near the axis of q_8. When q_8 is such that the flange of the PUMA's wrist overlaps the tape, then the PUMA will be in the **no flip** configuration.

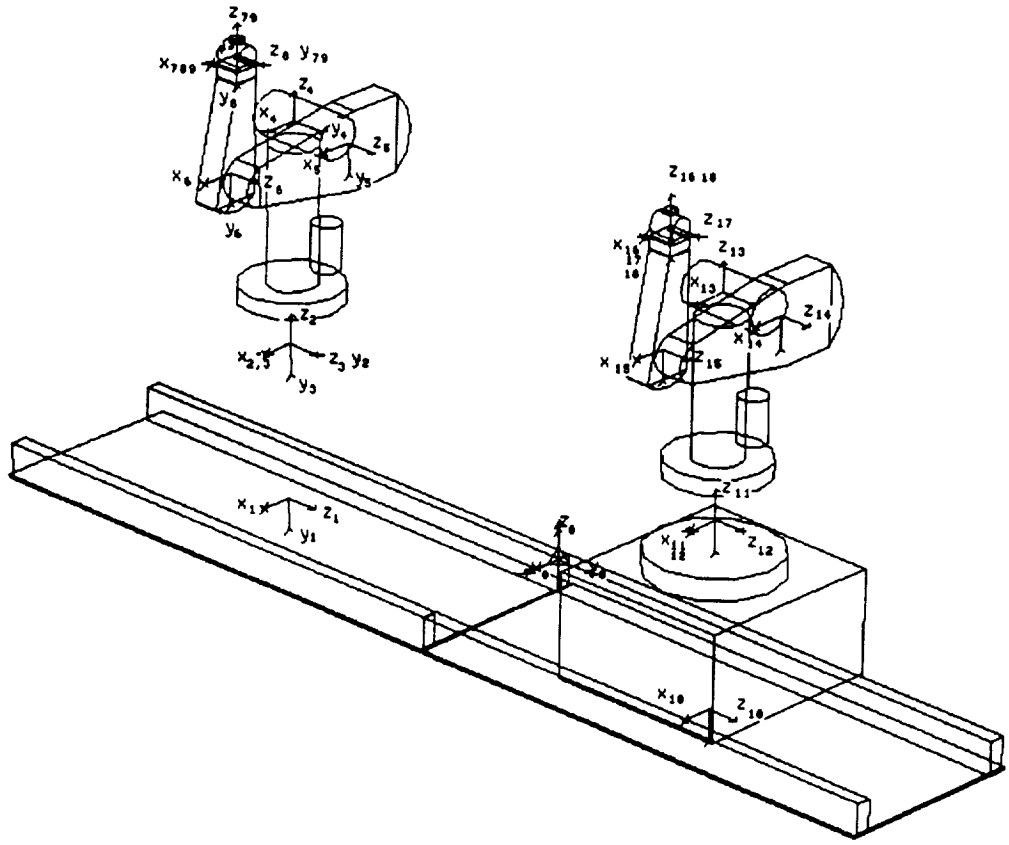


Figure A.1: Coordinate Frame Assignments

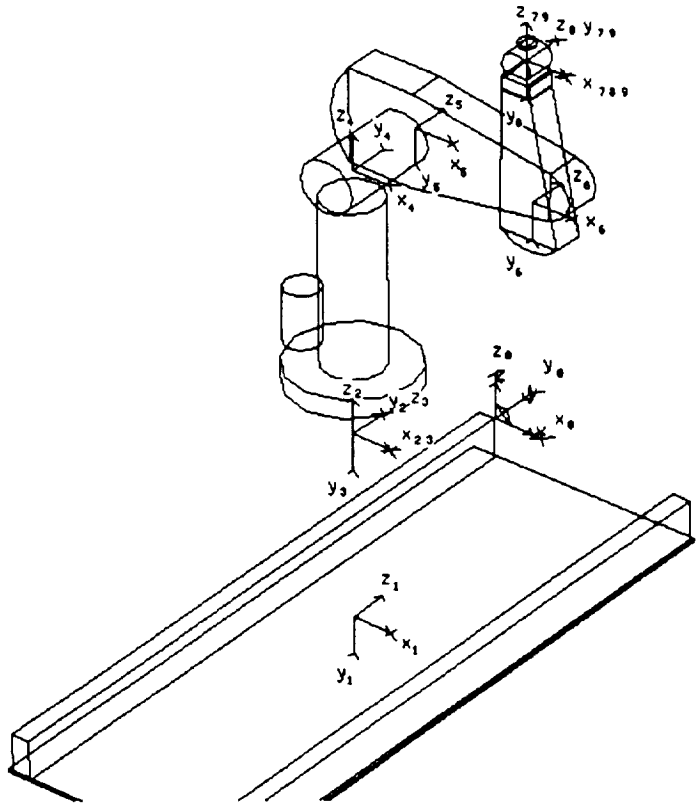


Figure A.2: Left Half Coordinate Frame Assignments

APPENDIX B
Data for Examples Presented in Thesis

This Appendix provides the task and obstacle descriptions for the examples presented earlier in Chapter 6. The task description has the form of start and goal joint angles. Revolute joints are measured in degrees and prismatic joints are measured in mm. The obstacle descriptions have the following format with dimensions in mm:

/ obstacle no. / number of points / polytope radius / origin of reference frame (X,Y,Z) / (X,Y,Z) of first point / $\cdots$ / (X,Y,Z) of last point /

Solution times and other solution parameters were presented earlier in Chapter 6.

The point coordinates are in local coordinates. The obstacle reference frames have the same orientation as the world reference frame. The world reference frame and the robot joint angle definitions are defined in [101].

B.1 Data for Examples 1 and 2

Examples 1 and 2 are identical except that the lower three joints remain fixed for Example 1 but are allowed to move for Example 2. The start and goal joint angles for these examples are:

$\Theta_0 = (0, 0, 0, 16.03, -148.79, -8.35, 0.00, -22.86, 106.03)$ and

$\Theta_f = (0, 0, 0, -184.37, -158.90, 20.22, 0.00, -41.32, 265.63)$, respectively. The eight obstacles are as follows:

/ 1 / 2 / 40 / (1000,-100,800) / (200,0,0) / (-200,0,0) /

/ 2 / 2 / 40 / (1000, -100, 800) / (200, 0, 0) / (0, 0, 346) /

/ 3 / 2 / 40 / (1000, -100, 800) / (-200, 0, 0) / (0, 0, 346) /

/ 4 / 8 / 0 / (500, 600, 1250) / (100, 100, 0) / (-100, 100, 0) / (-100, -100, 0) /
(100, -100, 0) / (100, 100, 200) / (-100, 100, 200) / (-100, -100, 200) /
(100, -100, 200) /

/ 5 / 8 / 0 / (-200, 200, 1000) / (-100, -100, 0) / (-100, 100, 0) / (100, 100, 0) /
(100, -100, 0) / (-100, -100, 100) / (-100, 100, 100) / (100, 100, 100) /
(100, -100, 100) /

/ 6 / 8 / 0 / (-350, 200, 800) / (-50, -100, 0) / (-50, 100, 0) / (50, 100, 0) /
(50, -100, 0) / (-50, -100, 300) / (-50, 100, 300) / (50, 100, 300) / (50, -100, 300) /

/ 7 / 8 / 0 / (-50, 200, 800) / (-50, -100, 0) / (-50, 100, 0) / (50, 100, 0) /
(50, -100, 0) / (-50, -100, 300) / (-50, 100, 300) / (50, 100, 300) / (50, -100, 300) /

/ 8 / 8 / 0 / (-200, 200, 800) / (-100, -100, 0) / (-100, 100, 0) / (100, 100, 0) /
(100, -100, 0) / (-100, -100, 100) / (-100, 100, 100) / (100, 100, 100) /
(100, -100, 100) /

B.2 Data for Example 3

For this example, platform 1 is fixed at $(-900, -90, 0)$ and platform 2 is fixed at $(900, -90, 0)$. The start robot 1 and 2 joint angles for this example are:
$\mathbf{\Theta 1_0} = (64.40, -178.80, 121.20, 0.00, 57.60, 115.60)$ and
$\mathbf{\Theta 2_0} = (-226.97, -185.55, 136.58, 0.00, 48.98, 226.97)$, respectively. The goal joint angles for this example are:
$\mathbf{\Theta 1_f} = (-42.00, -169.46, 115.96, 0.00, 53.40, 222.00)$ and
$\mathbf{\Theta 2_f} = (-111.92, -176.85, 133.07, -0.14, 43.75, 112.02)$, respectively. The six obstacles are as follows:

/ 1 / 8 / 0 / (400, 0, 1750) / (275, 150, 0) / (275, -150, 0) / (-275, -150, 0) /
(-275, 150, 0) / (275, 150, 100) / (275, -150, 100) / (-275, -150, 100) / (-275, 150, 100) /

/ 2 / 8 / 0 / (500, 0, 2000) / (375, 150, 0) / (375, -150, 0) / (-375, -150, 0) /
(-375, 150, 0) / (375, 150, 100) / (375, -150, 100) / (-375, -150, 100) / (-375, 150, 100) /

/ 3 / 8 / 0 / (180, 0, 2100) / (50, 150, 0) / (50, -150, 0) / (-50, -150, 0) / (-50, 150, 0) /
(50, 150, 300) / (50, -150, 300) / (-50, -150, 300) / (-50, 150, 300) /

/ 4 / 8 / 0 / (180, 0, 1450) / (50, 150, 0) / (50, -150, 0) / (-50, -150, 0) / (-50, 150, 0) /
(50, 150, 300) / (50, -150, 300) / (-50, -150, 300) / (-50, 150, 300) /

/ 5 / 8 / 0 / (825, 0, 1850) / (50, 150, 0) / (50, -150, 0) / (-50, -150, 0) / (-50, 150, 0) /
(50, 150, 150) / (50, -150, 150) / (-50, -150, 150) / (-50, 150, 150) /

/ 6 / 8 / 0 / (-175, 0, 1750) / (50, 150, 0) / (50, -150, 0) / (-50, -150, 0) / (-50, 150, 0) /
(50, 150, 250) / (50, -150, 250) / (-50, -150, 250) / (-50, 150, 250) /

B.3 Data for Example 4

The start robot 1 and 2 joint angles for this example are:
$\mathbf{\Theta 1_0} = (-1300.00, 0.00, -40.00, -5.00, -110.70, 19.20, 4.30, -48.80, 83.40)$ and
$\mathbf{\Theta 2_0} = (-500.00, 0.00, -40.00, -184.92, -72.57, 170.91, 4.76, 41.97, 82.76)$, respectively. The goal joint angles for this example are:
$\mathbf{\Theta 1_f} = (500.00, 0.00, 40.00, -149.81, -163.61, 46.29, 23.80, 72.91, 242.14)$ and
$\mathbf{\Theta 2_f} = (1300.00, 0.00, 40.00, -171.09, -157.32, 33.29, 7.09, 74.44, 82.36)$, respectively.
The eight obstacles are as follows:

/ 1 / 8 / 0 / (-750, -850, 700) / (-50, -100, 0) / (-50, 100, 0) / (50, 100, 0) /
(50, -100, 0) / (-50, -100, 300) / (-50, 100, 300) / (50, 100, 300) / (50, -100, 300) /

/ 2 / 8 / 0 / (-1000, -850, 700) / (-50, -100, 0) / (-50, 100, 0) / (50, 100, 0) /
(50, -100, 0) / (-50, -100, 300) / (-50, 100, 300) / (50, 100, 300) / (50, -100, 300) /

/ 3 / 8 / 0 / (-875 -350, 700) / (-100, -50, 0) / (-100, 50, 0) / (100, 50, 0) /
(100, -50, 0) / (-100, -50, 300) / (-100, 50, 300) / (100, 50, 300) / (100, -50, 300) /

/ 4 / 8 / 0 / (-875, -1350, 700) / (-100, -50, 0) / (-100, 50, 0) / (100, 50, 0) /
(100, -50, 0) / (-100, -50, 300) / (-100, 50, 300) / (100, 50, 300) / (100, -50, 300) /

/ 5 / 8 / 0 / (-875, -850, 600) / (-175, -550, 0) / (-175, 550, 0) / (175, 550, 0) /
(175, -550, 0) / (-175, -550, 100) / (-175, 550, 100) / (175, 550, 100) /
(175, -550, 100) /

/ 6 / 2 / 100 / (1450, 1000, 0) / (0, 0, 0) / (0, 0, 650) /

/ 7 / 8 / 0 / (1550, 100, 750) / (-200, -100, 0) / (-200, 100, 0) / (200, 100, 0) /
(200, -100, 0) / (-200, -100, 100) / (-200, 100, 100) / (200, 100, 100) /
(200, -100, 100) /

/ 8 / 8/ 0 / (1800, 1000, 550) / (-100, -50, 0) / (-100, 50, 0) / (100, 50, 0) /
(100, -50, 0) / (-100, -50, 300) / (-100, 50, 300) / (100, 50, 300) / (100, -50, 300) /