PATH PLANNING FOR ASSEMBLY OF STRUT-BASED STRUCTURES

NAGW-1333

by

Rolf Münger

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering
Troy, New York 12180-3590

May 1991

CIRSSE REPORT #91

CONTENTS

LIST OF FIGURES iv

ACKNOWLEDGEMENT vi

ABSTRACT vii

1. INTRODUCTION 1
   1.1 Nomenclature 3

2. LITERATURE REVIEW 4

3. GEOMETRIC MODELING 7

4. USING A ROBOT TO FOLLOW A PATH 14
   4.1 Modeling the robot's geometry 14
   4.2 Potential field method in the robot's joint space 14
      4.2.1 Attraction Field 15
      4.2.2 Obstacle Avoidance 20
      4.2.3 Derivation of the manipulator Jacobian 24
      4.2.4 Joint Range 25
      4.2.5 Imposing restrictions in cartesian space 27
      4.2.6 A method for solving general linear systems 30

5. COMBINING GLOBAL AND LOCAL PATH PLANNING METHODS . . 35
   5.1 Potential field methods and their problems 35
   5.2 Different ways of performing a given task 35
   5.3 Splitting into subtasks using graph search 37
      5.3.1 Application of the A* algorithm 38
      5.3.2 Selecting subgoals for the truss structure problem 40
      5.3.3 Extraction of tetrahedra 41

6. SOFTWARE DOCUMENTATION 44
   6.1 Concepts 44
      6.1.1 Global variables 44
      6.1.2 Data types 45

6.1.3 Module hierarchy 46
6.1.4 Make 47
6.2 Documentation of modules 47
6.2.1 The "global" module 49
6.2.2 The "spec" module 49
6.2.3 The "lst" module 49
6.2.4 The "stack" module 50
6.2.5 The "vector" module 51
6.2.6 The "alg" module 51
6.2.7 The "graph" module 52
6.2.8 The "parser" module 55
6.2.9 The "model" module 56
6.2.10 The "graphics" module 56
6.2.11 The "env" module 57
6.2.12 The "robot" module 57
6.2.13 The "lpath" module 58
6.2.14 The "gpath" module 58
6.2.15 The "main" module 59
6.3 Interface to CIRSSE 59
6.3.1 CIRSSE interface procedures 59
6.3.2 Input file 62
7. RESULTS 68
8. DISCUSSION AND CONCLUSIONS 76
8.1 Computational Complexity 76
8.2 Generalization of geometric model 78
8.3 Experiences with the potential field algorithm 78
8.4 Configurations and singularities 80
LITERATURE CITED 82
APPENDICES 85
A. Simulation input file 85
B. Listings of header files 86

LIST OF FIGURES

Figure 1.1	An example structure	2
Figure 3.1	Basic geometric model	8
Figure 3.2	Shortest distance between two swept sphere cylinders	9
Figure 3.3	Shortest distance between two line segments	10
Figure 3.4	Distance computation	11
Figure 4.1	Overview	17
Figure 4.2	Sense of rotation	19
Figure 4.3	Basic collision avoidance procedure for a revolute joint	21
Figure 4.4	Basic collision avoidance procedure for a prismatic joint	23
Figure 4.5	Numbering convention for the joints and links of the robot	24
Figure 4.6	$\delta q_r(q)$	25
Figure 4.7	Domain mapping for joint range function	26
Figure 4.8	$r(u)$	27
Figure 5.1	Two ways of inserting a strut	36
Figure 5.2	Choosing the correct sense of rotation	37
Figure 5.3	Splitting a task into two subtasks	38
Figure 5.4	Intermediate steps placed around a tetrahedron	40
Figure 6.1	Module hierarchy	48
Figure 6.2	List structure	50
Figure 6.3	Data structures in the graph module	54
Figure 6.4	Conventions for specifying a strut in a structure	63
Figure 7.1	A robot arm of the CIRSSE testbed	69
Figure 7.2	A simple motion task	70

Figure 7.3 Completing a tetrahedron 72

Figure 7.4 An example of a large angle of rotation 73

Figure 7.5 Initial state of obstacle avoidance demonstration 74

Figure 7.6 Obstacle avoidance demonstration 75

Figure 8.1 Step size selection 79

ACKNOWLEDGEMENT

I would like to express my thanks to Professor Arthur C. Sanderson, my thesis advisor, for his guidance, encouragement and patience.

Special thanks go to Dr. Steve Murphy and Michael Eppinger for their help in robotics issues and to Jonathan Weaver for reviewing the manuscript.

ABSTRACT

A path planning method with collision avoidance for a general single chain nonredundant or redundant robot is proposed. Joint range boundary overruns are also avoided. The result is a sequence of joint vectors which are passed to a trajectory planner.

A potential field algorithm in joint space computes incremental joint vectors $\mathbf{\Delta q} = \mathbf{\Delta q_a} + \mathbf{\Delta q_c} + \mathbf{\Delta q_r}$. Adding $\mathbf{\Delta q}$ to the robot's current joint vector leads to the next step in the path.

$\mathbf{\Delta q_a}$ is obtained by computing the minimum norm solution of the underdetermined linear system $\mathbf{J \Delta q_a} = \mathbf{x_a}$ where $\mathbf{x_a}$ is a translational and rotational force vector that attracts the robot to its goal position and orientation. $\mathbf{J}$ is the manipulator Jacobian.

$\mathbf{\Delta q_c}$ is a collision avoidance term encompassing collisions between the robot (links and payload) and obstacles in the environment as well as collisions among links and payload of the robot themselves. It is obtained in joint space directly.

$\mathbf{\Delta q_r}$ is a function of the current joint vector and avoids joint range overruns.

A higher level discrete search over candidate safe positions is used to provide alternatives in case the potential field algorithm encounters a local minimum and thus fails to reach the goal. The best first search algorithm A* is used for graph search. Symmetry properties of the payload and equivalent rotations are exploited to further enlarge the number of alternatives passed to the potential field algorithm.

CHAPTER 1
INTRODUCTION

In earlier applications of robotic systems and in most present systems, the task of planning a feasible path is done by a human. The most basic way of entering the path is typing a sequence of points in cartesian or even in joint space and having the system perform a linear interpolation between the points. If the robot is operating in a constant environment, this approach is often sufficient. Some variation in the environment can be taken into account by prestoring a set of paths and selecting the appropriate one depending on the current state of the environment.

In some applications, as for example spray painting a car part, a complex continuous motion must be accomplished. This kind of motion would be very tedious to enter by typing long lists of cartesian or joint coordinates. In this case, a method known as "teach-in" is used, where a competent person is guiding the robot's end effector, while the system is storing the motion. Later on, this motion can be reproduced repeatedly and accurately.

The drawback of all these methods is their inability to work in a largely unknown environment. If an unexpected obstacle is entering the workspace, the robot will probably collide with it.

In many applications it is desirable to have more intelligent robotic systems capable of working more autonomously. The work herein has been done under CIRSSE (Center for Intelligent Robotic Systems for Space Exploration). The goal of CIRSSE is the development of planning, sensing, and control methods that will eventually allow mostly autonomous assembly of parts of a space station in outer space. Autonomy is particularly important in outer space, since risk and costs of assembly by humans are extremely high. Another application is autonomous exploration of other planets, where teleoperation is impossible due to large communication delays.

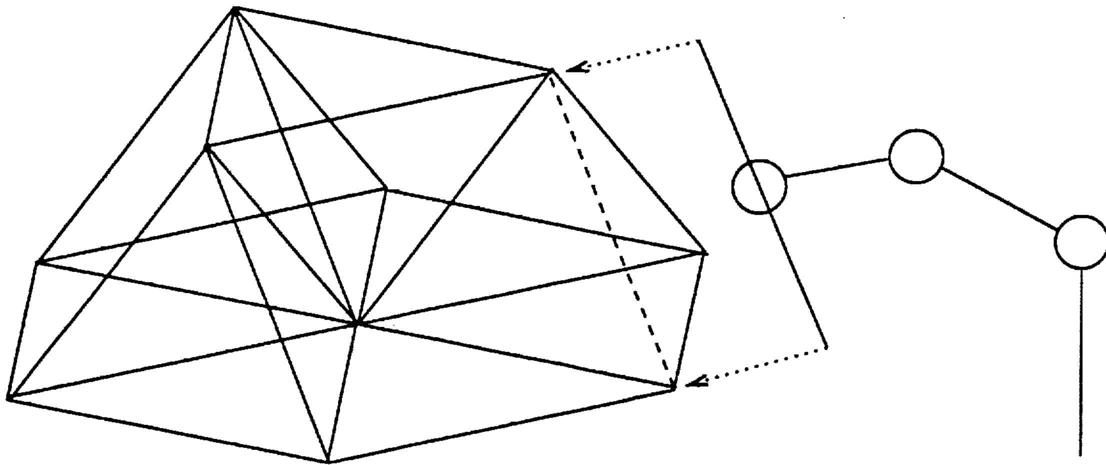

Figure 1.1: An example structure

An example of a structure being assembled in space is the truss structure shown in Figure 1.1. The basic cell in this structure is the tetrahedron. Tetrahedra have two useful properties: They are mechanically stable and their six edges all have the same length, so only one type of strut is needed. Using them as a unit cell, one can build arbitrarily large, stable structures.

The first step in doing this automatically is determining an assembly sequence such that subassemblies are stable whenever this is possible [6]. Once the sequence has been determined, a path planner is needed to find a collision free path to move the struts from their storage location to their goal position in the structure. This project is proposing a solution to the path planning problem. It is a free motion planner, so it does not cover the grasping and mating operations and the associated fine motion planning which requires an online visual feedback and force feedback. Another important part of the system is a high level supervisor which decides about new strategies if a part of the plan turns out to be unfeasible or if other unexpected events require a change of strategy.

The path planner is first requesting information about the current state of the environment. It needs to know the positions of all struts currently in the environment and the current joint vector of the robot. After this initialization it accepts motion commands telling it to move the end effector to a certain position and orientation in cartesian space. It must also be informed of grasping and ungrasping operations, since the changes in the environment have an influence on the path planning process. The path planner provides a sequence of joint vectors which is passed to the trajectory planner. In this stage the velocity and acceleration information is added and the result is passed to the low level robot controller.

This text is organized as follows: Chapter 2 gives an overview of the different path planning strategies and algorithms that have been developed in the past. In chapter 3, the geometric model for object representation used in this work is introduced and a distance computation algorithm is presented. Chapter 4 covers the description of the robot used to execute the tasks and presents the local path planning algorithm based on potential fields. The global path planning based on graph search is covered in chapter 5. The documentation of the path planner's implementation is presented in chapter 6, and chapter 7 shows the results that have been obtained. A discussion of the results and conclusions can be found in chapter 8. An input file of a simulated task in included in appendix A and listings of the program's header files can be found in appendix B.

1.1 Nomenclature

Throughout this text, scalars are printed in normal typeface (x or X), vectors in lowercase bold face ($\mathbf{x}$) and matrices in uppercase bold face letters ($\mathbf{X}$).

CHAPTER 2
LITERATURE REVIEW

Currently known path planning algorithms fall into two categories: global and local.

The global methods typically consider the environment as a whole when planning a path. They generally have the advantage of finding a path if one exists, but they are computationally expensive if the environment is complex. There are three subgroups of global path planning algorithms:

- Cell decomposition methods
- Search algorithms on a visibility graph
- Optimization using calculus of variation

Cell decomposition methods can be further divided into exact and approximate methods whereas exact methods decompose free space into cells complex enough to exactly represent the free space. Approximate methods however split free space into very simple units, for instance cubes, that cannot represent free space exactly. There will be a number of cubes that contain both free and occupied space. These cells are further subdivided into smaller cells, until the desired accuracy is obtained. After the cell model is established, graph search techniques are used to generate the shortest path that includes only cells representing free space [20], [21].

Instead of splitting the space into cells, one can also find a sufficiently large set of feasible intermediate goals, usually located near corners of obstacles, and establish a graph based on visibility between pairs of these goals. Both of these methods rely on an efficient graph search algorithm. A suitable algorithm is A*, a best first search algorithm that considers a lower bound estimate of the remaining distance in order to expand the most promising nodes first [4], [18].

A different approach that doesn't involve graph search is the optimization of a cost function that typically includes path length and clearance from obstacles. This approach uses methods from calculus of variation [19].

The oldest local path planning method is the "hypothesize and test" method. The robot proceeds a step in the desired direction and checks the feasibility of the path that leads to the new position. If the path is not feasible, another direction is chosen according to some heuristic. This approach has not been very successful.

A more recent method is based on potential fields [7], [8], [9], [17]. The gradient of a potential field is guiding the robot towards the goal and keeps it at a safe distance from obstacles. This potential field has very large values near obstacles in order to produce a repulsive force between the robot and the obstacles. The potential assumes its lowest value at the goal position in order to attract the robot to the goal. The drawback of all potential field methods is the fact that they do not guarantee finding a collision free path even if one exists. If the potential field function is not carefully chosen, it is likely to contain local minima, in which the path planning algorithm can get stuck. An attempt to solve this problem is made in [10] by adding vortex fields. However, even if a path can be found, it is not guaranteed to be optimal.

Variations of the potential field methods include information about the dynamics of the robot and its payload. These methods have the advantage that they generally produce paths which can be executed at a comparatively high speed [15], [16].

If the robot is not assumed to be a point or a sphere, then its shape must be included in the path planning process. A particular path might be feasible depending on the orientation of the robot as a function of its position. In order to account for the robot's orientation, the path planning problem can be solved in configuration space [22], [23], [24], [25]. The rotational degrees of freedom of the robot are added

as additional dimensions to the cartesian space. If we consider a three dimensional space and a robot with another three degrees of freedom for rotation, the associated configuration space is six dimensional. In configuration space, the robot is shrunk to a point and the obstacles are enlarged according to the robot's orientation. This reduces the problem to planning a path for a point, however in a higher dimensional space.

CHAPTER 3
GEOMETRIC MODELING

Every path planning algorithm needs a geometric description of the objects it works with. If a very simple model is used to represent the objects, it is in most cases necessary to choose the model's volume considerably larger than the real object's volume, since the object's volume must be completely enclosed in the volume of its model. This restricts the path planning process and might lead to a failure even though a path exists. On the other hand, distance computations on simple models are typically performed very efficiently. More involved models allow for a better approximation to the real shape of the objects, but they are computationally more expensive.

In this work, cylinders with spherical caps have been chosen as the basic unit for geometric modeling. An example is shown in Figure 3.1. It can also be viewed as the volume that is swept by a sphere with radius r moving between the two points $\mathbf{p_1}$ and $\mathbf{p_2}$ on a straight line. This volume can be described as a set S in the following way:

$$\begin{aligned} \mathbf{s} &= \mathbf{p_2} - \mathbf{p_1} \\ S(\mathbf{p_1}, \mathbf{p_2}, r) &= \{\mathbf{p} \mid \mathbf{p} = \mathbf{p_1} + t\mathbf{s} + \mathbf{q} \quad \text{with } t \in [0 \ldots 1],\ |\mathbf{q}| \leq r\} \end{aligned} \tag{3.1}$$

This model is particularly well suited for the objects of interest in strut-based structure assembly, namely struts and robot links, but any kind of object can be approximated by using a union of several basic cylinders.

This chapter presents the distance computation for this particular model in detail, but every other model as for instance a polyhedral model can be used, as long as the two points of closest distance between two objects can be computed.

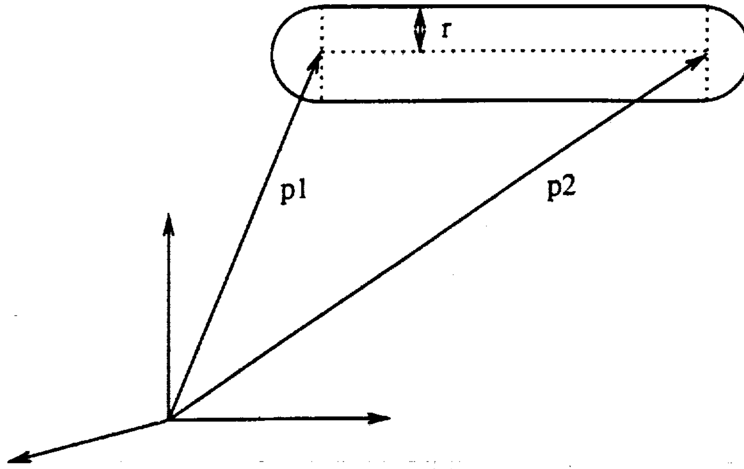

Figure 3.1: Basic geometric model

A generalization of the swept sphere model discussed here is a swept sphere model in which the sphere radius is a linear function of the sphere's position between the endpoints: The radius is $r = (1-t)r_1 + tr_2$ where r_1 is the radius of the sphere at point $\mathbf{p}_1$ and r_2 the radius at point $\mathbf{p}_2$. This model allows for more accurate representation of conic objects and also provides a very efficient distance computation method [26].

In the following the distance computation algorithm for the cylindrical model is presented (Figure 3.2). In order to find the distance d between two objects, we find the distance d_s between the two line segments in the center of the cylinders and subtract the radii of the cylinders r_p and r_q.

$$d = d_s - r_p - r_q \tag{3.2}$$

The line segment in the center of a cylinder $S(\mathbf{p}_1, \mathbf{p}_2, r)$ can be described as $S(\mathbf{p}_1, \mathbf{p}_2, 0)$.

The remainder of this chapter discusses the computation of d_s, the distance between the line segments of two cylinders.

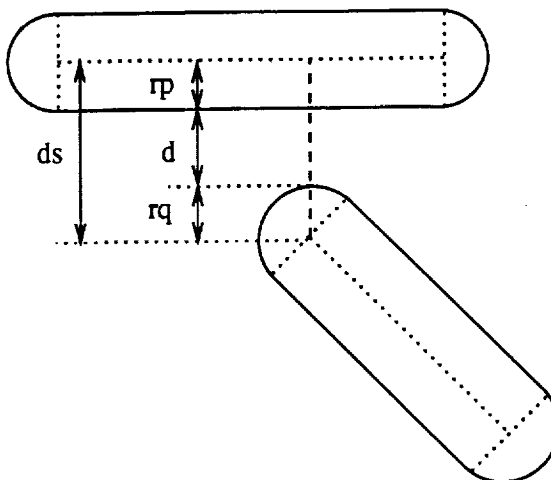

Figure 3.2: Shortest distance between two swept sphere cylinders

The distance between two line segments $S_p(\mathbf{p_1}, \mathbf{p_2}, 0)$ and $S_q(\mathbf{q_1}, \mathbf{q_2}, 0)$ can be obtained by first finding the distance between the two unbounded lines defined by the endpoints of the line segments. The two unbounded lines can be described as follows (see also Figure 3.3):

$$\mathbf{s_p} = \mathbf{p_2} - \mathbf{p_1}$$

$$\mathbf{s_q} = \mathbf{q_2} - \mathbf{q_1}$$

$$\mathbf{p} = \mathbf{p_1} + t_p \mathbf{s_p} \tag{3.3}$$

$$\mathbf{q} = \mathbf{q_1} + t_q \mathbf{s_q} \tag{3.4}$$

The parameters t_p and t_q are real numbers. If the lines are not parallel, then the two points of minimal distance

$$\mathbf{p_{min}} = \mathbf{p_1} + t_{pmin} \mathbf{s_p} \tag{3.5}$$

$$\mathbf{q_{min}} = \mathbf{q_1} + t_{qmin} \mathbf{s_q} \tag{3.6}$$

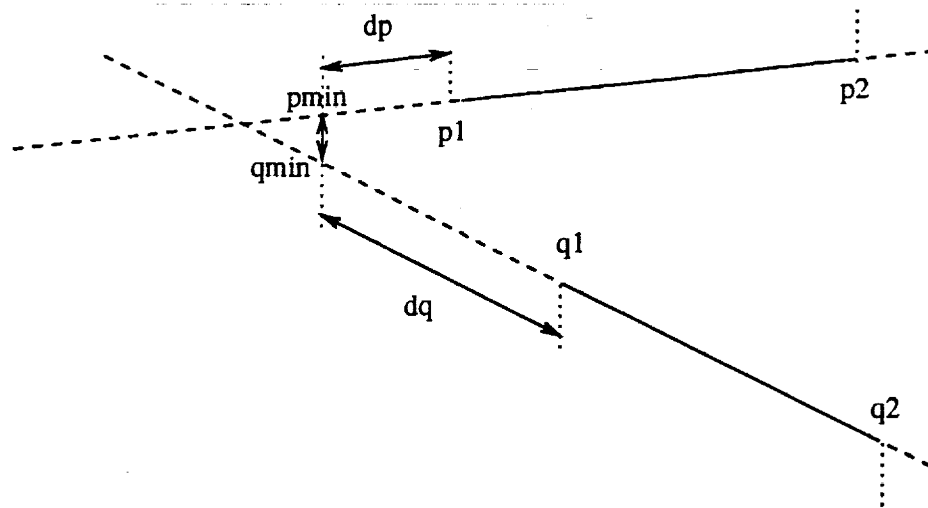

Figure 3.3: Shortest distance between two line segments

can be obtained by solving the following linear system for t_{pmin} and t_{qmin}:

$$\mathbf{p_1} + t_{pmin}\mathbf{s_p} + x(\mathbf{s_p} \times \mathbf{s_q}) = \mathbf{q_1} + t_{qmin}\mathbf{s_q} \tag{3.7}$$

If these points are also on the line segments ($t_{pmin} \in [0 \dots 1]$ and $t_{qmin} \in [0 \dots 1]$), then the distance $|\mathbf{p_{min}} - \mathbf{q_{min}}|$ is the distance between the line segments and the desired distance d between the cylinders is:

$$d = |\mathbf{p_{min}} - \mathbf{q_{min}}| - r_p - r_q \tag{3.8}$$

If t_p or t_q or both are not element of $[0 \dots 1]$ as it is the case in Figure 3.3, then we define an auxiliary distance d_p as follows:

$$d_p = \begin{cases} |\mathbf{p_1} - \mathbf{p_{min}}| & \text{for } t_p < 0 \\ |\mathbf{p_2} - \mathbf{p_{min}}| & \text{for } t_p > 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

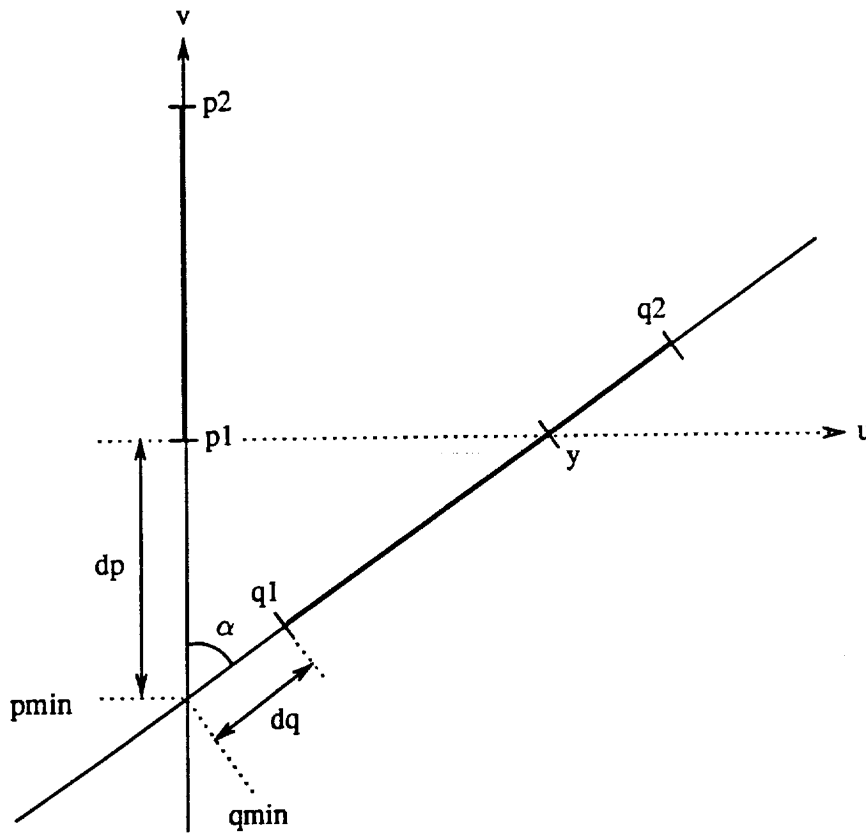

Figure 3.4: Distance computation

An equivalent distance d_q for the other line is also defined. The values d_p and d_q represent the distances between the points $\mathbf{p_{min}}$ and $\mathbf{q_{min}}$ and their line segments $S_p(\mathbf{p_1}, \mathbf{p_2}, 0)$ and $S_q(\mathbf{q_1}, \mathbf{q_2}, 0)$ respectively. Let us also define the two points $\mathbf{z_p}$ and $\mathbf{z_q}$ of minimal distance on the two line segments:

$$\min |\mathbf{z_p} - \mathbf{z_q}| \qquad \text{subject to} \quad \mathbf{z_p} \in S_p(\mathbf{p_1}, \mathbf{p_2}, 0) \text{ and } \mathbf{z_q} \in S_q(\mathbf{q_1}, \mathbf{q_2}, 0) \tag{3.10}$$

For the remainder of the discussion we can assume without loss of generality that $d_p \geq d_q$, $t_p \leq 0$ and $t_q \leq 0$. Consider Figure 3.4 which shows a projection of the two line segments on a plane perpendicular to the line defined by $\mathbf{p_{min}}$ and $\mathbf{q_{min}}$. Let us also assume that angle $\alpha \leq 90^o$.

Under these assumptions we can state that $\mathbf{z_p} = \mathbf{p_1}$.

Proof: Consider the orthogonal coordinate system (u, v) in Figure 3.4. Let us first state that the v coordinate of $\mathbf{q_1}$ is always negative, since $d_p \geq d_q$. We can also state that if $\mathbf{z_p} \neq \mathbf{p_1}$ then the v coordinate of $\mathbf{z_q}$ must be positive and equal to the v coordinate of $\mathbf{z_p}$ (not considering the case that $\mathbf{z_p} = \mathbf{p_2}$). If the v coordinate of $\mathbf{q_2}$ is negative, then a $\mathbf{z_q}$ with a positive v coordinate does not exist. Otherwise $\mathbf{z_q}$ must be an element of $S' = S(\mathbf{y}, \mathbf{q_2}, 0) \neg \mathbf{y}$. $\mathbf{y}$ is the element of $S(\mathbf{q_1}, \mathbf{q_2}, 0)$ with a zero v coordinate. But since S' is an open set, there is a point $\mathbf{z_q}' \in S'$ in every neighborhood of $\mathbf{z_q}$ such that $|\mathbf{z_p} - \mathbf{z_q}'| < |\mathbf{z_p} - \mathbf{z_q}|$, so $\mathbf{z_p} \neq \mathbf{p1}$ does not exist. □

This reduces the problem to finding the minimum distance between point $\mathbf{p_1}$ and line segment $S(\mathbf{q_1}, \mathbf{q_2}, 0)$. The point $\mathbf{q_{min}}$ on the unbounded line defined in equation 3.4 closest to $\mathbf{p_1}$ is defined by:

$$\begin{aligned} \mathbf{q_{min}} &= \mathbf{q_1} + t_{qmin}\mathbf{s_q} \\ t_{qmin} &= \frac{\mathbf{s_q} \cdot (\mathbf{p_1} - \mathbf{q_1})}{\mathbf{s_q} \cdot \mathbf{s_q}} \end{aligned} \tag{3.11}$$

From this we can find $\mathbf{z_q}$ as follows:

$$\mathbf{z_q} = \begin{cases} \mathbf{q_1} & \text{for } t_{qmin} < 0 \\ \mathbf{q_2} & \text{for } t_{qmin} > 1 \\ \mathbf{q_{min}} & \text{otherwise} \end{cases} \tag{3.12}$$

Then $d_s = |\mathbf{z_p} - \mathbf{z_q}|$ and we can find the distance between the cylinders using equation 3.2. This concludes the discussion of the distance computation for nonparallel line segments.

If the line segments are parallel, then we compute the four distances

$$\begin{aligned} d_1 &: \text{ distance between } \mathbf{p_1} \text{ and} S_q(\mathbf{q_1}, \mathbf{q_2}, 0) \\ d_2 &: \text{ distance between } \mathbf{p_2} \text{ and} S_q(\mathbf{q_1}, \mathbf{q_2}, 0) \\ d_3 &: \text{ distance between } \mathbf{q_1} \text{ and} S_p(\mathbf{p_1}, \mathbf{p_2}, 0) \\ d_4 &: \text{ distance between } \mathbf{q_2} \text{ and} S_p(\mathbf{p_1}, \mathbf{p_2}, 0) \end{aligned}$$

using the algorithm described above. It turns out that at least two of the four distances will be equal to the minimum distance d_s, so it is sufficient to compute any three of the four distances and find the minimum, for instance $d_s = \min(d_1, d_2, d_3)$.

CHAPTER 4
USING A ROBOT TO FOLLOW A PATH

In real applications, objects are moved from their start to their goal positions by a robot, so the robot's geometry must be taken into consideration. The robot imposes constraints on the path in that the path must entirely lie in the robot's workspace. It also adds to the list of parts that may be involved in a collision. There may be collisions between obstacles and parts of the robot, and there may even be collisions among different parts of the robot itself. The limited range of the robot's joints further complicate things.

In this work, a single robot that holds a cylindrical object at its center is considered. The method presented could be used for all single chain robots with at least six degrees of freedom. All joints can be either revolute or prismatic.

4.1 Modeling the robot's geometry

In the process of path planning, we must consider collisions between different struts, collisions between a strut and a part of the robot and collisions among parts of the robot. In order to simplify distance computations, it makes sense to use the same model for robot links and other parts. The cylindrical model as shown in Figure 3.1 can also provide sufficient modeling accuracy for the robot links when performing free motion planning. Distance computation is done as described in chapter 3.

4.2 Potential field method in the robot's joint space

Potential field methods as outlined in chapter 2 work in cartesian space. All gradients of the potential fields are vectors in normal three dimensional space. They define the direction of motion of the robot's end effector, so the new location is

first given in cartesian space. This means that an inverse kinematics routine is required to find the new joint vector. Since this must be done at every step of the path, the algorithm becomes very computationally expensive, since in general the inverse kinematics problem can only be solved iteratively. In this work, however, an alternate strategy is taken whereby the robot's new location is first obtained in joint space, rather than in cartesian space. To find the location in cartesian space, a simple forward kinematics routine is required.

Thus the goal of the algorithm is to find a joint vector increment $\mathbf{\Delta q}$ that is added to the current joint vector $\mathbf{q}$ to obtain the next step of the path. $\mathbf{\Delta q}$ is the sum of several components each performing a different task:

$$\mathbf{\Delta q} = \mathbf{\Delta q_a} + \mathbf{\Delta q_r} \tag{4.1}$$

$\mathbf{\Delta q_a}$ is the joint vector increment that moves the payload closer to its goal position and orientation (attractive forces), while $\mathbf{\Delta q_r}$ is the sum of all joint vector increments that keep the robot from colliding or running out of joint range (repulsive forces).

$$\mathbf{\Delta q_r} = \mathbf{\Delta q_{rc}} + \mathbf{\Delta q_{rr}} \tag{4.2}$$

where $\mathbf{\Delta q_{rc}}$ is doing the collision avoidance and $\mathbf{\Delta q_{rr}}$ keeps the robot from running out of joint range.

Note that all fields are defined as gradient fields and not as potential fields. This often leads to a more natural definition and also saves the derivation of the field's gradient.

4.2.1 Attraction Field

The attractive field is the only field that must be defined in cartesian space. First the definition of the cartesian vector $\mathbf{x_a}$ is given, then the transformation into

joint space is shown.

$\mathbf{x_a}$ is a 6 by 1 vector. The first three elements ($\mathbf{x_{at}}$) represent the translational part (force) while the last three elements ($\mathbf{x_{ar}}$) represent the rotational part (torque):

$$\mathbf{x_a} = \begin{bmatrix} \mathbf{x_{at}} \\ \mathbf{x_{ar}} \end{bmatrix} \tag{4.3}$$

If $\mathbf{p_c}$ is the cartesian position of the center of the strut in the robot's gripper (current position) and $\mathbf{g_c}$ the cartesian position of the center of the strut in its goal position, then

$$\mathbf{x_{at}} = C_{at} \frac{\mathbf{g_c} - \mathbf{p_c}}{|\mathbf{g_c} - \mathbf{p_c}|} \tag{4.4}$$

C_{at} is the field constant of the translational attraction field. The rotational part $\mathbf{x_{ar}}$ is more difficult to obtain. Let $\mathbf{p_1}$ and $\mathbf{p_2}$ be the endpoints of the object in the robot's gripper and let $\mathbf{g_1}$ and $\mathbf{g_2}$ be the endpoints of the same object in its goal position. Furthermore let $\mathbf{a_p}$ be the approach vector of the end effector in its current orientation and let $\mathbf{a_g}$ be the approach vector of the end effector when it reaches its goal orientation. The approach vector represents the direction the end effector must move to properly approach an object before grasping it. Both $\mathbf{a_p}$ and $\mathbf{a_g}$ are unit vectors. We also need two unit vectors that point along the axes of the struts:

$$\mathbf{p_p} = \frac{\mathbf{p_2} - \mathbf{p_1}}{|\mathbf{p_2} - \mathbf{p_1}|}$$

$$\mathbf{p_g} = \frac{\mathbf{g_2} - \mathbf{g_1}}{|\mathbf{g_2} - \mathbf{g_1}|}$$

The approach vectors are perpendicular to the axes of the struts:

$$\mathbf{a_p} \perp \mathbf{p_p} \tag{4.5}$$

$$\mathbf{a_g} \perp \mathbf{p_g} \tag{4.6}$$

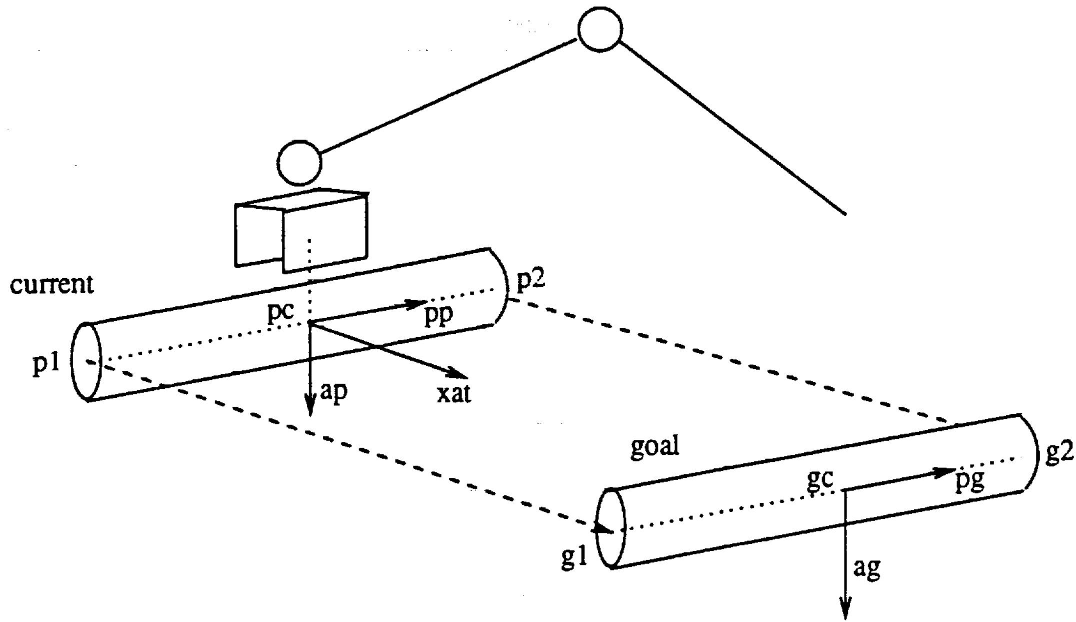

Figure 4.1: Overview

All these vectors are illustrated in Figure 4.1. The current orientation and the goal orientation can be represented by rotation matrices that give the respective orientations with respect to the world coordinate system:

$$\mathbf{T_p} = \left[\begin{array}{ccc} \mathbf{p_p} & (\mathbf{a_p} \times \mathbf{p_p}) & \mathbf{a_p} \end{array} \right] \tag{4.7}$$

$$\mathbf{T_g} = \left[\begin{array}{ccc} \mathbf{p_g} & (\mathbf{a_g} \times \mathbf{p_g}) & \mathbf{a_g} \end{array} \right] \tag{4.8}$$

Since rotation matrices are orthonormal, their inverse is equal to their transpose. Thus the rotation matrix that rotates the current gripper into the gripper in its goal orientation is defined as

$$\mathbf{T} = \mathbf{T_g}\mathbf{T_p^T} \tag{4.9}$$

This is the rotation we are interested in, but we need the axis of rotation and not the rotation matrix. In order to find the axis of rotation, an eigenvalue problem must be solved. Every rotation matrix has an eigenvalue of 1 and a corresponding eigenvector **a**. The axis of the rotation performed by the rotation matrix and eigenvector **a** have the same orientation. Vector **a** can be found by solving the following linear system:

$$(\mathbf{T} - \mathbf{I})\mathbf{a} = \mathbf{0} \tag{4.10}$$

This system can be solved by first setting the first component of **a** to 1. If solving for the other two components fails, then we set the second component to 1 and try to solve for the other two components. If this also fails, we set the third component to 1 and solve for the first two.

The vector **a** represents the axis of rotation, but it doesn't give any information about the sense of rotation, since $-\mathbf{a}$ is a valid eigenvector if **a** is an eigenvector. We can of course rotate both ways to reach the goal orientation but in most cases it is preferable to rotate in the sense that involves a smaller angle of rotation. For the method used to find the correct sense of rotation or the correct sign of **a** consider Figure 4.2.

It is assumed that **p** can be rotated into **g** by using **a** as the axis of rotation. The angle of rotation is less than or equal to 180° if the following condition holds:

$$\mathbf{g} \cdot (\mathbf{a} \times \mathbf{p}) \geq 0 \tag{4.11}$$

If the condition doesn't hold, we just change the sign of **a** to get the correct sense of rotation. Then the rotational part from equation 4.3 is:

$$\mathbf{x}_{\mathbf{ar}} = C_{ar}\mathbf{a} \tag{4.12}$$

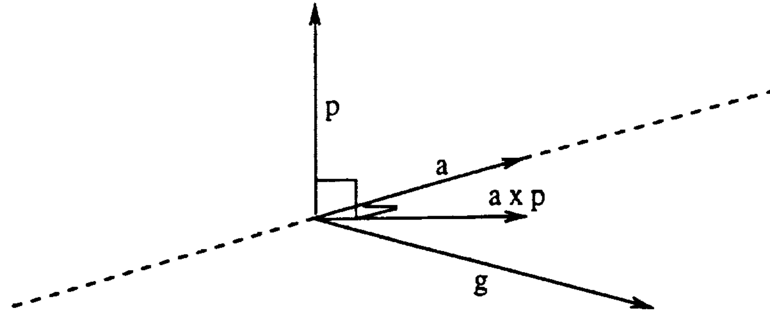

Figure 4.2: Sense of rotation

C_{ar} is again a field constant. All this yields $\mathbf{x_a}$, a direction vector in cartesian space. In order to avoid inverse kinematics, a direction vector in joint space is needed. The direction vectors in either space can be considered velocity vectors, so the manipulator Jacobian $\mathbf{J}$ can be used to transform the direction vector from cartesian to joint space. The derivation of $\mathbf{J}$ is discussed in subsection 4.2.3, so let us assume for now that it is available to us. In order to obtain $\mathbf{\Delta q_a}$ from equation 4.1 we need to solve the following linear system:

$$\mathbf{J \Delta q_a = x_a} \tag{4.13}$$

The Jacobian $\mathbf{J}$ is a 6 by n matrix, where n is the number of degrees of freedom of the manipulator. If $n > 6$ then the linear system is underdetermined, so there is an infinite number of solutions for $\mathbf{\Delta q_a}$. In this case the term $\mathbf{\Delta q_a^T Q \Delta q_a}$ will be minimized, where $\mathbf{Q}$ is a weight matrix that will normally have nonzero terms on the diagonal only. This weight matrix is important for robots with both revolute and prismatic joints, since vector $\mathbf{q_a}$ has mixed units in this case. A method for solving this minimization problem is presented in subsection 4.2.6.

This method is a variation of iterative inverse kinematics algorithms that have been developed for redundant robots or for nonredundant robots with kinematic

properties for which no closed form inverse kinematics solution exists. [1], [2], [3], [11], [12], [13], [14].

4.2.2 Obstacle Avoidance

The goal of this part of the algorithm is to keep the moving objects (the links and the payload of the robot) from colliding with items in the environment and also to avoid collisions among the moving parts. These goals are achieved by two separate algorithms that both contribute a joint vector to $\mathbf{\Delta q_{rc}}$ from equation 4.2.

$$\mathbf{\Delta q_{rc}} = \mathbf{\Delta q_{env}} + \mathbf{\Delta q_{self}} \tag{4.14}$$

Before beginning the discussion of the two methods, it is worth mentioning that the moving parts are the manipulator links *and* the object held by the manipulator. The payload is treated exactly like a manipulator link.

The idea behind both methods for collision avoidance is to go through all pairs of items that may collide and for each pair go through all joints that may change the distance between the items of the pair. The joints that have an influence on the distance of two objects are located between the objects in the kinematic chain. Adding an increment to these joint values may increase or decrease the distance or it may have no effect on the distance at all. Figure 4.3 shows two items modeled using the cylindrical model presented in chapter 3. The joint in Figure 4.3 is a revolute joint; prismatic joints are discussed later.

The points $\mathbf{z_p}$ and $\mathbf{z_q}$ are the two closest possible points on the line segments of the models, they are found using the algorithm described in chapter 3. The distance d is the minimum distance between the models (see equation 3.2). The vectors $\mathbf{a_o}$ and $\mathbf{a_d}$ represent the axis of rotation of the joint under consideration. $\mathbf{a_o}$ is the origin of the frame rotated by that joint and $\mathbf{a_d}$ is a unit vector representing the direction of the axis of rotation. Now we define a unit vector $\mathbf{r}$ pointing from $\mathbf{z_p}$

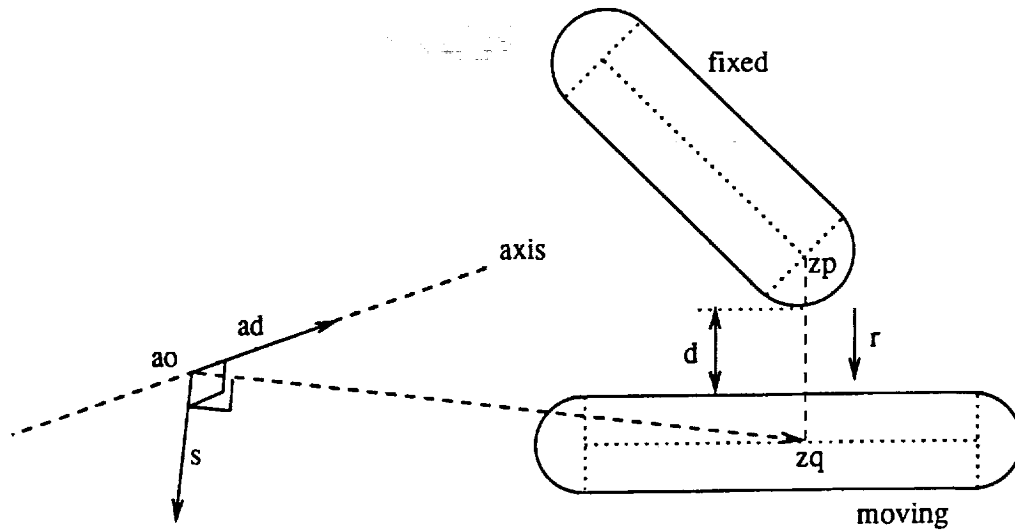

Figure 4.3: Basic collision avoidance procedure for a revolute joint

towards $\mathbf{z_q}$:

$$\mathbf{r} = \frac{\mathbf{z_q} - \mathbf{z_p}}{|\mathbf{z_q} - \mathbf{z_p}|} \tag{4.15}$$

We assume that model S_q is moving and model S_p is fixed, then vector $\mathbf{r}$ is the best possible direction in which model s_q and point $\mathbf{z_q}$ in particular can be moved in order to avoid a collision between the two models. We also define a vector $\mathbf{s}$ as follows:

$$\mathbf{s} = \mathbf{a_d} \times (\mathbf{z_q} - \mathbf{a_o}) \tag{4.16}$$

Vector $\mathbf{s}$ represents the direction in which point $\mathbf{z_q}$ *will* move when the joint is rotated in positive sense of rotation. Vector $\mathbf{s}$ is really this joint's column of the Jacobian matrix for point $\mathbf{z_q}$. Note that the absolute value of $\mathbf{s}$ is equal to the distance between $\mathbf{z_q}$ and the axis of rotation. Now we can define the joint increment δq_r that will be added to the component of this joint in the joint vector increment:

$$\delta q_r = C_r \frac{\mathbf{r} \cdot \mathbf{s}}{d^2} \tag{4.17}$$

It turns out that a degree of repulsion proportional to d^{-2} leads to good results. The repulsive force grows very fast when the models get close together. When the distance gets larger, the force goes down fast enough, so that there is almost no influence on the path when the distances to obstacles are reasonably large. The force is also proportional to $|\mathbf{s}|$ increasing the contribution of joints that have their axis of rotation far off the point of interest. These joints can cause a large increase in distance between the objects with comparatively small angles of rotation. The force is finally proportional to the dot product of $\mathbf{r}$ and $\mathbf{s}$ which is a degree of matching between the optimal direction to increase the critical distance and the direction that $\mathbf{z_q}$ will actually go if the current joint is moved. Note that this dot product can be negative, which means that the joint must be rotated in negative sense of rotation in order to increase the critical distance.

Figure 4.4 shows the same situation with a prismatic joint.

The variables $\mathbf{z_p}$, $\mathbf{z_q}$, $\mathbf{r}$, $\mathbf{a_o}$, $\mathbf{a_d}$ and d are defined as in the case of a revolute joint. The vector $\mathbf{a_d}$ is the direction of motion of this joint, so δq_p is defined as follows:

$$\delta q_p = C_p \frac{\mathbf{r} \cdot \mathbf{a_d}}{d^2} \tag{4.18}$$

Based on this elementary procedure we can define the joint vector increments $\mathbf{q_{env}}$ and $\mathbf{q_{self}}$ from equation 4.14. For this purpose we define a function $\mathbf{f_q}(\cdot,\cdot,\cdot)$ as follows:

$$\mathbf{f_q}(A, B, i) = [v_1 \ldots v_n]^T \qquad \text{where } n \text{ is the number of joints} \tag{4.19}$$

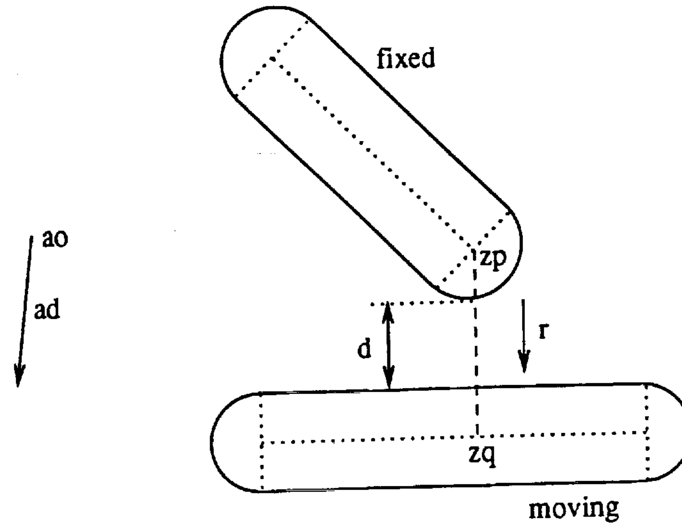

Figure 4.4: Basic collision avoidance procedure for a prismatic joint

$$v_j = \begin{cases} \delta q_r & \text{for } j = i \text{ and joint } i \text{ revolute} \\ \delta q_p & \text{for } j = i \text{ and joint } i \text{ prismatic} \\ 0 & \text{for } j \neq i \end{cases} \tag{4.20}$$

The points $\mathbf{z_p}$ and $\mathbf{z_q}$ and the distance d are found by applying the distance computation algorithm described in chapter 3 to the models A and B.

Furthermore, let $O_i, i \in [1 \ldots o]$ be the model of the ith obstacle in the environment and $L_j, j \in [1 \ldots l]$ the model of the jth link of the robot. Thus there are o obstacles and l links to be considered. Then $\mathbf{q_{env}}$ and $\mathbf{q_{self}}$ are defined as follows:

$$\mathbf{q_{env}} = \sum_{i=1}^{o} \sum_{j=1}^{l} \sum_{k=1}^{j} \mathbf{f_q}(O_i, L_j, k) \tag{4.21}$$

and

$$\mathbf{q_{self}} = \sum_{i=2}^{l} \sum_{j=1}^{i-1} \sum_{k=j+1}^{i} \mathbf{f_q}(L_i, L_j, k) \tag{4.22}$$

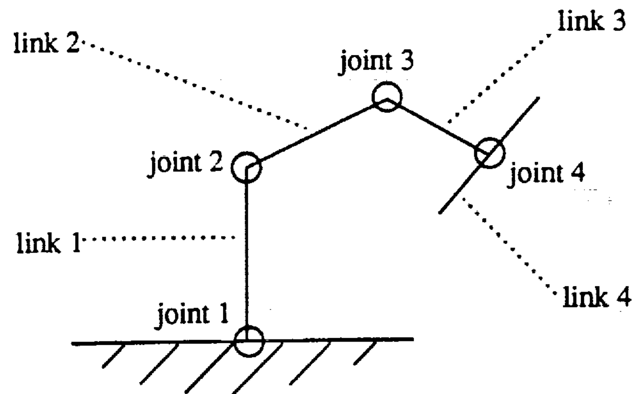

Figure 4.5: Numbering convention for the joints and links of the robot

The numbering convention of the joints and links is shown in Figure 4.5. Note that the robot's base must be modeled as an obstacle and not as a link.

This concludes the derivation of $\Delta \mathbf{q_{rc}}$.

4.2.3 Derivation of the manipulator Jacobian

Before proceeding to the joint range problem, let us discuss the derivation of the manipulator Jacobian $\mathbf{J}$. The origins of the link frames $(\mathbf{a_o})_i$ and the joint axes $(\mathbf{a_d})_i$ have been introduced in the previous subsection, as they are used for collision avoidance. Once they have been computed, the manipulator Jacobian can be obtained very easily and with relatively little computational effort:

$$\mathbf{J} = \left[\mathbf{j_1} \ \ \mathbf{j_2} \ \ \dots \ \ \mathbf{j_n} \right] \tag{4.23}$$

with n being the number of degrees of freedom of the manipulator. Then

$$\mathbf{j_i} = \left[\begin{array}{c} (\mathbf{a_d})_i \times (\mathbf{p} - (\mathbf{a_o})_i) \\ (\mathbf{a_d})_i \end{array} \right] \tag{4.24}$$

if joint i is a revolute joint or

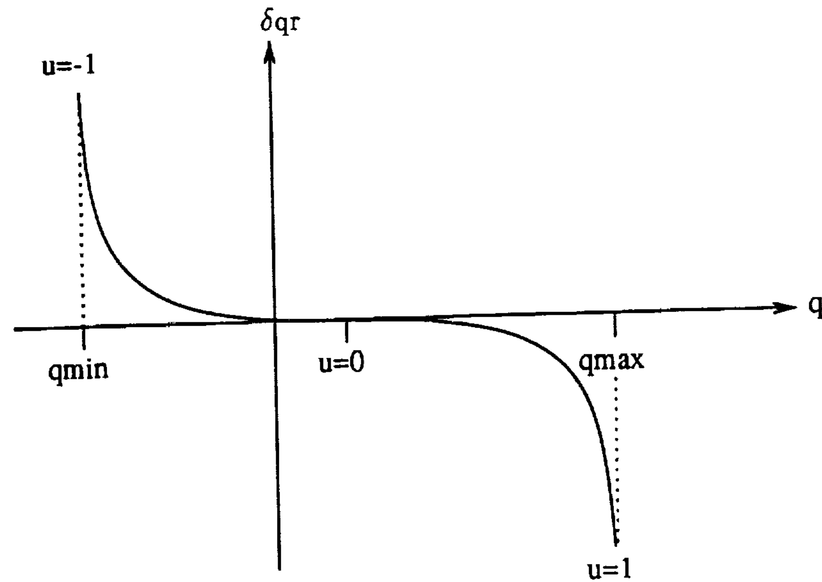

Figure 4.6: $\delta q_r(q)$

$$\mathbf{j_i} = \begin{bmatrix} (\mathbf{a_d})_i \\ \mathbf{0} \end{bmatrix} \quad (4.25)$$

if joint i is a prismatic joint; vector $\mathbf{p}$ is the current end effector position. The computation of a row corresponding to a revolute joint involves a cross product and a vector subtraction, while the rows corresponding to prismatic joints are already fully computed.

4.2.4 Joint Range

Besides avoiding collisions, a path planner must also guarantee that every joint vector on the path is in the robot's joint range. This goal can be achieved by monitoring the joint angles and creating a corrective joint value δq_R if an angle is close to an end of its range. These values for all joints form the joint vector $\mathbf{\Delta q_{rr}}$ from equation 4.2. The value δq_R is a function of the respective joint value q and should behave similarly to the function shown in Figure 4.6.

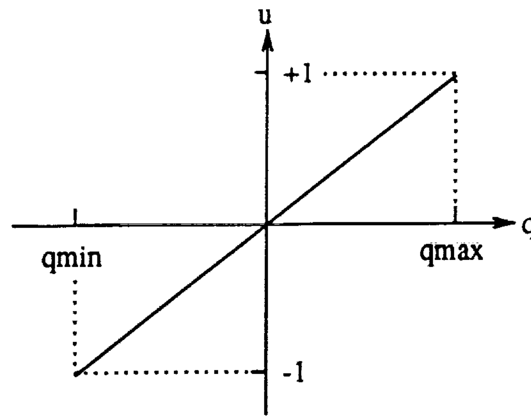

Figure 4.7: Domain mapping for joint range function

In our current implementation, the function δq_R is defined as follows:

The domain of q, $[q_{min} \ldots q_{max}]$, is first mapped onto the interval $[-1 \ldots 1]$ by applying the function

$$u(q) = \frac{q - \bar{q}}{R_q/2} \qquad \text{with } \bar{q} = \tfrac{q_{max}+q_{min}}{2} \text{ and } R_q = q_{max} - q_{min} \tag{4.26}$$

A graph of this mapping is shown in Figure 4.7.

On the domain u, a function $r(u)$ as shown in Figure 4.8 is defined:

$$r(u) = \frac{a}{u+b} + c \tag{4.27}$$

This function is used to define $\delta q_R(u)$ as follows:

$$\delta q_r(u) = \begin{cases} -C_R r(1-u) & \text{for } u \geq 0 \\ C_R r(u+1) & \text{for } u < 0 \end{cases} \tag{4.28}$$

The parameters a, b and c in equation 4.27 determine the shape of function $r(u)$, but their geometrical influence on the function is not intuitively clear. For this reason we introduce the parameters x_1 and y_0 also shown in Figure 4.8. From the three equations

$$y_0 = r(0) \tag{4.29}$$

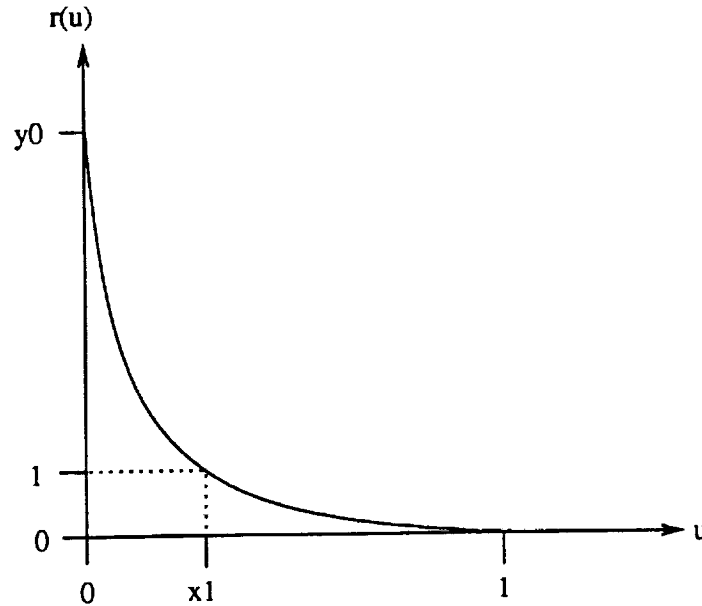

Figure 4.8: $r(u)$

$$\begin{aligned} 1 &= r(x_1) && (4.30) \\ 0 &= r(1) && (4.31) \end{aligned}$$

we can find the expressions for a, b and c in terms of x_1 and y_0:

$$\begin{aligned} c &= \frac{x_1 y_0}{1 - y_0 + x_1 y_0} && (4.32) \\ b &= -\frac{c}{y_0} && (4.33) \\ a &= -c(b+1) && (4.34) \end{aligned}$$

The values for x_1 and y_0 must be found by trial and error. It was determined that $x_1 = 0.2$ and $y_0 = 10$ lead to satisfactory results.

4.2.5 Imposing restrictions in cartesian space

Recall the attraction field described in subsection 4.2.1. Translation and rotation to the goal position is achieved by finding a direction vector and an axis of

rotation that describe the necessary motion (equation 4.3). Then we solve for the joint vector using the manipulator Jacobian (equation 4.13). The two algorithms that provide direction vector and axis of rotation are independent and the length of the direction vector as well as angle of rotation are constant for each step. A consequence of this approach is the fact that rotation and translation are generally not completed at the same time. In a given situation it might happen that the payload has reached the correct orientation, but it has not reached the goal yet. In this case we would detect that the correct orientation has been reached and thus set $\mathbf{x_{ar}}$ in equation 4.3 to zero. After solving equation 4.13 we obtain a joint vector increment that will not rotate the end effector when it is added to the current joint vector of the manipulator, so a rotation free motion is obtained during the remaining part of the path.

A problem arises when the joint vector increments for collision avoidance and joint range protection are added to $\mathbf{\Delta q_a}$ to obtain the final joint vector increment $\mathbf{\Delta q}$ (equation 4.1). Since the joint vector increments for obstacle avoidance and joint range protection are computed in joint space, they will in general result in both a translation and a rotation in cartesian space. This is undesirable, since it will affect the orientation or position, that has been determined to be equal to the goal.

To overcome this problem we add the two joint vector increments to obtain $\mathbf{\Delta q_r}$ as done in equation 4.2, and split the result into two orthogonal components $\mathbf{\Delta q_{r0}}$ and $\mathbf{\Delta q_{rx}}$ such that $\mathbf{\Delta q_{r0}}$ is in the null space of the Jacobian $\mathbf{J}$:

$$\mathbf{\Delta q_r} = \mathbf{\Delta q_{r0}} + \mathbf{\Delta q_{rx}} \tag{4.35}$$

$$\mathbf{\Delta q_{r0}} \cdot \mathbf{\Delta q_{rx}} = \mathbf{0} \tag{4.36}$$

$$\mathbf{J\Delta q_{r0}} = \mathbf{0} \tag{4.37}$$

If we use Δq_{r0} instead of Δq_r in equation 4.1, then the collision avoidance will not have any effect on the end effector position in cartesian space. Nevertheless, if the robot is redundant, the robot's joint vector will still change and increase the distance to obstacles and joint range boundaries.

As discussed earlier, we would like to stop either translational or rotational motion, once either has reached its goal. If we reduced $\mathbf{\Delta q_r}$ to the null space of the full Jacobian, then collision avoidance would have no effect on any cartesian motion, neither translation nor rotation. But if we assume that the rotation was completed first, then there is no reason to inhibit effects of collision avoidance on the translational motion. However, any effect on the rotational motion must be avoided, since the correct orientation has been reached. This partial effect on cartesian motion can be obtained by reducing the joint vector to the null space of a part of the Jacobian matrix. As we have seen in equations 4.24 and 4.25, the rows of the Jacobian have a translational and a rotational part. In this sense we can split the complete Jacobian matrix into a translational and a rotational part:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J_t} \\ \mathbf{J_r} \end{bmatrix} \tag{4.38}$$

Both $\mathbf{J_t}$ and $\mathbf{J_r}$ are 3 by n matrices, where n is the number of degrees of freedom of the manipulator. If we need to inhibit translational motion, we reduce $\mathbf{\Delta q_r}$ to the null space of $\mathbf{J_t}$ and if we need to inhibit rotation, we reduce it to $\mathbf{J_r}$'s null space.

There remains the question of how to reduce a joint vector to the null space of a Jacobian, complete or partial. The derivation follows:

$$\mathbf{J\Delta q_r} = \mathbf{J(\Delta q_{r0} + \Delta q_{rx})} \tag{4.39}$$

$$\mathbf{J\Delta q_r} = \mathbf{J\Delta q_{r0} + J\Delta q_{rx}} \tag{4.40}$$

$$\mathbf{J}\boldsymbol{\Delta}\mathbf{q_{r0}} = \mathbf{0} \tag{4.41}$$

$$\Rightarrow \mathbf{J}\boldsymbol{\Delta}\mathbf{q_{r}} = \mathbf{J}\boldsymbol{\Delta}\mathbf{q_{rx}} \tag{4.42}$$

Here we can find the minimum norm solution for $\boldsymbol{\Delta}\mathbf{q_{rx}}$ as described in subsection 4.2.6 below. Then

$$\boldsymbol{\Delta}\mathbf{q_{r0}} = \boldsymbol{\Delta}\mathbf{q_{r}} - \boldsymbol{\Delta}\mathbf{q_{rx}} \tag{4.43}$$

provides the null space component of $\boldsymbol{\Delta}\mathbf{q_{r}}$.

4.2.6 A method for solving general linear systems

Some of the methods discussed in the previous sections involve solving linear systems of the form

$$\mathbf{AX} = \mathbf{B} \tag{4.44}$$

so it is worthwhile to show a method of finding the solution $\mathbf{X}$. $\mathbf{A}$ is an n by m matrix, which leads to three different cases:

$n = m$ $\mathbf{A}$ is a square matrix; there is one solution if $\mathbf{A}$ is nonsingular.

$n > m$ The linear system is overdetermined; in general there is no solution, but we can find the least square error solution.

$n < m$ The linear system is underdetermined; let's assume that in this case the unknown is a vector ($\mathbf{x}$). There is an infinite number of solutions. We are interested in the solution that minimizes $\mathbf{x}^{\mathbf{T}}\mathbf{Q}\mathbf{x}$ where $\mathbf{Q}$ is a weight matrix. If $\mathbf{Q} = \mathbf{I}$, then the minimum norm solution is found.

Let us first describe the case of $\mathbf{A}$ being a square matrix, since the other two cases are based on that case.

A Gaussian elimination algorithm is used, so the original system must first be transformed into an equivalent system $\mathbf{A}'\mathbf{X} = \mathbf{B}'$ in which $\mathbf{A}'$ is an upper triangular matrix, and which has the same solution $\mathbf{X}$. The new system can then easily be solved by using back substitution. This transformation is done in $n-1$ steps known as Householder transformations. [5] A Householder transformation is a matrix $\mathbf{H}$ that transforms a vector $\mathbf{a}$ into another vector $\mathbf{b}$ of equal norm:

$$\mathbf{Ha} = \mathbf{b} \qquad \text{with} \quad |\mathbf{a}| = |\mathbf{b}| \tag{4.45}$$

$\mathbf{H}$ can be found from $\mathbf{a}$ and $\mathbf{b}$ with reasonable computational effort:

$$\mathbf{H} = \mathbf{I} - 2\mathbf{x}\mathbf{x}^{\mathbf{T}} \qquad \text{with} \quad \mathbf{x} = \frac{\mathbf{a}-\mathbf{b}}{|\mathbf{a}-\mathbf{b}|} \tag{4.46}$$

It is now possible to obtain the upper triangular matrix $\mathbf{A}'$ by applying $n-1$ Householder transformations to parts of matrix $\mathbf{A}$ each one transforming a column of $\mathbf{A}$ into a column of an upper triangular matrix. We start with the leftmost column $\mathbf{a}_1$ of $\mathbf{A}$ and find its norm $(n_a)_1$. Now we can find a Householder transformation $\mathbf{H}_1$ that transforms $\mathbf{a}_1$ into a vector with $(n_a)_1$ as the first element and all the other elements equal to zero, since the two vectors will clearly have the same norm. We obtain a new system of equations:

$$\mathbf{H}_1\mathbf{A}\mathbf{X} = \mathbf{H}_1\mathbf{B} \tag{4.47}$$

The new system matrix $\mathbf{H}_1\mathbf{A}$ has a particular structure:

$$\mathbf{H}_1\mathbf{A} = \left[\begin{array}{c|ccc} (n_a)_1 & a_{12} & \dots & a_{1n} \\ \hline 0 & & & \\ \vdots & & \mathbf{A}_2 & \\ 0 & & & \end{array}\right] \tag{4.48}$$

This is clearly the first step towards an upper triangular matrix. We repeat this procedure for the submatrix $\mathbf{A_2}$ and an equivalent submatrix of $\mathbf{H_1B}$, namely $\mathbf{H_1B}$ with the first row omitted. After the second step, the second column will contain all zeroes except the first two elements. It can be seen that this procedure leads to an upper triangular matrix after $n-1$ steps.

The advantage of using Householder transformations as opposed to the simple Gaussian elimination procedure is that Householder transformation exhibit better numerical properties. The numerical behavior can be further improved by exchanging the first column of the current submatrix with the column having the largest norm. Then the elements on the diagonal of the resulting upper triangular matrix will be sorted; the element in the top left corner will be the largest, the element in the bottom right corner the smallest element in the diagonal. All elements on the diagonal will be greater than zero, since they are the norms of the columns before transformation. This fact provides a very convenient way of checking whether the matrix is singular or not. If the determinant of a matrix is zero, then the matrix itself is singular. The determinant of an upper triangular matrix is equal to the product of the diagonal elements. Since the elements on the diagonal are sorted by absolute value, we just have to check the last element on the diagonal. If it is equal to zero, the matrix is singular; if it is greater than zero, the matrix is nonsingular. Note that the exchanges performed on the columns of the system matrix must also be performed on the rows of $\mathbf{X}$ after back substitution.

This covers the case of $\mathbf{A}$ being a square matrix. The other two cases can be reduced to the first case, as shown in the following. Let us begin with the case $n > m$, in which the system is overdetermined. In this case the least square error solution to the system can be found by solving the following system:

$$\mathbf{A^TAX} = \mathbf{A^TB} \qquad (4.49)$$

Finally, let us consider the case $n < m$ and let's assume that the unknown is the vector $\mathbf{x}$. We are interested in the solution that minimizes $\mathbf{x^T Q x}$ aubject to the constraint equation $\mathbf{Ax} = \mathbf{b}$. This is a minimization problem that can be solved by introducing a Lagrangian vector λ:

$$\min \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + \lambda^T(\mathbf{b} - \mathbf{A}\mathbf{x}) \tag{4.50}$$

The partial derivatives with respect to $\mathbf{x}$ and λ must equal zero:

$$\mathbf{Q}\mathbf{x} - \mathbf{A}^T\lambda = \mathbf{0} \tag{4.51}$$

$$\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0} \tag{4.52}$$

Premultiplying 4.51 with $\mathbf{Q}^{-1}$ leads to

$$\mathbf{Q}^{-1}\mathbf{A}^T\lambda = \mathbf{x} \tag{4.53}$$

Both λ and $\mathbf{x}$ are unknown in this equation, so we have $n + m$ unknowns. We can however obtain an equation with only λ as an unknown which reduces the number of unknowns to less than half, since λ is an n by 1 vector and $n < m$. So let us premultiply 4.53 with $\mathbf{A}$:

$$\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^T\lambda = \mathbf{A}\mathbf{x} \tag{4.54}$$

From equation 4.52 this leads to

$$\mathbf{A}\mathbf{Q}^{-1}\mathbf{A}^T\lambda = \mathbf{b} \tag{4.55}$$

The inverse of $\mathbf{Q}$ can be obtained very easily if $\mathbf{Q}$ is a diagonal matrix: The diagonal elements of $\mathbf{Q}^{-1}$ are the reciprocals of the diagonal elements of $\mathbf{Q}$. λ can be solved for as described above. The solution $\mathbf{x}$ can be found from equation 4.51:

$$\mathbf{x} = \mathbf{Q}^{-1}\mathbf{A}^{\mathbf{T}}\lambda \tag{4.56}$$

CHAPTER 5

COMBINING GLOBAL AND LOCAL PATH PLANNING METHODS

5.1 Potential field methods and their problems

Potential fields can be used effectively to produce safe, smooth paths around obstacles, but they have a serious drawback: If a path exists, but the potential field contains local minima, then there is no guarantee that the path will be found. In relatively simple two dimensional cases, it is often possible to choose the potential field such that it can be proven to contain no minima other than the global minimum in the goal. This task becomes very difficult when considering a strut modeled by a line segment moving in an environment of other line segments, and it becomes hopelessly complex when a reasonably accurate model of the manipulator is added.

Thus it is not possible to rely on a potential field method alone. This chapter describes how a potential field approach can be embedded into a backtracking algorithm that calls the potential field method under various conditions, until a path is found.

5.2 Different ways of performing a given task

As an example of a task, consider the insertion of a strut into an existing structure. For the case that a strut is symmetrical, there would be two ways to complete this task. Figure 5.1 shows the two possibilities. The strut is drawn asymmetrically for clarity.

In this example, the choice A is probably more likely to be successful than choice B, since the angle of rotation is smaller. This is of course not more than a heuristic statement and given certain joint angle configurations, it could very well turn out that B is successful and A runs out of joint range. But it is still preferable

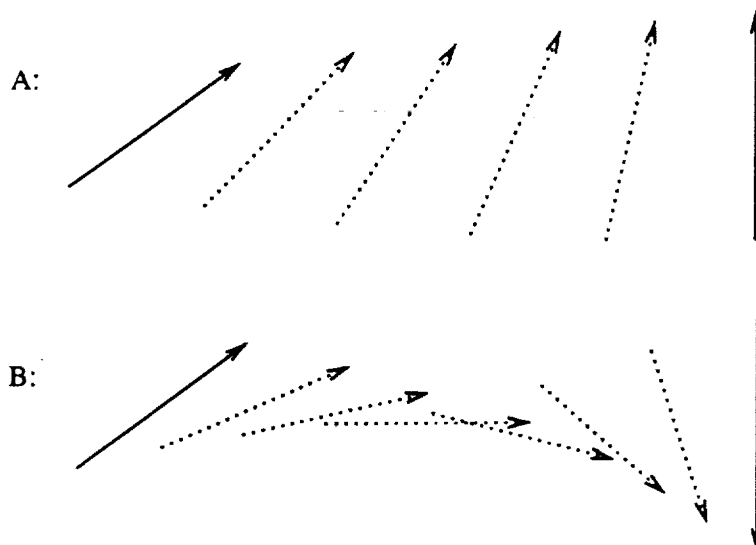

Figure 5.1: Two ways of inserting a strut

to try A first, since the probability for success is still higher and the resulting path also looks more natural. At this point we need an algorithm that finds the preferable orientation. Instead of comparing the angles of rotation involved in the two possibilities, which is somewhat difficult to find, we compare two much simpler expressions:

$$D = |\mathbf{g}_1 - \mathbf{p}_1| + |\mathbf{g}_2 - \mathbf{p}_2| \tag{5.1}$$

D is the sum of the distances between each end of the strut in its start and its goal position. We compare D_A and D_B, the distances for either orientation possibility; the smaller D value corresponds to the favorable orientation.

Once we have decided on an orientation, there are two possible ways to perform the task. In subsection 4.2.1 of chapter 4 the rotation of the gripper from its current orientation to its goal orientation was discussed. We obtained the axis of rotation by finding the eigenvector of the rotation matrix, that corresponds to eigenvalue 1. It was also mentioned that this eigenvector describes the axis of rotation that allows

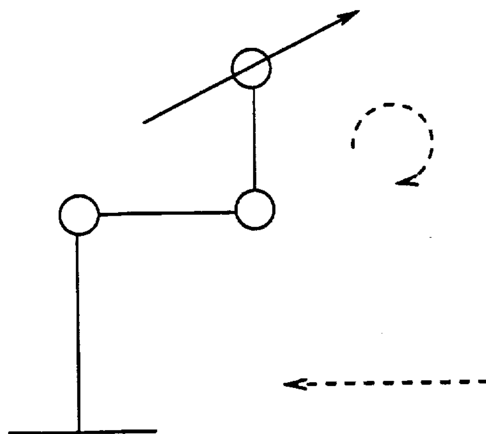

Figure 5.2: Choosing the correct sense of rotation

the gripper to reach its goal orientation, but it provides no information about the sense of rotation. In fact, we can reach the goal orientation no matter what sense of rotation we choose, but the angles of rotation will in general be different, since the sum of the two angles is always 360°. Here we will of course first try the rotation involving the smaller angle of rotation, but even this is not always the better choice. Consider Figure 5.2.

The indicated sense of rotation is clearly the only possible choice in this particular case, even though the angle of rotation is greater than 180°.

This yields four possible ways of performing the same task of moving a strut from a given start to a given goal position.

5.3 Splitting into subtasks using graph search

If none of the four cases leads to a successful completion of the potential field method, then it is necessary to split the task into subtasks. An example is shown in Figure 5.3.

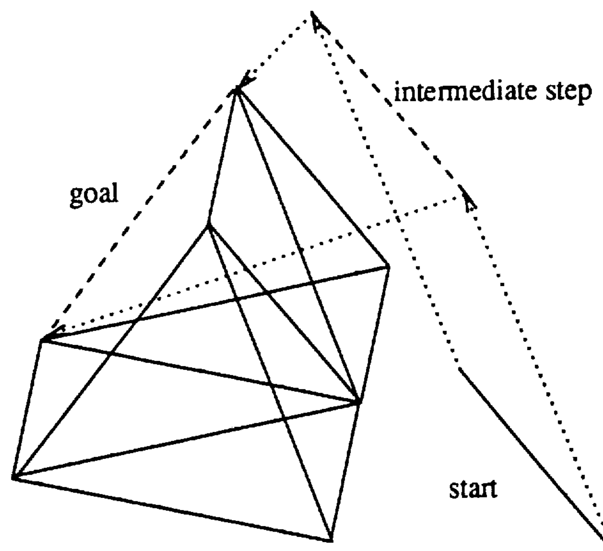

Figure 5.3: Splitting a task into two subtasks

The figure shows only one of many ways the intermediate step could be chosen. The idea is to provide a set of possible intermediate steps and use it to build an undirected graph. The nodes of this graph are the intermediate steps plus the start and the goal position of the strut. The selection of suitable intermediate steps is crucial for the success of the algorithm; it is described in subsection 5.3.2. After the nodes are created, edges are inserted such that every node is connected to every other node. In the example of Figure 5.3, we would obtain a graph with three nodes and three edges connecting them.

5.3.1 Application of the A* algorithm

Now we use the A* algorithm to find the shortest path leading from the start to the goal node. [4] We introduce the notion of distance by assigning a weight to every edge. This weight is the sum of the distance between the centers of the two node struts p and g attached to the edge and a measure for the degree of rotation involved:

$$w(p,g) = \frac{\left|\frac{\mathbf{p_1}+\mathbf{p_2}}{2} - \frac{\mathbf{g_1}+\mathbf{g_2}}{2}\right|}{s} + C_w(1 - (\mathbf{p_P} \cdot \mathbf{p_g})^2) \qquad \text{with} \quad s = |\mathbf{p_1} - \mathbf{p_2}| \tag{5.2}$$

Consider Figure 4.1 for the meaning of the vectors; $\mathbf{p_P}$ and $\mathbf{p_g}$ are unit vectors, s is the strut's length and C_w is a positive constant that determines the weight assigned to rotation differences. If this value is large, more weight is assigned to rotation. A rotation of 90° and a translation along the distance of sC_w have the same weight. Note that identical orientation and exactly opposite orientation are both leading to zero orientation weight, so struts are again assumed to be symmetrical.

The A* algorithm requires an estimate h of the cost of going from the node it currently works on (p) to the goal node. This estimate must be lower than the actual cost of reaching the goal in order to guarantee that A* will find the optimal path. In this application h can be defined using the weight function defined above:

$$h(p) = w(p, \text{goal}) \tag{5.3}$$

The first path found by the A* algorithm is very obvious: It consists of the one edge connecting start and goal node directly. This result is passed to the potential field stage, so before trying any intermediate steps, the algorithm will always give the potential field method a chance to find a path by itself. If it fails (after trying all four possibilities described in section 5.2), this edge is eliminated from the graph and A* is applied to the rest of the graph. The result will now include at least two edges; they are both passed to the potential field stage. As soon as one of the edges leads to a failure, this edge is removed from the graph and a new path is generated by A*. This procedure is repeated until the potential field method is successful on all edges of a path or until no path connecting the start and the goal node is left. In the latter case, all resources are exhausted and the program reports a failure to the supervisor level.

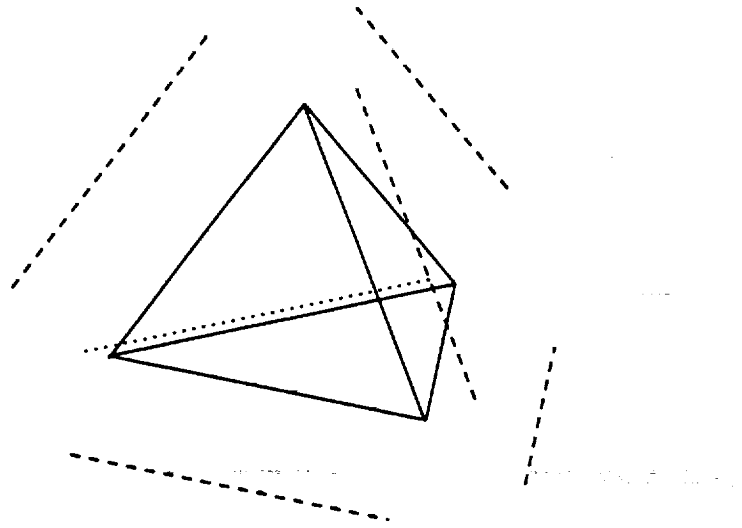

Figure 5.4: Intermediate steps placed around a tetrahedron

5.3.2 Selecting subgoals for the truss structure problem

The generation of subgoals is a very important step in every global path planner. There should not be too many intermediate steps, since the number of paths grows exponentially with the number of nodes in the graph. The intermediate steps should also be placed in positions where they can help to guide the potential field algorithm around difficult areas. So examples of good intermediate steps are corners of obstacles.

The approach taken for the specific case of truss structures is simple: Since the tetrahedron is the unit of all larger structures, all tetrahedra are extracted from the list of struts in the environment. When five of the six struts of a tetrahedron are present, then the structure is recognized and treated as a tetrahedron. The extraction algorithm is presented in subsection 5.3.3 below. Then an intermediate step is placed along every edge of every tetrahedron, as shown in Figure 5.4 for one tetrahedron.

This approach could be improved in the two following ways:

- Generate an approximate model of the robot's workspace and omit intermediate steps outside the workspace. If an inverse kinematics procedure is available, apply it on every intermediate step and reject it, if the inverse kinematics has no solution.

- Find the convex hull of the existing structure and accept only intermediate steps outside the convex hull.

While this approach is described here for specific structural models, analogous methods may be used for many common polyhedral models of objects.

5.3.3 Extraction of tetrahedra

The input to the program is a list of struts given by their positions in cartesian space. There is no a priori information about existence and location of tetrahedra, and they must be extracted from the list of struts.

A tetrahedron as a three dimensional object consists of four different geometric primitives:

- four vertices (0-dimensional)
- six edges (1-dimensional)
- four faces (2-dimensional)
- the tetrahedron itself (3-dimensional)

The edges represent the struts in the tetrahedron. The vertices are found by traversing the edge list and checking the two ends of each edge. If there is already a vertex in the vertex list at the position of a given end, then we create a pointer from the edge to that vertex, otherwise we create a new vertex at that position and a pointer from the edge to the new vertex. Every new vertex is put into the vertex list.

The extraction of the faces is more complicated. Again we traverse the list of edges and do the following for each edge e: It turns out that in a structure based on tetrahedra as shown in Figure 1.1 every vertex can have at most 12 edges attached to it. So for every end of e we traverse the edge list and store every edge that points to the same vertex this end of e points to. This procedure produces two lists with a maximum of twelve edges in it — the neighbors of the two ends of e. Then we traverse one of the lists of neighbors and for every neighbor n_1 we traverse the other list of neighbors and call the elements in this list n_2. If a pair $(n_1 \quad n_2)$ has a common node, then we detected a face bounded by the edges e, n_1 and n_2. Now we just have to make sure that this face was not detected before. This can be done by computing the center of gravity of the new face and comparing it to the centers of gravity of the other faces. If there is no face at the same position, we add it to the face list. Note that in the structures we are considering, there are never two different faces with the same center of gravity. If $\mathbf{p_1}$, $\mathbf{p_2}$ and $\mathbf{p_3}$ are the vectors representing the three vertices of a face, then the center of gravity can be obtained as follows:

$$\mathbf{p_c} = \frac{\mathbf{p_1} + \mathbf{p_2} + \mathbf{p_3}}{3} \tag{5.4}$$

Having the list of faces, the extraction of the tetrahedra is not difficult. We traverse the list of faces and for every face f_1, we traverse the list of faces again and call the current face f_2. Now we compute the distance d between the centers of gravity of f_1 and f_2. If the faces are part of a common tetrahedron, then $d = s/3$ if s is the length of the struts. As mentioned before, we treat two faces of a tetrahedron as a full tetrahedron, since at most one strut is missing once we detected two faces. We again have to make sure that this tetrahedron was not detected earlier. This can again be done by comparing the centers of gravity. If the four vertices in a tetrahedron are represented by the vectors $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$ and $\mathbf{p_4}$, then the center of

gravity is:

$$\mathbf{p}_c = \frac{\mathbf{p}_1 + \mathbf{p}_2 + \mathbf{p}_3 + \mathbf{p}_4}{4} \tag{5.5}$$

This yields a list of tetrahedra found in the structure. All comparisons of vectors involved in the process must allow for a considerable error, since the position inputs are noisy real world data. In order to create the intermediate steps, we compute the vector pointing from the tetrahedron's center of gravity to the center of the edge under consideration. This vector must be multiplied with a constant that will specify the distance between the edges and the corresponding intermediate steps. The ends of the step can be obtained by adding this scaled vector to the ends of the edge in the tetrahedron.

CHAPTER 6
SOFTWARE DOCUMENTATION

The path planning algorithm described in the previous chapters has been implemented in C language under the UNIX operating system. It has been developed on the SUN 3/60 and SPARC workstations in CIRSSE. The graphical user interface is based on SUNcore, SUN Microsystems' graphics library, which restricts this implementation to SUN platforms. However, all other parts of the program are portable to other platforms.

6.1 Concepts

The code is divided into 15 modules each consisting of a code file (`filename.c`) and a header file (`filename.h`). The code files contain the public and private procedures, the private constants and type definitions and the private global variables. The header files contain public constants, type definitions and procedure declarations. After the declarations, every header file also contains a documentation of the module with an overview and a detailed description of all public procedures. Thus the header files should provide enough information to enable a user to use the module effectively.

6.1.1 Global variables

It was decided that no global variables should be visible from outside a module. All data transfer between the modules is carried out through procedure calls. This makes module communication slightly slower, but changes in the module are generally easier to accomplish without affecting the module interface.

6.1.2 Data types

A module typically contains one or more data types on which the procedures of the module operate. A typical data type as found in the module's header file may be defined as follows:

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

This could be the data type of a module implementing operations on complex numbers. As a convention, all instances of data types printed in all uppercase (COMPLEX) are allocated in the heap memory. These data types all have a procedure to make a new instance and a procedure to kill an instance associated to them:

```
COMPLEX *New_Complex ()

void Kill_Complex (c)
     COMPLEX *c;
```

As a convention, other modules using the complex number module only use the pointer returned by the creator procedure and never directly access data inside the data structure. So instructions as

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

are not allowed outside the complex number module. A procedure that adds the two complex numbers $4 + 5i$ and $2 - 8i$ would look as follows:

```
#include "complex.h"

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

  a = New_Complex ();              /* make three new instances     */
  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, put result to c */
  printf ("Result: %f + %f*i\n",
           Real (c), Imag (c));    /* print result                 */
  Kill_Complex (a);                /* remove instances from heap   */
  Kill_Complex (b);
  Kill_Complex (c);
}
```

It can be seen that the fields of the variables a, b and c are never accessed in the code, even though the C compiler would allow this, since the definition of data type `COMPLEX` resides in the header file which is included in the user module. If other modules actually accessed the fields directly, then a change in data type `COMPLEX` would necessitate changes in all modules that make use of complex numbers. With this convention, changes in the data type `COMPLEX` don't affect other modules.

6.1.3 Module hierarchy

There are two ways in which module A can make use of module B. A can use B's procedures and declare variables of a type defined in B. For this type of link, A would include the header file of B into its source file. This is the more common way of using another module. If A contains a public procedure that returns a value of a type defined in B, then A must include the header file of B into its own header file, in order to have B's type definitions available in A's header file. This leads to a problem, if a third module makes use of both modules A and B, and thus includes

both header files into its code file. Since header file B is included into header file A, it will be included twice and all types and procedures will be redeclared. In order to avoid this, the code in every header file is inclosed in the following structure:

```
#ifndef MODULE_LABEL
#define MODULE_LABEL

  definitions, declarations ...

#endif
```

It ensures that every header file is compiled only once. Figure 6.1 shows the complete module hierarchy. Solid lines stand for normal links and dotted lines represent an inclusion of a header file into another header file. Arrows point from the including to the included module.

6.1.4 Make

The whole program can be compiled and linked using the UNIX utility `make`. It checks the last update time of every file and decides which files need to be recompiled. `make` needs information about the dependencies among the files of the program. This information is stored in the `makefile`. Code files *and* header files are included in this dependency tree, so a change in a header file will lead to an automatic recompilation of all files that include this header file. The dependency tree reflects the hierarchy shown in Figure 6.1.

6.2 Documentation of modules

This section includes a description of each module of the program. The same descriptions appear in the overview sections of the module's header files.

The modules are ordered by increasing level in the hierarchy. This way the reader becomes familiar with the prerequisites as they are needed to understand the

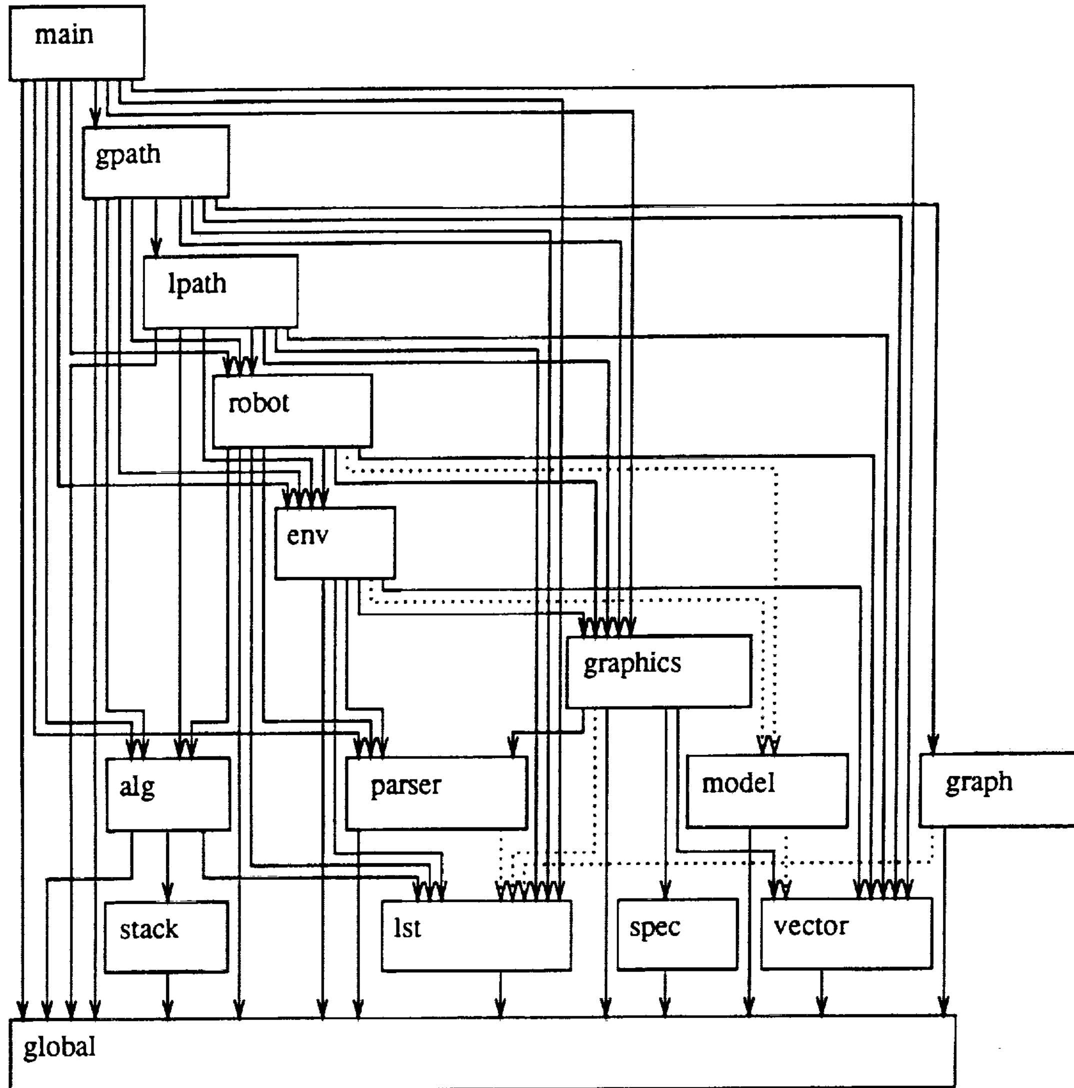

Figure 6.1: Module hierarchy

modules in higher levels.

6.2.1 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 constant Π and 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.

6.2.2 The "spec" module

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

6.2.3 The "lst" module

The list module provides a way of putting any kind 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. This is the main advantage of using this module over using a simple array.

The `LIST_EL` datatype contains a pointer `next` pointing to the next `LIST_EL` 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. The list structure is shown in figure 6.2.

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

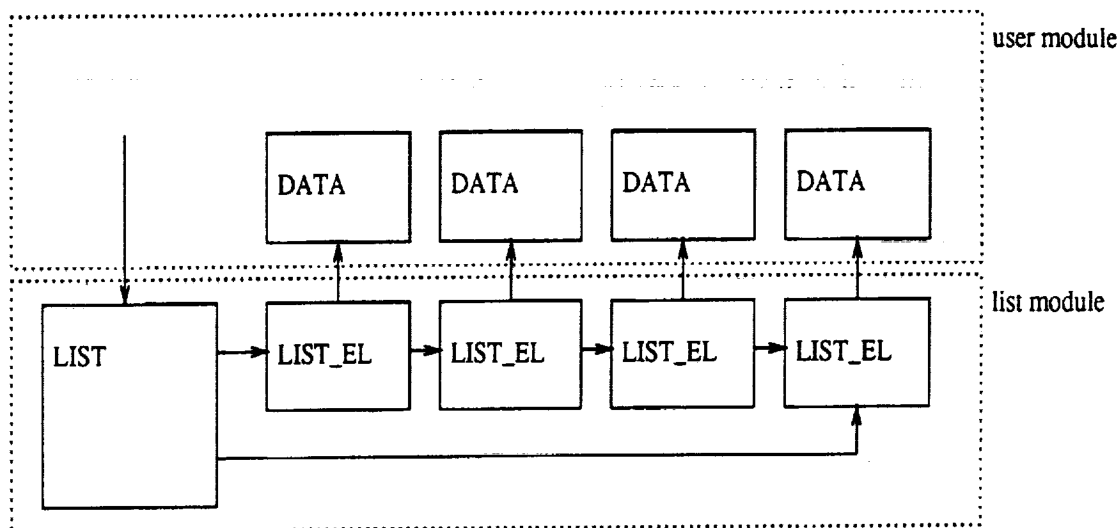

Figure 6.2: List structure

The 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 index contains the pointers to the data elements in a contiguous block of memory so that quick random 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.

6.2.4 The "stack" module

The stack module provides a way of organizing any data in stacks (LIFO - buffers). Each data entry is represented by an instance of STACK_EL. This data structure holds a pointer to the user's data and the chaining information. Access to the stack is accomplished by the procedures Push and Pop. The procedure Read_Top allows reading the latest entry on the stack without removing it.

6.2.5 The "vector" module

The vector module provides three data types:

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

The elements of the vector are floats, the columns of the 3x3 matrix are vectors and the homogeneous matrix is comprised of a matrix and a vector as the 4th column. All three data types are typically used for normal variable declarations; no instances of these types are allocated in memory.

The module also provides a set of useful operations on vectors and matrices. For instance the distance computation between two line segments as described in section 3 is realized in this module.

6.2.6 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 does not 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 are completed successfully and `FALSE` if a problem is encountered.

The algorithm for solving general linear systems as described in subsection 4.2.6 is implemented in this module as function `Vsolve`.

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

$$A = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} \qquad C = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

```
#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 a 2x1 variable  [1] */
                    3.0), A);         /*                         [3] */
  Put (Vuser (2, 2, 2.0, 0.0,         /* makes B a 2x2 matrix  [2 0] */
                    1.0, 3.0), B);    /*                       [1 3] */
  Put (Vuser (2, 1, 0.0,              /* makes C a 2x1 variable [ 0] */
                   -1.0), C);         /*                        [-1] */
  Put (Vadd (A, Vmult (B, C)), X);    /* eval 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               */
}
```

6.2.7 The "graph" module

The graph module provides a means 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 (or 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. These data structures are shown on Figure 6.3.

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*.

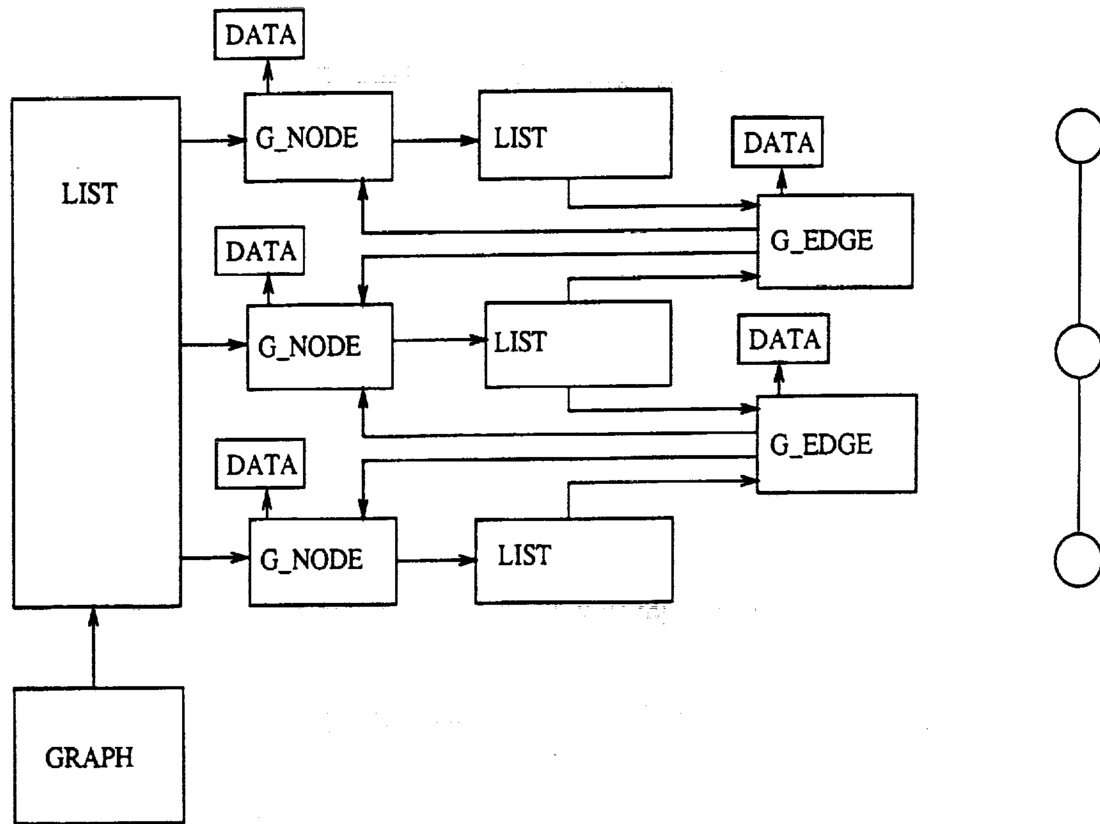

Figure 6.3: Data structures in the graph module

6.2.8 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.

6.2.9 The "model" module

This module provides a geometric primitive which is useful for modeling of solids. The primitive is described by two points $\mathbf{p}_1$ and $\mathbf{p}_2$ and a radius r.

It is the object obtained by moving a sphere of radius r on a straight line from point $\mathbf{p}_1$ to point $\mathbf{p}_2$ (a cylinder with spherical caps).

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

6.2.10 The "graphics" module

This module provides an interface to a subset of SUNcore that 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 screen axis by moving the mouse horizontally or vertically respectively. This mode stops in the current orientation when the middle mouse button is pressed.

The reason for using segments is the segment concept of SUNcore. The 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, but this process is visible on the screen, especially on slow machines. The segment concept allows the user to split the picture into parts, in order to avoid reconstruction of the whole picture when a single primitive is deleted.

6.2.11 The "env" module

"env" stands for environment, so this module holds the data about all items that belong neither to the robot nor to its payload. At initialization, the module reads the locations of the struts from the input file (see subsection 6.3.2 for strut position descriptions) and from the `CIRSSE` interface. Then it automatically tries to extract tetrahedra and places intermediate steps around the tetrahedra it found. This process is described in subsection 5.3.3.

Procedures are provided to get models of the struts and intermediate steps currently in the environment and to add and remove struts. Whenever a strut is added or removed, all intermediate steps are deleted, tetrahedra are extracted and the intermediate steps reestablished based on the new set of tetrahedra.

6.2.12 The "robot" module

This module contains a model of 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 the file `robot.def`, so that the files `robot.c` and `robot.h` can be applied for any single chain robot without changes. The robot's kinematics are described using Modified Denavit Hartenberg parameters [27]. The module maintains a set of transformation matrices that represent the transformation from

each link to world coordinates. They are derived from the modified Denavit Hartenberg parameters and the current joint vector and are updated each time the robot changes its joint vector. The module also maintains a swept sphere model of each reasonably large link and a picture comprised of a set of lines for each link. The model and the picture 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 and a part can be added to or removed from the gripper.

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

6.2.13 The "lpath" module

The path planning algorithm using potential fields as described in section 4.2 is implemented in this module. The user must specify the initial joint vector and the desired goal position in cartesian space and the module will return a list of joint vectors that describe a path leading there. If this is not possible, it returns a failure.

6.2.14 The "gpath" module

The global path planning algorithm as described in chapter 5 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.

6.2.15 The "main" module

The main module is responsible for initializing all other modules that require initialization. Then it checks to see if there is a command sequence in the input file. If it finds a `START` instruction, it takes the commands from the input file, otherwise it calls a CIRSSE interface procedure to read commands. This procedure is the main means of communication between this program and the higher level coordination program. Through it, commands are received and paths are returned.

The set of available commands is described below for both input file and CIRSSE interface.

6.3 Interface to CIRSSE

6.3.1 CIRSSE interface procedures

The communication of this program with other modules of the CIRSSE software is handled by dedicated interface procedures. They are marked with the string `####`, so they can be located easily using a text editor or the UNIX command `grep`. They all have extensive comments and an example implementation to show their purpose, but most of them are not actually written, since they depend on how the communication between the programs is implemented.

In the following, a list of the avaliable interface procedures is given:

- "main": initialize communication

  This procedure is called before any other interface procedure and can be used to establish the communication channels to other programs.

- "main": command interface

  The program receives commands and their parameters through this procedure. The following commands can be sent:

  - `MOVE`: initiate a path planning process

- GRASP: grasp a strut from the environment
- UNGRASP: release a strut to a specified position
- UNGRASP_FREE: release a strut where the robot put it
- ADD_STRUT: add a strut to the environment
- REMOVE_STRUT: remove a strut from the environment
- JOINTS: specify new joint vector
- VIEW: change orientation of screen display
- QUIT: quit program

- "main": path output

  The joint vectors describing the path are sent through this procedure. It is called once after every successful MOVE command.

- "main": report failure

  If a MOVE command cannot be executed because no path can be found, this procedure is called to report the failure to the coordinator.

- "robot": initial joint vector

  This procedure is called once at initialization time to read the joint vector of the robot. Then the program assumes that the robot follows its path. If this is not the case, then the joint vector can be adjusted in the course of the program by sending a JOINTS command.

- "env": initial strut positions

  This procedure is called once at initialization time to read the positions of all struts in cartesian space. The program will keep track of position changes in the course of the program if they are caused by the robot. It can be notified of other changes in the course of the program by using the ADD_STRUT and REMOVE_STRUT commands.

- "env": strut length

  The length of the struts in use is passed to the program through this procedure.

- "env": strut symmetry

  This procedure reads information about symmetry of the struts. A strut is considered symmetric in this context if it can be added to the structure upside down. The path planner will take advantage of these geometric properties.

- "robot": robot position

  The coordinate frame used to specify the initial strut positions is assumed to be the world coordinate system. If the robot's zero frame does not coincide with the world frame, then the position and orientation of the robot's frame with respect to the world frame can be specified here. If the 9 degrees of freedom robot of CIRSSE is used, this procedure is obsolete. However, if the 6 degree of freedom PUMA robot is used, then the position of the PUMA's base frame with respect to the lab's world frame must be specified here.

- "robot": robot definition

  This is the point where the file `robot.def` is included into the robot module. All information about a particular robot is stored in this file.

- "robot": calibration

  This procedure reads a set of Modified Denavit Hartenberg parameters obtained by some calibration procedure.

- "graphics": display coordinate transformation

  This procedure specifies the way the world frame is displayed on the screen. All three axes of the world frame can be displayed in six directions: up, down, left, right, pointing out of the screen, or pointing into the screen. This specifies the initial display which can be changed by user interaction when a `VIEW` command

is sent.

6.3.2 Input file

The input file is an auxiliary input source for the program that can be used for testing. When the program is embedded in CIRSSE, it will not need an input file, it will receive all input from the CIRSSE interface procedures. However, the two input sources are not exclusive, information can be passed to the program by using both CIRSSE interface procedures and an input file simultaneously. Usually the program will first check the file and then the CIRSSE interface for input. Some concepts are only available through the input file commands since they are not useful in an embedded system. For instance graphics display can only be activated through a command in the input file.

There are two ways of describing the position of a strut in space. A strut can be located anywhere in space, for instance in its storage location. In this case a convenient description is the position of the strut's endpoints in cartesian space. This allows the description of every possible position in space. If we want to describe the position of a strut that is part of the structure being assembled, the endpoint coordinate description is not the most convenient one, since the endpoint coordinates depend on the overall position and orientation of the structure and on the strut's position in the structure. It requires some calculation to find the endpoint coordinates. It is easier to specify the position and orientation of the structure once at initialization time and then describe a strut by giving the tetrahedron it is part of and the strut number in this tetrahedron, according to some numbering convention for the six struts of a tetrahedron. Consider Figure 6.4.

The coordinate system $(\mathbf{x_0}, \mathbf{y_0}, \mathbf{z_0})$ is the world coordinate system. Vector $\mathbf{p}$ describes the origin of the structure expressed in world coordinates. The vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ specify the orientation of the structure with respect to world coordinates.

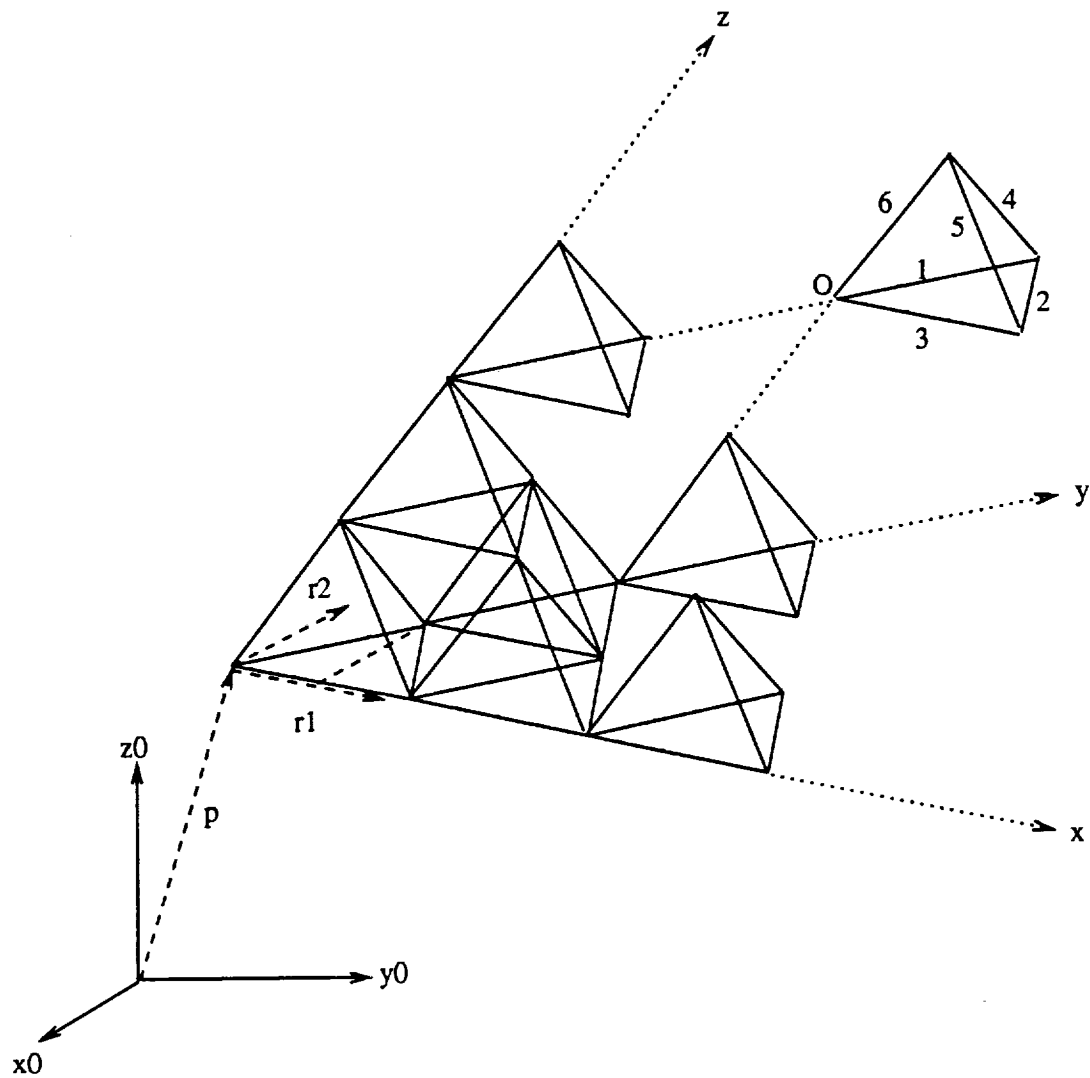

Figure 6.4: Conventions for specifying a strut in a structure

The coordinate system (x, y, z) is used to specify a tetrahedron in the structure. The lengths of this system's unit vectors $\mathbf{e_x}$, $\mathbf{e_y}$ and $\mathbf{e_z}$ are equal to the length of the tetrahedra's edges. Then every triple of integer numbers describes the O point of one particular tetrahedron. For instance $(0, 2, 2)$ describes the O point of the rightmost tetrahedron in the figure ($\mathbf{O} = \mathbf{p} + 0\mathbf{e_x} + 2\mathbf{e_y} + 2\mathbf{e_z}$). This tetrahedron also shows the numbering convention used to specify a particular strut in a given tetrahedron. Thus it is possible to describe a strut fully by giving the four integer numbers (X, Y, Z, N) once strut length and structure position and orientation have been specified.

In the following, the commands that can be used in input files are listed. The unit of length is meters, angles are given in degrees. The parameters `x1, y1, z1, x2, y2, z2` will always denote a strut position defined by its endpoints whereas parameters `X, Y, Z, N` denote a strut position defined by its position in the structure. The first list contains the commands that supply "static" information to the system. They are read at initialization time; their order in the input file is unimportant, as long as they do not appear between `START` and `QUIT`.

- `STRUT (x1, y1, z1, x2, y2, z2)`

  A strut defined by its endpoints is added to the environment. If necessary, the strut's length is adjusted to the current strut length such that the center stays fixed.

- `STRUT (X, Y, Z, N)`

  A strut defined by its position in the structure is added to the environment.

- `TETRA (X, Y, Z)`

  All six struts of the specified tetrahedron are added to the environment at once.

- `STRUTLENGTH (l)`

Specifies the length of the struts.

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

  Defines the location of the structure with respect to world coordinates. The parameters are vectors $\mathbf{p}$, $\mathbf{r}_1$ and $\mathbf{r}_2$ from Figure 6.4.

- GRAPHICS

  Causes the path planning process to be shown on the screen.

- ZOOM (z)

  Parameter z causes the display to shrink or expand.

- B&W

  Causes the graphics to be displayed in black and white even if the program is run on a machine with color screen. This is useful for making screendumps.

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

  With this command the zero frame of the robot can be oriented arbitrarily with respect to the world coordinate system. Vector $\mathbf{p}$ denotes the origin of the robot's zero frame with respect to world coordinates, $\mathbf{r}_1$ denotes the orientation of the robot's x_0 axis and $\mathbf{r}_2$ denotes the orientation of its y_0 axis.

The second list shows commands that can be used in a command sequence. This sequence begins at the START command and ends at the QUIT command or at the end of the input file. The commands in between are executed according to their order in the file.

- 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)

The program plans a path that leads the payload strut (imaginary or real) to the indicated position without collision. The vector **d** specifies the direction of approach. The path will be planned such that the last few steps of the path will move the strut in this direction. This direction vector **d** is stored and will determine the direction of start for the next `MOVE` command. The end effector will start moving in direction $-\mathbf{d}$ and then turn towards the goal location. The very first `MOVE` command immediately starts moving towards the goal location.

- `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 grasped by the robot. The path planner will treat this strut as another link. Recomputation of the set of intermediate steps takes place in the environment module.

- `JOINTS (theta1, theta2, ... ,thetaN)`

  The robot jumps to the new pose specified by the joint vector. The number of parameters of this command is equal to the number of degrees of freedom of the robot. Future `MOVE` commands start off with this joint vector. Units are degrees for revolute joints and meters for prismatic joints. Note that the CIRSSE interface procedure described above requires radians for revolute 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 user may want to move the strut to a position just short of the goal and leave the last few inches to a fine motion planner with visual feedback.

However the parameters of this command should be the precise goal position, so that the path planner has a correct model of the world. All three versions of `UNGRASP` also recompute the intermediate steps.

- `UNGRASP`

  Unlike the other two `UNGRASP` commands, this command releases the strut and places it in the environment exactly at the position the robot brought it to. This position might be slightly different from the position given in the previous `MOVE` command due to tolerances in the path planning algorithm.

- `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. If the robot is the only tool used to manipulate the environment, then `ADD_STRUT` and `REMOVE_STRUT` below should be obsolete. However they enable the coordinator to inform this program of changes in the environment that occurred due to other reasons.

- `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`

  Execution of the command sequence is stopped and the user can use the mouse to change the orientation of the display. Execution resumes when the user presses the middle mouse button. This command has no effect if there is no `GRAPHICS` command in the file.

- `QUIT`

  Denotes the end of the command sequence.

CHAPTER 7
RESULTS

The performance of the proposed algorithm has been tested in various simulations, the results are presented in this chapter.

A model of one of the robot arms at CIRSSE was used for all simulations. This robot is shown on Figure 7.1. It is a PUMA arm with six degrees of freedom, mounted on a platform with three degrees of freedom. Eight of the nine joints are revolute, the first joint is prismatic. The axes of all joints are shown in the figure. The model uses the correct kinematic parameters of the actual robot, however, the outlines are not to scale.

Figure 7.2 shows the trivial path planning task of moving a strut from an initial to a goal position. The same scene is shown from different points of view in order to help visualization of the three dimensional model. The path is displayed by showing the position of the strut in the robot's gripper at every iteration. In this simulation, the direction of start is along the z axis and the direction of approach is against the direction of the z axis. It can be seen that the program automatically generates an intermediate goal above the true goal position in order to approach the goal position in the specified direction. The seemingly unnecessary rotation during the first few iterations is due to the fact that the program starts rotation towards the goal orientation only after the distance of the end effector to its goal position has decreased under a critical value. Before this phase, the end effector orientation is dictated by collision avoidance and joint range requirements. In this case, the purpose of the rotation is to move the joints in the wrist towards the middle of their range.

Figure 7.3 shows the insertion of the last strut into a tetrahedron. This task involves avoiding a collision with the partially completed tetrahedron. Note that

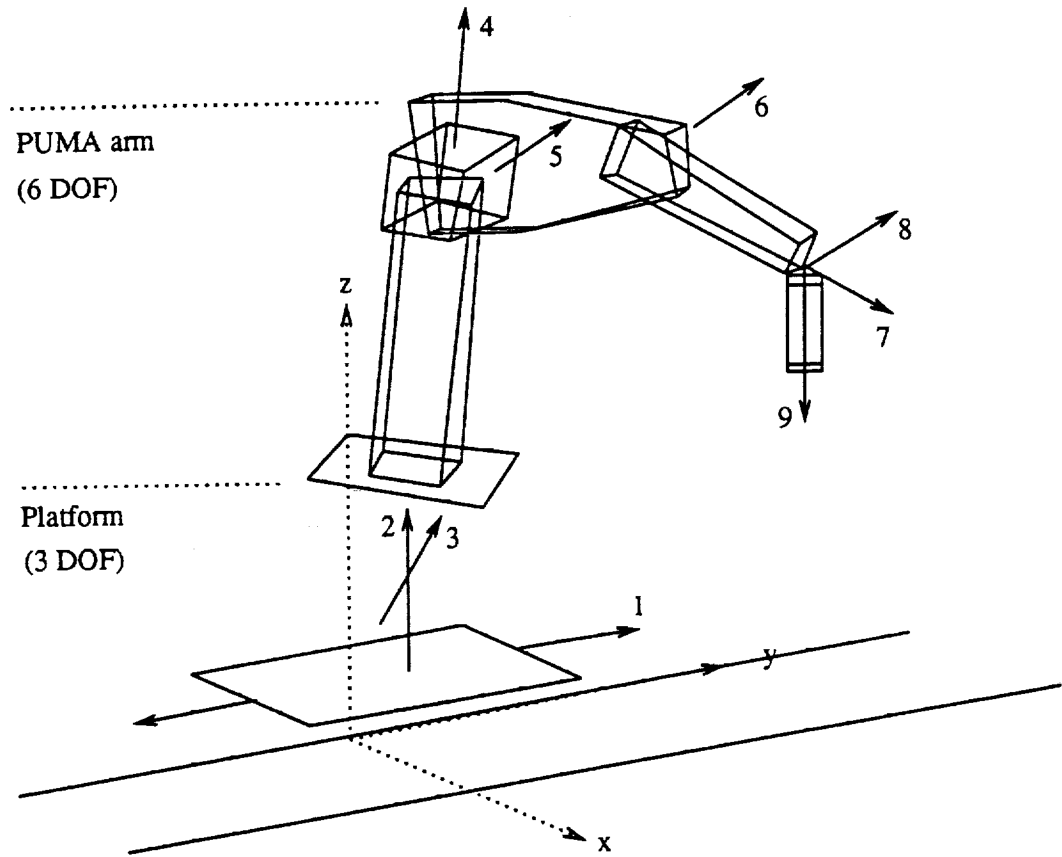

Figure 7.1: A robot arm of the CIRSSE testbed

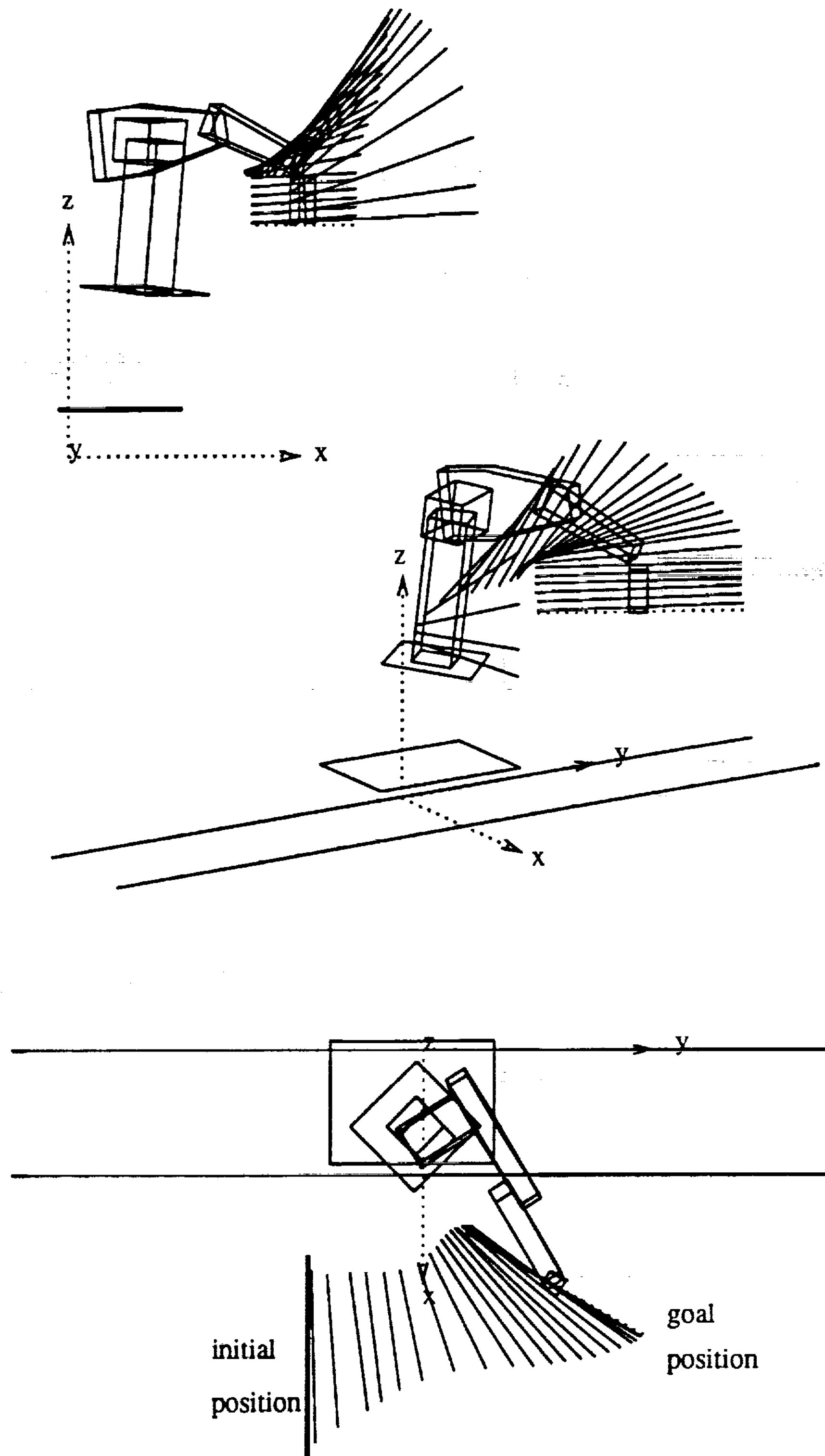

Figure 7.2: A simple motion task

the strut is touching the tetrahedron in its goal position. This is possible because all repulsive forces are reduced as the goal position and orientation are approached. The repulsive forces are reduced to zero when

- the strut's orientation is correct.
- the end effector's distance to its goal position is small enough.
- the end effector has reached a position such that a motion along the specified approach direction will lead to the goal position.

The same task is performed on Figure 7.4, but the approach direction is chosen slightly different (compare the end effector orientation in Figure 7.3). This leads to a failure when the natural rotation as in Figure 7.3 is attempted; the joint range is violated. As a consequence, the algorithm performs the task with opposite sense of rotation, which leads to a large angle of rotation. This case is discussed in section 5.2 and illustrated on Figure 5.2.

Figure 7.5 shows the initial situation of a more difficult problem. The robot must move the strut to the marked goal position appoaching it in x direction.

The result is shown on Figure 7.6.

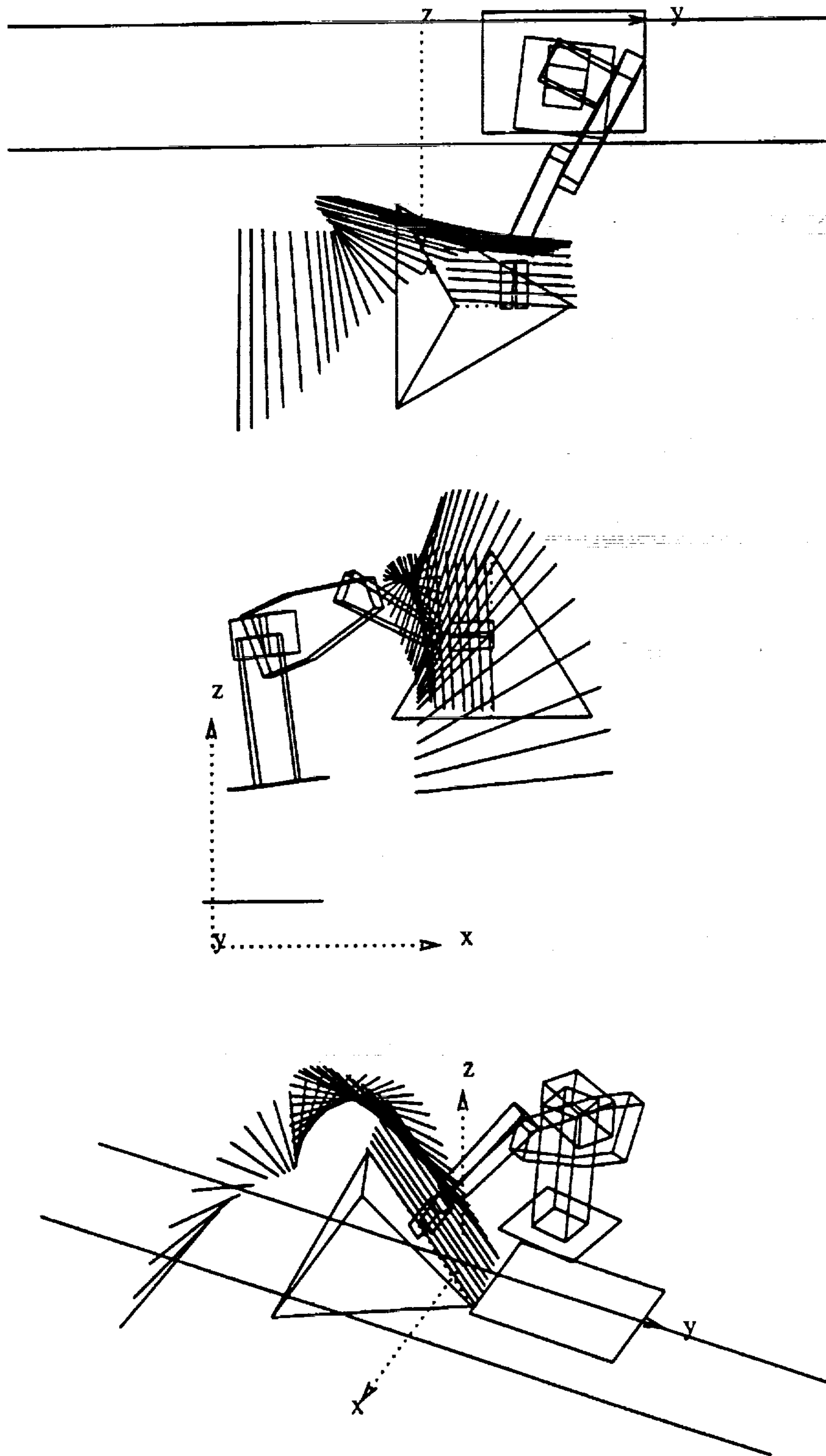

Figure 7.3: Completing a tetrahedron

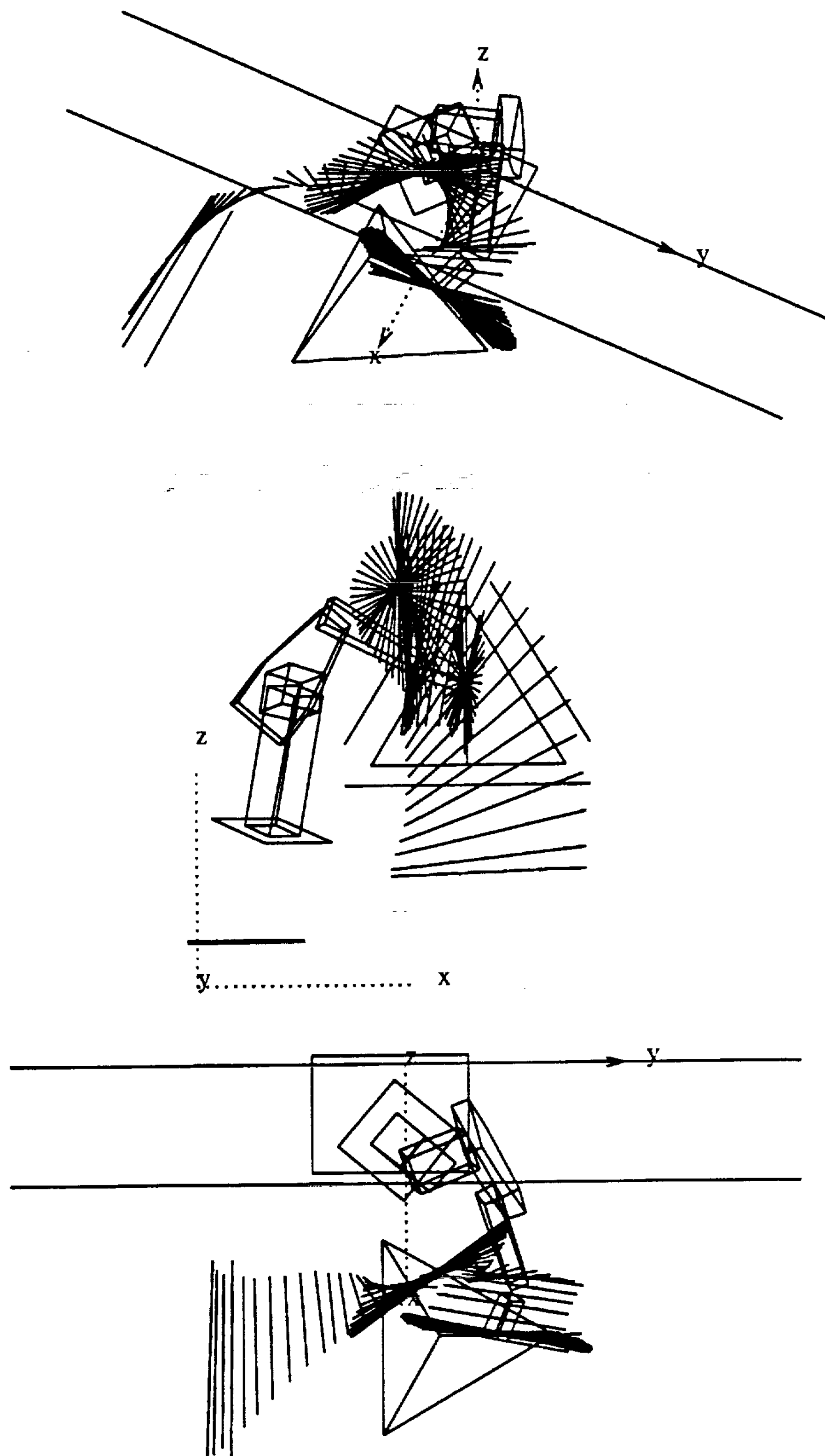

Figure 7.4: An example of a large angle of rotation

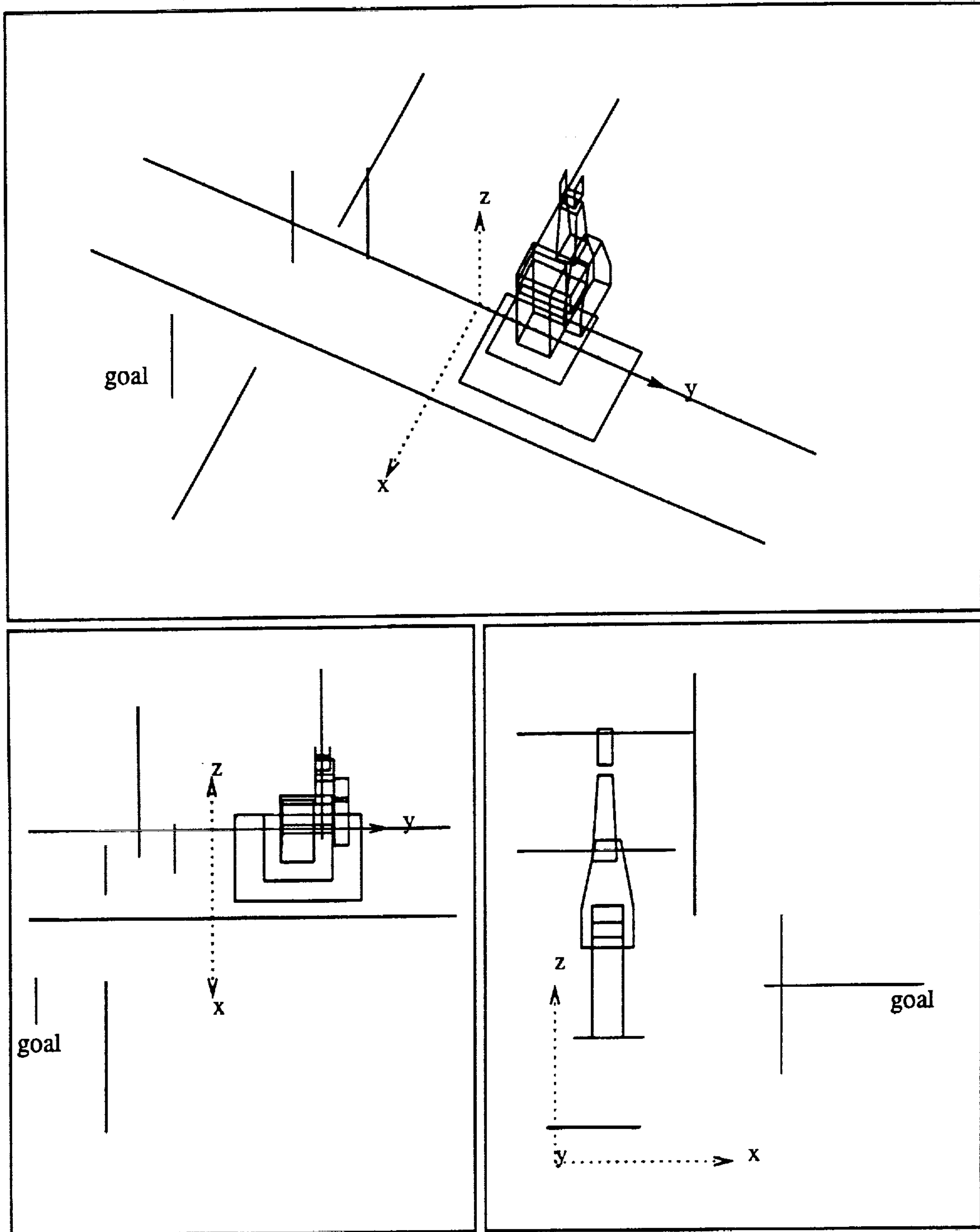

Figure 7.5: Initial state of obstacle avoidance demonstration

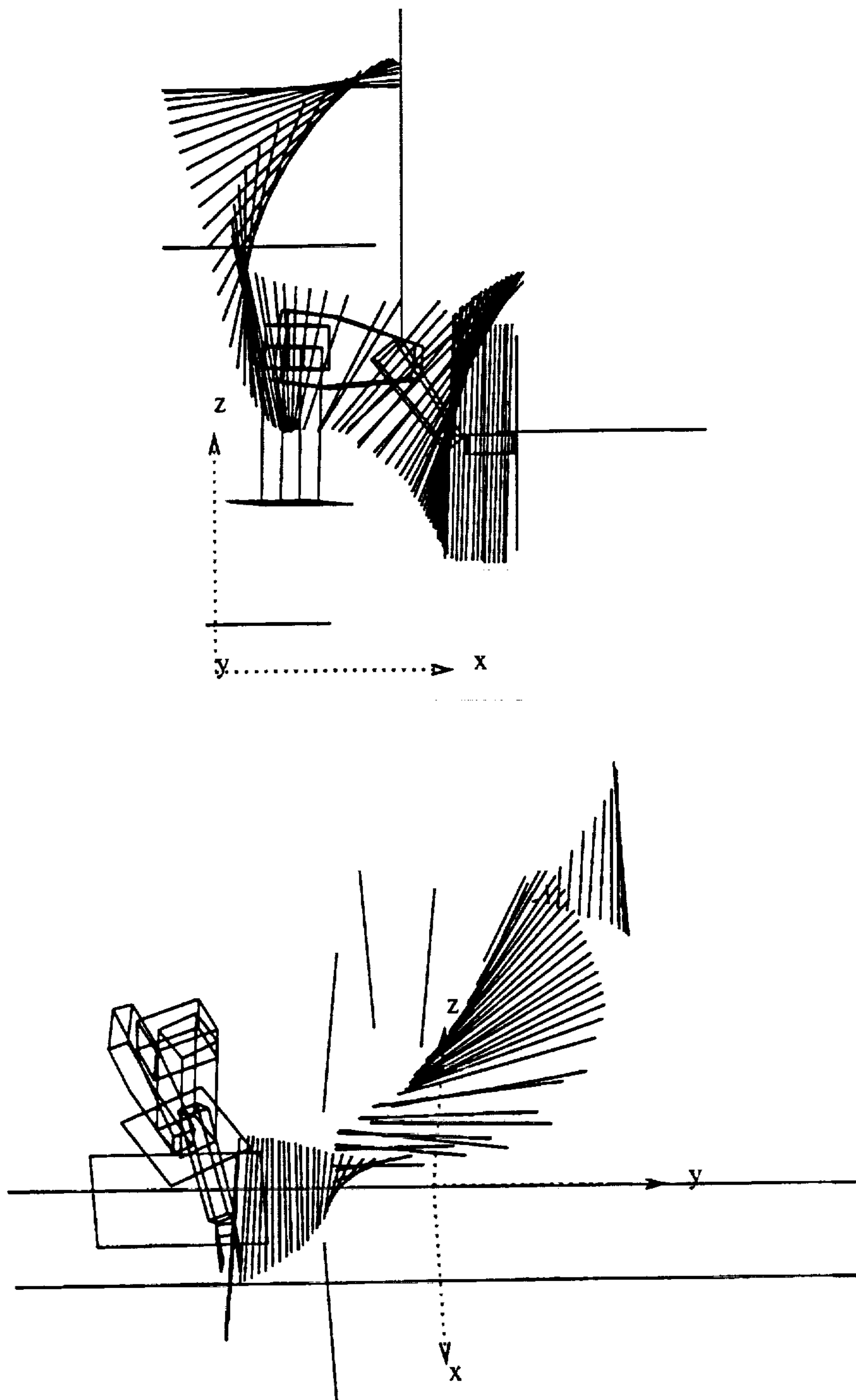

Figure 7.6: Obstacle avoidance demonstration

CHAPTER 8
DISCUSSION AND CONCLUSIONS

8.1 Computational Complexity

In this section, the complexity of the path planning algorithm is discussed. Due to their different structure, the local and the global stage are discussed separately.

The local path planner performs an iteration in three steps: It first finds the attractive joint vector, which involves finding a solution of the Jacobian equation as the limiting algorithm. The Jacobian is a 6 by n matrix, where n is the number of degrees of freedom of the manipulator. The algorithm involves a transformation of this redundant system to a nonredundant system with a 6 by 6 system matrix, as shown in subsection 4.2.6. Thus the actual elimination algorithm takes a constant computing time per iteration, but the transformation of the nonlinear system involves some matrix multiplications; their complexity is $O(n)$.

The second step in performing an iteration is the computation of the self collision avoidance vector $\mathbf{q_{self}}$. If equation 4.22 is considered and the computing time for $\mathbf{f_q}$ is assumed to be constant, then the complexity of $\mathbf{q_{self}}$ is $O(n^3)$.

The third step, the computation of the joint vector $\mathbf{q_{env}}$ avoiding collisions between links of the robot arm and obstacles in the environment has a complexity of $O(n^2m)$, where m is the number of obstacles in the environment. This can be seen from equation 4.21. These three steps turned out to be the limiting parts of the algorithm for all cases that have been examined.

Another step is the computation of the joint range vector $\Delta\mathbf{q_{rr}}$, which is linear in n. However, the computing time for this step is negligible for all n.

The number of degrees of freedom of the manipulator n is not a critical parameter since it will never be very large. A much more critical parameter for the

local path planner is the number of obstacles m, even though computing time grows only linearly with m. A possible solution to this problem is the implementation of a quick distance check between the manipulator and a given obstacle. If this distance was large enough, then this obstacle would be ignored in the collision avoidance procedure. Another method to reduce computing time is the exclusion of pairs of objects from the collision avoidance procedure that cannot possibly collide. This is typically the case for neighboring links and for collisions between very small links and obstacles. The small links can normally be included in the model representing a neighboring link. This method has proven to be very effective and has been implemented in the current software.

The global path planning stage is based on the A* algorithm, which is a very efficient graph search algorithm, but it still doesn't solve the search problem in polynomial time. So care must be taken to keep the number of subgoals as low as possible. However, in all but very large cases the local path planning stage is computationally much more expensive than the graph search.

If the local path planner fails to find a path between two given subgoals, then the corresponding edge will be deleted from the graph and the path planning algorithm will be restarted. If we assume that the number of subgoals is roughly proportional to the number of obstacles, then the number of edges in the graph grows quadratically with the number of obstacles, since every node in the graph is connected to every other node. This leads to a complexity of the backtracking algorithm of $O(m^2)$ in the number of obstacles.

Another stage in the global planning context is the selection of subgoals. In the example of truss assembly, this is done by first extracting the tetrahedra and then defining the subgoals close to their edges. The complexity of the extraction algorithm is $O(m^2)$ with m being the number of struts. The computing time of this part proved to be negligible compared to other parts.

8.2 Generalization of geometric model

The current implementation of the path planner uses the swept sphere cylindrical model to represent objects. However, every geometric model can be used, since the path planning algorithm only requires the computation of a distance vector between any two models. It is also required that the volume of the actual object is completely enclosed in the volume of the model representing it.

8.3 Experiences with the potential field algorithm

Potential field methods have the advantage that various soft constraints can be incorporated by adding up all components of the potential field. However, every component of the field has a constant factor associated to it and these constants must be carefully balanced in order to obtain good results. This is particularly important for the attractive field and the obstacle avoidance field. If the obstacle avoidance field is too strong compared to the attraction field, then a narrow but feasible path between two obstacles may not be found. If the obstacle avoidance field is chosen too weak, then collision avoidance cannot be guaranteed if the step size is large.

The step size is defined by the norm of $\Delta \mathbf{q}$, the incremental joint vector. If this step size is chosen small, then the local path planning process will be slow, since the computation time for one iteration is constant in a given environment. But if the step size is chosen too large, then problems occur when approaching an obstacle. One step might take the robot from a position with sufficient clearance to a position very close to the obstacle or even to a collision. A position very close to an obstacle leads to a strong repulsive force that drives the robot back out of the field's immediate proximity, which leads to oscillations in the path, as shown in Figure 8.1.

A related problem is the choice of the range of the repulsive force (equations

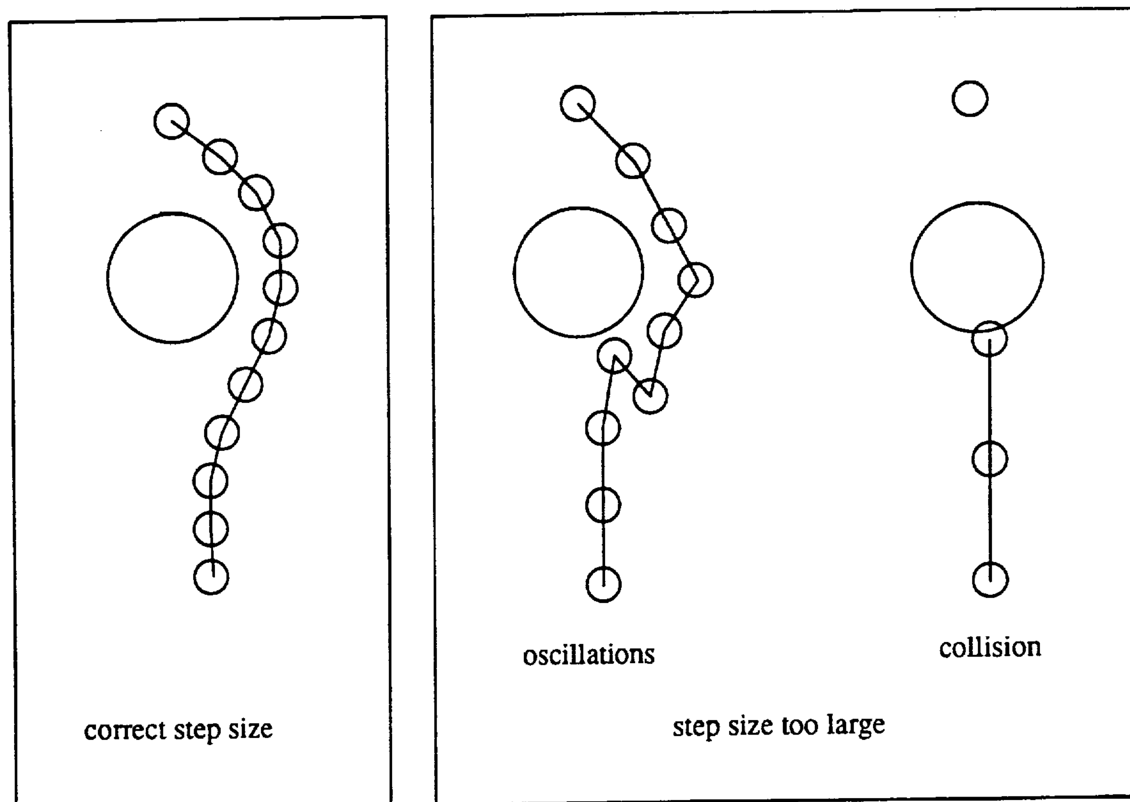

Figure 8.1: Step size selection

4.17 and 4.18). If the range is very small, then there will be a sudden increase in the repulsive force as the robot approaches an obstacle. Then oscillations may occur even if the step size is small. On the other hand, if the range is chosen too large, then the repulsive field of a cluster of several obstacles accumulates to an excessively strong field, which leads to a path with an unnecessarily large clearance to the cluster. The repulsive field range must be particularly carefully chosen, if any part of the path lies in the inside of a truss structure. In these cases it may be necessary to adapt the step size and the repulsive field range depending on the proximity of obstacles. This has not been tested in the current implementation.

The last parameter discussed in this context is the function that keeps the joints within their boundaries (see Figure 4.6). If this function permits the joints to run very close to their boundaries and the step size is large, then the joints may run out of range. However, if the function keeps the joints far off their boundaries, then the workspace of the robot is restricted unnecessarily.

8.4 Configurations and singularities

Another component to the potential field could be a field that prevents the robot from moving close to singularities. The potential field would be a function of $p = \det^2(\mathbf{J}\mathbf{J}^\mathbf{T})$. If $p = 0$ then the robot is in a singular position. The distance from singularities is related to the dexterity of the manipulator [14] in the sense that the robot is more dextrous if its pose is not close to any singular position. Another advantage of avoiding singular positions is that the Jacobian equation is guaranteed to have a solution. Apart from begin very difficult to implement in an efficient manner, singularity avoidance has another drawback: it prevents the robot from changing configuration.

A nonredundant robot normally has several joint angles in which it reaches a given end effector position and orientation. Every joint angle corresponds to a

configuration of the robot. A redundant robot has in general infinitely many joint angles for a given end effector position and orientation, but these joint angles lie on a number of self motion manifolds in joint space that correspond to the configurations of the nonredundant robot. The robot can only change these configuration manifolds by passing through a singularity. Now assume the robot is in a given initial position that lies in one of the configuration manifolds. It may not be possible to reach a goal position within this manifold due to joint range restrictions. Then the only way to reach the goal is to change configuration manifold, so it is necessary to pass through a singularity. Our algorithm does not force the robot to explore all configuration manifolds as part of the search process, but it also does not actively prohibit the robot from changing manifold by driving it away from singularities.

This concludes the discussion of the proposed path planning algorithm.

LITERATURE CITED

[1] 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.

[2] 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.

[3] 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.

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

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

[6] R. K. Mathur, A. C. Sanderson (1990). *A Hierarchical Planner for Space Truss Assembly* Proceedings SPIE Conference on Cooperative Intelligent Robotics in Space, Vol. 1387, R. J. de Figueiredo and W. E. Stoney, Editors, pp. 47-57.

[7] O. Khatib (1986). *Real–Time Obstacle Avoidance for Manipulators and Mobile Robots.* International Journal of Robotics Research, Vol. 5, No. 1, pp. 90 - 98.

[8] Pradeep Khosla, Richard Volpe (1988). *Superquadric Artificial Potentials for Obstacle Avoidance and Approach* IEEE 1988 International Conference On Robotics & Automation, Vol. 3, pp. 1778 - 1784.

[9] 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.

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

[11] M. S. Konstantinov, S. P. Patarinski, V. B. Zamanov, D. N. Nenchev. *A Contribution to the Inverse Kinematic Problem for Industrial Robots.* 12th International Symposium on Industrial Robots 1982, pp. 459 - 465.

[12] Vladimir J. Lumelsky (1984). *Iterative Coordinate Transformation Procedure for One Class of Robots* IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-14, No. 3, May/June 1984, pp. 500 - 505.

[13] Andrew A. Goldenberg, B. Benhabib, Robert G. Fenton (1985). *A Complete Generalized Solution to the Inverse Kinematics of Robots.* IEEE Journal of Robotics and Automation, Vol. RA-1, No. 1, March 1985, pp. 14 -19.

[14] Charles A. Klein, Bruce E. Blaho (1987). *Dexterity Measures for the Design and Control of Kinematically Redundant Manipulators.* International Journal of Robotics Research, Vol. 6, No. 2, Summer 1987, pp. 72 - 83.

[15] Shaygan Kheradpir, James S. Thorp (1987). *Robust Real Time Control of Robot Manipulators in the Presence of Obstacles.* IEEE 1987 International Conference On Robotics & Automation, Vol. 2, pp. 1146 - 1151.

[16] Elon Rimon, Daniel E. Koditschek (1988). *Exact Robot Navigation using Cost Functions: The Case of Distinct Spherical Boundaries in E^n.* IEEE 1987 International Conference On Robotics & Automation, Vol. 1, pp. 1 - 6.

[17] 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.

[18] R. F. Richbourg, Neil C. Rowe, Michael J. Zyda, Robert B. McGhee. *Solving Global, Two-Dimensional Routing Problems using Snell's Law and A* Search.* IEEE 1987 International Conference On Robotics & Automation, Vol. 3, pp. 1631 - 1636.

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

[20] 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.

[21] 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.

[22] Tomás Lozano Perez, 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] John J. Craig (1989). *Introduction to Robotics: Mechanics and Control.* 2nd edition, Addison–Wesley 1989, Chapter 3.

APPENDIX A

Simulation input file

```
{ Command sequence
  ---------------- }
  start
  view
{  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)
  view
{  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)
  view
{  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)
  view
{  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)
  view
{  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)
  view
{  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)
  view
  quit

{ Environment info
  ---------------- }
  graphics
  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)
  strutlength (0.89)
  zoom (2.0)
```

APPENDIX B

Listings of header files

Module *alg* (linear algebra):

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

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

   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' floats;
------------------------------------
```

Returns an r*c matrix with user defined contents. The floats in the parameter list are filled in the matrix from left to right and from top to bottom. The user must supply r*c floats 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;
        float s;
----------------------
```

Returns 'v' multiplied elementwise with 's'.

Example:

[1 2 3] multiplied with 2.0 is [2 4 6].

```
   float Vector_Norm (v)
        VAR *v;
------------------------
```

Returns the 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'.

```
   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). 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' floats;
--------------------------------------------------
```
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' floats after the 'num' parameter. If the matrix would overflow over the bottom right corner, an error occurs and no values are written at all.

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

```
   BOOLEAN Write_El (v, r, c, val)
        VAR *v;
        int r;
        int c;
        float val;
----------------------------------
```
Writes 'val' to the element at row 'r' and column 'c' in variable 'v'.

```
   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'.

```
   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.

   float 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_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 and intermediate
step models and are equivalent to the standard list readout procedures
described in the list module.

   BOOLEAN Add_Strut (p1, p2)
        Vector p1;
        Vector p2;
-----------------------------
Adds a strut to the environment. Its endpoints are at 'p1' and 'p2'.
Intermediate steps are deleted, tetrahedra reextracted and intermediate steps
recomputed.

   BOOLEAN Remove_Strut (p1, p2, rad)
        Vector p1;
        Vector p2;
        float *rad;
```

```
-----------------------------------
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 radius is returned
in 'rad'. FALSE is returned if there was 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;
        float *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*:

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

#define PI 3.141592654

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

#define BOOLEAN int
#define TRUE    1
#define FALSE   0

          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.

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

Module *gpath* (global path planner applying graph search):

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

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

LIST *Find_Path (path, p1, p2, dir)
     LIST *path;
     Vector p1;
     Vector p2;
     Vector dir;
--------------------------------------
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).
```

Module *graph* (graph theory):

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

  typedef struct graph
  {
    LIST   *nodelist;
    float  (*Weight) ();
    BOOLEAN directed;
  } GRAPH;
------------------------
The data structure for a graph. 'nodelist' is a list with all nodes of the
graph. The data type of the nodes is 'G_NODE' (see graph.c). 'Weight' is
a pointer to a function that returns the weight of an edge. The graph is
directed if 'directed' is TRUE, undirected otherwise.


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

  GRAPH *New_Graph (Weight, directed)
       float (*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:
   float 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 eqivalent 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;
        float (*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:

```
   float 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*:

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

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

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

   typedef struct line
   {
     Vector p1;
     Vector p2;
     int    color;
     int    style;
  } LINE;
----------------------
LINE represents a line on the screen. 'p1' and 'p2' are the endpoints,
'color' can take one of the values defined above and 'style' is either
SOLID, DASHED, or DOTTED.

   typedef struct character
   {
     char   c;
     Vector pos;
     int    color;
   } CHAR;
---------------------------
CHAR represents a character 'c' on the screen. 'pos' indicates its position,
'color' is one of the colors defined above.

   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.

```
   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'.
```

Module *lpath* (local path planning with potential fields):

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

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

  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'.

  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)
       VAR     *q;
       Vector  p1;
       Vector  p2;
       Vector  d;
       LIST    *path;
       LIST    *pict;
       SEG     *seg;
       BOOLEAN rot_normal;
-----------------------------------------------------------------------
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 *list*:

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

   typedef struct list_element
   {
     struct list_element *next;
     char                *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;
     int                  length;
     char               **index;
   } 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* (geometric modeling):

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

  typedef struct model
  {
    Vector p1, p2;
    float r;
  } MODEL;
-----------------------
MODEL represents a swept sphere geometrical model. Its volume is the volume
that a sphere of radius 'r' sweeps when moving from point 'p1' to point 'p2'
on a straight line.


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

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

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

   void Set_Model_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;
        float r;
-------------------------------
Changes model 'm's radius leaving its endpoints unaffected.

   float 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 (parameters 'p1', 'p2').

   void Get_Model_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;
```

```
        float  *r;
---------------------------------
Returns the model radius in 'r'.

   void Swap_Model_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 description
        ---------------------

  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_Float (par, v)
        char  *par;
        float *v;
------------------------------
```

Converts string 'par' into float '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 *robot*:

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

  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.

  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.

  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.

```
   float 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;
        float *lower;
        float *upper;
--------------------------------------
```

Returns the lowest and the highest possible value of joint 'nb'. The first joint is number 1.

```
   float 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)
        Vector p1;
        Vector p2;
        float  rad;
---------------------------------
```

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.

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

Empties the robot's gripper and returns the last endpoint positions of the payload.

```
   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* (functions providing CIRSSE/RAL specific information):

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

#define EARTH      0x110070e1
#define MARS       0x13004882
#define MERCURY    0x1700c726
#define JUPITER    0x1700cdd7
#define SOL        0x21000411
#define VENUS      0x5100c045
#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:
  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 *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)
       float x;
       float y;
       float 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;
        float s;
--------------------
Multiplies the elements of a with s and returns the result.

   Vector Div (a, s)
        Vector a;
        float 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].

   float 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.

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

   Vector Scale (a, len)
        Vector a;
        float 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;
        float len;
        float 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.

```
   float 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.

```
   float 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;
        float *t_plane1;
        float *t_plane2;
        float *t_line;
        float *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'.

   float 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'.

   Vector Random_Vector (len)
        float 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.
```