6-1-1991

AMP-CAD: Automatic Assembly Motion Planning Using C AD Models of Parts

S. N. Gottschlich
Purdue University

A. C. Kak
Purdue University

Follow this and additional works at: <https://docs.lib.purdue.edu/ecetr>

Gottschlich, S. N. and Kak, A. C., "AMP-CAD: Automatic Assembly Motion Planning Using C AD Models of Parts" (1991).
Department of Electrical and Computer Engineering Technical Reports. Paper 737.
<https://docs.lib.purdue.edu/ecetr/737>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

AMP-CAD: Automatic Assembly Motion Planning Using CAD Models of Parts

S. N. Gottschlich
A. C. Kak

TR-EE-91-12
June 1991

**AMP-CAD: AUTOMATIC ASSEMBLY MOTION
PLANNING USING CAD MODELS OF PARTS**

S. N. Gottschlich and A. C. Kak

**Technical Report TR-EE-91-12
June 1991**

**Robot Vision Lab
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907**

AMP-CAD: Automatic Assembly Motion Planning using CAD Models of Parts *

by

S. N. Gottschlich and A. C. Kak

Robot Vision Lab
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907

ABSTRACT

Assembly with robots involves two kinds of motions, those that are point-to-point and those that are force/torque guided, the former kind of motions being faster and more amenable to automatic planning and the latter kind being necessary for dealing with tight clearances. In this paper, we describe an assembly motion planning system that uses descriptions of assemblies and CAD models of parts to automatically figure out which motions should be point-to-point and which motions should be force/torque guided. Our planner uses graph search over a potential field representation of parts to calculate candidate assembly paths. Given the tolerances of the parts and other uncertainties, these paths are then analyzed for the likelihood of collisions. Those path segments that are prone to collisions are then marked for execution under force/torque control.

The calculation of the various motions is facilitated by an object-oriented and feature-based assembly representation. A highlight of this representation is the manner in which tolerance information is taken into account: Representation of, say, a part contains a pointer to the boundary representation of the part in its *most material condition* form. As first defined by Requicha, the most material condition form of a geometric entity is obtained by expanding all the convexities and shrinking all the concavities by relevant tolerances.

An integral part of the assembly motion planner is the execution unit. Residing in this unit is knowledge of the different types of automatic EDR (error detection and recovery) strategies. Therefore, during the execution of the force/torque guided motion, this unit invokes the EDR strategies appropriate to the geometric constraints relevant to the motion.

This system, called AMP-CAD, has been experimentally verified using a Cincinnati Milacron T^3 -726 robot and a Puma 762 robot on a variety of assemblies.

This work was supported by the National Science Foundation under Grant CDR 8803017 to the Engineering Research Center for Intelligent Manufacturing Systems.

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION	2
SECTION 2 - RELATED WORK	5
SECTION 3 - FEATURE BASED ASSEMBLY REPRESENTATION	7
3.1. Feature Classes and Features	7
3.2. Representing a Part	11
3.3. Representing an Assembly	11
3.4. An Example of Assembly Representation	16
3.5. Calculation of the Mating Pose of the Grasped Part	18
3.6. Finding Global Constraints on Feature Symmetries	21
SECTION 4 - ARTIFICIAL POTENTIAL FIELDS	23
4.1. Potential Field Definitions	25
4.1.1. Calculation of min_dist	26
4.1.2. Potential and Clearance of the Grasped Part	41
4.2. Alternative Methods of Path Planning	45
SECTION 5 - PHASE 1 PLANNING: IGNORING UNCERTAINTY	47
5.1. Stage 1	49
5.2. Stage 2	61
SECTION 6 - PHASE 2 PLANNING: INCORPORATION OF UNCERTAINTY	64
6.1. Representing Uncertainty	64
6.2. Converting the Path into a Sequence of Motion States	67
6.3. Final Motion Plan Generation	69
6.4. Motion Plan Execution	73
SECTION 7 - FINE MOTION STRATEGIES	76
7.1. Strategies for a Rotationally-Asymmetric Feature Pair	77
7.2. Strategies for Multiple Feature Pairs	81
SECTION 8 - GEAR BOX ASSEMBLY EXPERIMENTS	87
SECTION 9 - CONCLUSION AND FUTURE WORK	96
ACKNOWLEDGEMENTS	97
REFERENCES	98
APPENDIX 1 - FINDING LOCAL MINIMA ON A SURFACE OR CURVE	102
APPENDIX 2 - AMP-CAD SOURCE CODE	105

1. INTRODUCTION

In any assembly operation there is always some uncertainty in the knowledge of the locations and the shapes of the parts to be mated; this uncertainty may exceed the clearances between the parts, which precludes the use of point-to-point motions, also known as *coarse motions*. In such cases, it becomes necessary to use motions that are guided by sensory feedback, especially of the force/torque kind. These latter types of motions are usually referred to as *fine motions* or *compliant motions*. An important goal of assembly motion planning is the automatic discovery of those plan segments where fine motions are required and the precise sensory-feedback and manipulation strategies to be used in the execution of the fine motions.

In our work we have developed a two-phase assembly motion planning system called *AMP-CAD*, diagramed in Fig. 1, that in Phase 1 ignores all the uncertainties and derives a sequence of motions under the assumption that the starting position of each part or each subassembly is known without error and that the dimensions of the parts are known precisely. Basically, in this phase of planning the system operates like a simple path planner. To illustrate, suppose the assembly calls for the piston shown in Fig. 2 to be placed in the engine block cylinder. Initially, our system will simply calculate a path such as the one shown in the figure. Of course, if the nominal diameter of the piston is larger than the nominal diameter of the cylinder, the planner would declare a failure.

Fig 1: The flow of control in AMP-CAD.

Fig 2: A path of the piston from START to GOAL involves tight clearances when passing through the cylinder but much looser clearances everywhere else.

Subsequently, in Phase 2 of planning, the system takes into account all the uncertainties and characterizes each segment of the path calculated during the initial phase by first testing it for the possibility of a collision. In all segments where collisions are expected, compliance will be introduced via force/torque guided motions. The method used to detect collision-prone segments of the Phase-1 path is based on first modifying the potential field representations of the parts used for Phase-1 path planning to take into account the tolerances and any pose uncertainties, and then computing the overall potential of the grasped part vis-a-vis the fixtured part -- if this computed potential is not less than some *a priori* specified threshold (theoretically infinity), a collision possibility is declared. A further characterization of each collision-prone segment is then made, based on a number of factors such as the types of uncertainties involved and the constraints on the grasped part during the operation. Finally a motion plan is generated by selecting a strategy for each path segment from a library of point-to-point and fine motion strategies. The motion strategies selected for the segments are based upon their characterization.

The assembly motion plan thus produced is shipped off to an execution unit which invokes appropriate error detection and recovery (EDR) routines to cope with the often unpredictable control errors encountered during the execution of compliant motion. These control errors may be caused by a number of factors, such as the ever-present noise in the output of the force/torque sensor, the sources of this noise being acceleration/deceleration effects, the unpredictable micro-collisions of the grasped part with the imperfections on the fixtured part, etc. (A discussion of these control errors can be found in [21]).

The representation used for parts and assemblies has a great bearing on the design of procedures used by AMP-CAD. Particularly important is the manner in which various uncertainties, especially the tolerance information, is handled. Since the discovery and characterization of

collision-prone segments of an assembly path calls for the processing of geometrical models of parts with and without uncertainties, the representation used for parts and assemblies must facilitate access to such models. Furthermore, while a good representation should cater to the needs of planning and execution, not to be forgotten is the fact that at some level the user has to specify the assembly. Therefore, an ideal representation should make only minimal demands of the user with regard to the amount of information needed, especially if this information can be generated by analyzing the geometry of the assembly; we are referring to the information needed ultimately for planning and execution.

To meet these algorithmic and interface requirements, a feature-based assembly representation was developed; the primitives of this representation are volumetric units that are meaningful from the standpoint of mating operations in assembly. A feature-based representation is to be contrasted with the more traditional boundary-based or CSG (constructive solid geometry) based representations in CAD; these CAD representations do not allow easy access to the features that one may have to reason over for the mating operations in an assembly. These shortcomings with the more traditional boundary and CSG based representations are exacerbated by the difficulty of incorporating in them vital information such as the tolerances, etc.

In our representation, each feature is an object represented by an attribute-value frame, with some of the attributes possessing procedural attachments for the automatic creation of the geometrical model of the part in its nominal condition, and in its most-material-condition, the latter condition being a means to capture the manufacturing tolerances associated with the feature [51]. There are also procedures, automatically invoked due to their attachment to the attributes, for computing the symmetry group of the feature, etc. At a higher level, a part, also an object represented by an attribute-value frame with procedural attachments for some of the attributes for the automatic creation of the relevant geometric models in the presence of uncertainty, is considered to be a union of features, where some of the features, such as a hole, may be subtractive. Finally, an assembly is represented by a tree, each node in the tree representing an assembly operation, and the leaves of the tree representing individual parts. In the attribute-value frame associated with each node resides information on which features from the two child nodes (the child nodes representing either parts or subassemblies) are to be mated.

The representation used for parts and assemblies -- since it plays such an important role in the workings of AMP-CAD -- will be discussed first in Section 3. Then in Section 4, we will briefly discuss relative merits of the competing approaches that can be used for the path planning portion of our system, followed by a more detailed presentation of how we have used the potential field method. The planning methodology itself will be presented in Sections 5 and 6. In Section 5, we will put to use the potential field representation of parts for path planning, ignoring all the uncertainties; and, in Section 6, we will then show how the uncertainties may be incorporated for the discovery and characterization of the collision-prone segments of an assembly path. Section 6 also includes a discussion on the motion execution unit whose function is to invoke the most appropriate error detection and recovery strategy for a compliant motion. For now, we will start with a literature survey in the next section.

2. RELATED WORK

Preliminary work in the field of assembly motion planning was aimed at developing fine motion strategies for various classes of assembly mating operations when the amount of uncertainty in the position and orientation of the parts in the assembly operation is such that the operation cannot be accomplished with simple point-to-point motions. Several researchers at the Charles Stark Draper lab [55, 64] developed a passive device for aligning a peg with a hole as it is being inserted into a hole assuming that the tip of the peg could somehow be brought within the entrance of the hole, usually assuming the hole will be chamfered thus allowing a larger opening and a much looser initial clearance.

Subsequent work in peg in hole insertion strategies has been aimed mostly at initially aligning the peg with the entrance of the hole when the initial clearance between the parts is not enough to guarantee that the peg could be brought to the entrance of the hole reliably. Inoue [26] developed a biased search strategy for inserting a loose fitting peg into a hole and a tilt and slide strategy for inserting a tight fitting peg into a hole. Strip [58] developed a similar strategy that works for pegs whose cross-section has a convex (not necessarily round) shape, so that uncertainty in the orientation of the peg about the mating axis can be accounted for as well. Caine, Lozano-Perez, and Seering [6] also developed a similar strategy that is limited to the insertion of rectangular solid pegs into rectangular solid holes.

In our experimental work we found that attempting to untilt a very tight fitting peg in a hole can sometimes cause the peg to become wedged in the hole, particularly when attempted by a robot with error-prone compliant motion control. We therefore developed an alternative to the tilt and slide strategy [21] that calls for bringing the peg as near as possible to the entrance of the hole. At this point there will be a torque exerted on the peg by the surface containing the hole and this torque is such that the peg will tend to tilt into the hole. Therefore by measuring the torque on the peg we can deduce which direction to slide the peg along in order to align it with the entrance of the hole.

Peshkin [47] has developed a method for finding a fine motion control matrix so that when the motion is executed, forces and torques exerted on the grasped part will cause the grasped part to be deflected in such a way that it will align itself with the fixtured part during the mating operation. Peshkin's work is a general approach for developing fine-motion strategies, but his method relies on using force/torque data alone to differentiate between distinct contact configurations of the grasped and fixtured parts which is not always possible.

While all of the strategies we have talked about, excluding Peshkin's work, can be used to mate parts given uncertain information about them, they are all specific to "peg and hole" operations. It is desirable then to be able to develop motion plans for mating operations in the general case where the parts to be mated have an arbitrary geometry, the geometry being known within some degree of uncertainty.

Mason [42] showed how, given the task geometry, compliant motions could be selected to perform the desired task. Unfortunately Mason's work, which is not limited to assembly

situations, cannot be used solely as the basis for developing assembly motion plans for two reasons. First, it requires a task description that does not, in general, readily fall out of an assembly representation but must instead be deduced. Second, it does not account for any uncertainty in the knowledge of geometry and positions of parts. Currently we are aware of two competing approaches to the one that we will present in this manuscript.

Many researchers [33,36,37] have advanced an approach based on a contact-space representation of the parts. We will discuss this representation in Section 4; it essentially details the contacts that will exist between the grasped part and the fixtured part in the assembly for all possible positions and orientations of the grasped part. Currently the most complete planner based on this approach is that discussed in [37]. A motion plan is developed in Laugier's system by chaining together motions that maintain contact between the mating parts. In this approach then, a motion plan is seen as an ordered sequence of well chosen contacts. Laugier's planner operates in two phases. In the first phase a state-space graph is generated that represents all of the feasible contact situations between the grasped part and fixtured part and also represents all of the feasible motions between any two such contact situations. In the second phase, the graph is heuristically searched to find a path corresponding to a fine motion plan that can be carried out reliably under the prevailing uncertainties.

To reduce the complexity of the search for a fine motion strategy, this approach uses heuristics on many different levels that serve to limit its generality. For instance, when selecting a motion direction, it looks at motions that are tangent or normal to surfaces of obstacles that are near the object, but the best motion direction may be neither tangent to nor normal to the obstacle surfaces. The effects of uncertainty are analyzed by a set of production rules and fine motions are selected via a set of heuristics. These production rules and motion selection heuristics seem ill-defined and are not justified in [37]. Furthermore, at this point in time there is no good method for synthesizing purely rotational motions using his system, and motions that involve simultaneous translation and rotation cannot be handled at all.

A number of researchers [5,9,15,17] have developed planning systems stemming from what is commonly known as the "LMT approach" which was introduced in [40]. The most complete such planning system to date is the one that has been produced by Donald, [15] therefore in the following we will refer to Donald's planner although what we will say about Donald's planner is, for the most part, true about the other cited planning systems. Donald's planner casts the problem into configuration space; this being a mechanism for representing parts which will be discussed further in Section 4.

The planning system then tries to find a motion vector, that when followed, will move the grasped part from the starting position and orientation of the part, or the *starting configuration* to the *goal configuration* without sticking on any surface of the fixtured parts under the given uncertainty conditions. In the event that such a motion vector cannot be found, the planner tries to find a subgoal such that the subgoal configuration can be achieved from the starting configuration and the goal configuration can be achieved from the subgoal configuration with non-sticking motion vectors. Arbitrarily many subgoal configurations can be introduced in

connecting the starting configuration to the goal configuration. Finally, compliant motions corresponding to each motion vector are used to form a motion plan.

“Sticking” is a result of friction between parts. It is necessary for these systems to take friction explicitly into account because otherwise it is quite possible that a motion vector into a friction cone might be formulated. Because the motions developed by both Laugier’s system and our system either comply along surfaces contacting the grasped part or are formulated from fine motion strategies that implicitly take friction into account, there is no reason for Laugier’s planner or our planner to explicitly consider friction between parts.

There are several limitations of Donald’s system that our planner addresses. It is not clear that there are methods to create a configuration space representation for some assembly geometries, for instance when the center of mass of the grasped part lies in a void of the grasped part. There are many assembly situations for which this approach could not develop a motion plan even though these situations have been addressed by earlier research. For instance, the strategies discussed in [21, 26, 56, 58] all rely on using torque measurements to reduce the translational uncertainty of the part which may be applied in many situations, but Donald’s system is unable to formulate such fine motions because they do not fit into the model assumed for motion, this model being based on a diagonal compliance matrix.

3. FEATURE BASED ASSEMBLY REPRESENTATION

The usual geometric representations involve either some form of a faceted representation of the boundary surfaces of the object or a combination of Boolean operators applied to volumetric primitives. In either case, we lose access to what might be called features on the object. For example, in the usual boundary representation of a block with a hole, it would not be easy to isolate from the representation the surfaces that constitute the hole. To address this shortcoming, in recent years there has been an interest in developing feature-based approaches to object representation. Of course, since what constitutes a feature on an object depends on the application, the feature-based representations must of necessity be tailored to a specific application. For example, as shown in [10, 27] the feature-based representation for process planning is centered around features such as slots, pockets, etc., these features corresponding to machining operations. In this section, we will discuss our feature-based representation for assembly.

3.1 Feature Classes and Features

In our representation a feature is a volumetric unit of a part, where the volume might actually correspond to a hole or a void. Features are assumed to be instances of feature classes and may be either mating features (a hole for example) that are to be mated with corresponding features on other parts, or body features (a stock for example) to which other features are attached. Since every feature is an instance of some feature class the set of available feature classes in the system will constrain the description of a valid feature. The set of feature classes is in turn dependent on the application and will need to be defined by an application engineer.

A feature class might be viewed as a composite of two or more feature classes, for instance a hex-nut feature could be considered to be a hex-peg feature with a threaded-hole feature at its center. Our system places no limitations on the complexity of feature classes, the application engineer is free to make the features as simple or complex as he chooses. The level of complexity of a feature class is ultimately dependent on the user's ability to express the desired mating relationships in a given assembly. If saying that gear1 meshes with gear2 adequately describes the mating between them, then in this case a feature class gear is a reasonable means of describing gear1 and gear2. If, on the other hand, tooth3 of gear1 must be put into the space between tooth1 and tooth2 of gear2 then both gear1 and gear2 must be expressed in terms of gear teeth and spaces between gear teeth. While we allow the application engineer the flexibility of defining his own feature classes, we place upon him the burden of defining them in such a way that they are complete and consistent.

Internally, feature classes are created and manipulated via the *Common Lisp Object System* (CLOS) which supports an object-oriented style of programming. (See [29] for an excellent introduction to CLOS). A feature class is represented by an attribute-value CLOS data structure, and each individual feature belonging to a particular class is an *instance* of this class. CLOS enables us to build a class from another class; the original class is the superclass of the new class and the new class is a subclass of the original class. The new class inherits both structure and behavior from its superclasses. We use this mechanism to establish a hierarchy of feature classes. Every feature class has as a superclass the class *feature*. Any specific feature that cannot be defined in any other way can always be defined to be an instance of the class *feature* but, as we shall explain shortly, defining a feature in such a way is a laborious process.

An example of a data structure for the feature class *spur-gear* is illustrated in Fig. 3a; an actual instance of which is illustrated in Fig. 3b. The class *spur-gear* is a subclass of the class *gear* which in turn is a subclass of the class *feature*. All of the attributes in the top block of Fig. 3a and b are those that are inherited from the class *feature*.

The *sym*, *additivep* and *csg* attributes are automatically instantiated whenever an instance of the *spur-gear* subclass is created. The attribute *sym* described the symmetry group of the feature; we will explain symmetry groups later in this section. The attribute *csg* points to a constructive solid geometry (CSG) representation of a CAD model of the assembly feature. This representation is calculated by a method specialized for spur-gears that is automatically invoked upon instantiation of a spur-gear, every feature class must have a similar method associated with it. The value of *additivep* is false if the feature is a hole or a void and true otherwise. We will discuss the attributes *nom* and *mmc* shortly.

The attributes in the second block in Fig. 3a represent the attributes inherited from the *gear* feature class. The remaining block in the figure represent the attributes specific to the *spur-gear* feature class. In order to create an instance of a *spur-gear*, a user must specify all of the *gear* attributes and all of the *spur-gear* attributes.

Note that a user may specify a list for the value of certain attributes rather than a singular entity. This list represents a manufacturing tolerance on that attribute and contains the nominal

Fig 3: (a) This is the data structure for the feature class spur-gear. Note that the attributes *csg*, *additivep* and *sym* are calculated automatically by a method for the spur-gear class associated with the CLOS initialize-instance function. The attributes *nom* and *mmc* are calculated only if they are needed by a method for the feature superclass. (b) An instantiation of the data structure in (a). (c) Rendering of the nominal boundary representation automatically generated for the spur-gear instance in (b).

value and tolerance for that attribute. For instance in Fig. 3b the attribute *radius* is to be within the range 1.45 ± 0.05 . Of course, it makes no sense to specify a manufacturing tolerance on some attributes, for instance *num-teeth*.

The value of the attribute *nom* is a pointer to the nominal realization of the boundary representation (Brep) of the feature, that is, the Brep of the feature when all of the attributes take on their nominal value. The value of the attribute *mmc* points to the most-material-condition Brep of the feature. This Brep is created by enlarging all the convexities and shrinking all the concavities, the enlargements and shrinkages corresponding to the tolerances associated with the

various dimensions. * Both Breps are created simultaneously using the CSG attribute when and if they are needed. Fig. 3c shows a rendering of the Brep pointed to by the *nom* attribute.

We have created a large set of feature classes typically found on assembly parts, some of which are described in Table 1. Of course this set is by no means complete; for example, we have not currently defined a feature subclass for bevel gears. If we were to deal with assemblies involving bevel gears, it would be desirable to develop a *bevel-gear* feature class, although, of course, it would also be possible to create a specific bevel gear feature by using the feature class *feature*. However, this would become too cumbersome to do repeatedly because if a feature is created using the feature class *feature* the values of the attributes *csg*, *additivep* and *sym* must be completely specified by the instantiation.

Feature Class	Attributes to be Specified	Explanation
feature	csg, sym, additivep	Any feature
round-peg	radius, height, bevel radius and height	Helical peg with possible bevel
comp-round-peg	list of component radii and heights	A peg composed of cones and cylinders
arb-peg	csg, sym	Square, hexagonal, or any arbitrarily shaped peg
round-hole	radius, height, chamfer radius and height	Cylindrical hole with possible chamfer
threaded-hole	radius, height, chamfer radius and height	Threaded cylindrical hole.
threaded-cyl	radius, height, bevel radius and height,	Threaded cylinder as in part of a bolt
spur-gear	radius, height, num-teeth, in-width-teeth, out- width-teeth, depth-teeth	Typical spur gear

Table 1: Examples of some of the feature classes that have been implemented in our system.

* We realize that representing tolerance uncertainty is a research issue in itself that has not been fully resolved. However, a good uncertainty model for our purposes can be built in the terms of the most material condition (mmc) representation proposed by Requicha in [51].

3.2 Representing a Part

In general, given a set of features, a part may be obtained by constructing Boolean expressions over the features using union, difference, and intersection operators, the operators being used in the regularized sense discussed in [50]. The intersection operator is really not necessary, since equivalent results can be obtained by using just the union and difference operators. The fact that from a mathematical standpoint, it is not necessary to use the intersection operator is particularly fortunate for us since that operator is capable of destroying the semantics of the primitives in a feature-based representation. What is being said is that in a feature based representation, it is more natural to think of all parts as being an aggregation of the features; of course, in this aggregation some of the features may be subtractive, a process controlled in our system by the attribute *additivep* shown in the data structure of Fig. 3a.

Like a feature, a part is represented by the CLOS data structure *part*, an uninstantiated example of which is shown in Fig. 4. The part is considered to be an aggregation, which can have both additive and subtractive components, of the features in the list of lists that is the value of the attribute *feature-list*. Each list in this list of lists consists of at most 2 elements: the first element is a pointer to the instantiated data structure for the feature being used, and the second is the pose transform of the feature in the coordinate frame of the part. When a feature is subtractive it is assumed that the material composing this feature is to be subtracted from the feature preceding this feature in the *feature-list*. In this way the remaining features may be added back into the void created by this subtractive feature.

The instantiated example of a *part* data structure is shown in Fig. 4b; this example constructs the handle of a lid for an assembly to be shown later. The attributes *mmc*, and *nom* have the same meanings as for the case of features, their values in Fig. 4b are pointers to the Brep of the part in its most-material-condition, and its nominal state where just the nominal values for the dimensions of the features are used. Fig. 4c shows a rendering of the Brep pointed to by the attribute *nom* for this part.

Note that we do not at present compute the symmetry group of the part from the symmetries of the features, although that could be done by using the methods presented in [48, 61]. Also note that there are some attributes not listed in the figure that are used for storing values needed for various internal purposes.

3.3 Representing an Assembly

Our assembly representation can in essence be thought of as a binary tree. The leaves of the tree contain parts and each non-leaf node in the tree represents an assembly operation to be performed on the part or subassembly in the left subtree and the part or subassembly in the right subtree. We assume that it is the part or subassembly in the left subtree that will be moved to the part or subassembly in the right subtree and therefore refer to the left subtree as the “grasped part” and the right subtree as the “fixtured part.” An assembly operation is a task that involves connecting a grasped part with a fixtured part.

Fig 4: (a) This is the data structure for the class *part*. Note that the value of *csg* is created automatically by a method for the *part* class if it is needed. Similarly, the values of *nom* and *mmc* are calculated automatically for a second method for parts. (b) An instantiation of the data structure in (a). (c) Rendering of the nominal boundary representation automatically generated for the part instance in (b).

To represent an assembly operation in our system, the user must instantiate a CLOS data structure of the class *assembly*. An uninstantiated *assembly* data structure is depicted in Fig. 5a. The initial uncertainty in the position and orientation of the grasped part and the control uncertainty of the robot are specified by the attributes *pose-uncert* and *control-uncert*. These quantities will be discussed in detail in Section 6.

The attributes *initial-mate-pose* and *final-mate-pose* refer to the pose of the grasped part at the beginning and at the ending of the mating operation respectively. We will discuss the computation of *final-mate-pose* in Section 3.5. The value of *initial-mate-pose* will automatically be calculated if it is not specified the first time it is queried. Typically an assembly task planner will be used to formulate a grasp plan to pick up the grasped part and a coarse motion plan to bring it into position for the start of the mating operation. The assembly task planner may specify the value for *initial-mate-pose* or query the value to trigger a computation of a reasonable starting point for fine motion planning which corresponds to a good ending point for the coarse motion plan; this computation is based on the value of *final-mate-pose* and the mating

<pre>(assembly csg nom mmc initial-mate-pose final-mate-pose grasped-part fixtured-part pose-unc cont-unc description)</pre>	<pre>(assembly csg <ptrg> nom <ptrh> mmc <ptri> initial-mate-pose (3.8 3.7375 .5 0 0 0) final-mate-pose (3.8 3.7375 .125 0 0 0) grasped-part handle fixtured-part lidpt pose-unc (.05 .05 .01 0 0 0) cont-unc (.01 .01 .01 0 0 0) description ((p1 (into) h1) (p2 (into) h2)))</pre>
(a)	(b)

(c)

Fig 5: (a) This is the data structure for the class assembly. Note that the attributes in the first block can or will be calculated automatically by methods for the assembly class. (b) An instantiation of the data structure in (a). (c) Rendering of the nominal boundary representation automatically generated for the assembly instance in (b).

features involved.

A good ending point for a coarse motion plan, in general, is at the point where the fixtured part or subassembly can no longer be represented by a convex hull completely containing it. For instance, assume we are developing an assembly plan to mate a gear with some parts inside of a gear box. Since coarse motion planning must typically be carried out during an assembly operation, it would be desirable to model the gear box subassembly as a single parallelepiped completely containing the subassembly so that planning could be carried out as quickly as possible. Therefore a good ending point for coarse motion planning would be at a point prior to the gear

entering the box. While coarse motion planning must usually be performed on-line because the parts are not guaranteed to be in the same place each time, we can normally assume that the structure of the fixtured part will be the same each time within some bound of uncertainty. Therefore an assembly motion plan created off-line should be valid for each execution of the assembly mating operation, so a detailed fixtured part representation can and must be used.

The *description* attribute of an assembly operation is used to describe how the parts are to be mated. The value of this attribute is a list of lists where every list contains a feature on the grasped part, a list describing an attachment, and a feature on the fixtured part. The grasped feature is to be mated with the fixtured feature as specified by the attachment. The first element in the list describing the attachment is the attachment type. Examples of attachment types are into, onto, thread, and mesh. The remaining elements in this list are optional and will be explained shortly. Upon instantiation of an assembly, data structures are automatically created for each attachment. Similar to features, each attachment is an instance of some class. The different types of attachments correspond to various attachment classes.

The remaining elements in the attachment list are used to specify values for attributes of the attachment. In CLOS we can define generic functions that provide a generic interface for performing various functions on classes. When invoked, a generic function will in turn invoke the method associated with the class of its argument. For instance, we have defined a generic function to calculate the mating transformation from one feature to another when they are mated. The actual method used to calculate the mating transform depends on the type of attachment between the two mating features. For example, by definition when a gear meshes with another gear the transformation between them is described by a translation in the X-Y plane possibly followed by a rotation about the Z axis. On the other hand, when a peg is inserted into a hole the transformation between them defaults to the identity transform.

The default peg and hole mating operation corresponds to having the bottom faces of the peg and hole touching. If the user instead wants the peg to be inserted a certain distance through the hole he or she can specify the keyword *:offset* and this distance as the second and third element in the attachment list. This distance, if specified, is encoded into the *offset* attribute of the *into* data structure, otherwise the value of the *offset* attribute defaults to zero. The method for *into* used to calculate the mating transform will take the value of the *offset* attribute into account. If no additional parameters are specified in the attachment list, the defaults will be taken on all pertinent attributes. Note that each attachment type is a subclass of the class *attachment*. This allows us to define generic methods for attachments.

Associated with some special types of attachments is a method that specifies an additional motion or motions necessary to make the attachment. In a threading attachment, the system would rely on the motion planner to find a sequence of motions to bring one feature into threading position with respect to the other. However the actual threading motion would be symbolically calculated by a method associated with the thread class rather than planned by the motion planner because it is far easier to find the threading motion this way.

The values of the remaining attributes in the *assembly* data structure in Fig. 5a are automatically calculated if and when they are needed. The attributes *csg*, *mmc*, and *nom* are the same as those defined for a part and are calculated by the functions *assembly-csg*, *assembly-mmc* and *assembly-nom* given in Appendix 2. The *final-mate-pose* attribute refers to the position and orientation of the grasped part when assembled with the fixtured part and can be calculated using the procedure outlined in Section 3.5 which is referred to as *endpt-loc* in the source code listed in Appendix 2. Fig. 5b shows an instantiated *assembly* data structure. Fig. 5c depicts a rendering of the assembly described in Fig. 5b.

Our assembly representation is different from those developed by others before us, largely since the intended application of our representation is different. For example, Sanderson, Homme de Mello, and Zhang, [48] have advanced a representation based on relational graphs for representing assemblies in order to facilitate the generation of optimum assembly sequences. A search for optimum assembly sequences necessitates that the representation used contain no biases with regard to the ordering of parts, hence the representation used by Homme de Mello and Sanderson is a graph data structure. On the other hand, since the principal aim of our representation is the automatic generation of assembly motion plans, we assume that the sequence of assembly operations is already known and serves as one of the inputs for the generation of our representation and therefore our representation can be more structured. This is why we can use a tree structure, whereas Sanderson et al. must use a graph structure.

Another major point of difference between the representation used in [48] and our representation is that while the mating relationships in [48] are only considered at the surface level, we consider them to be on the feature level, where a feature can consist of any number of surfaces. The use of features makes it easy to capture in the representation important information about uncertainties, articulations, nonrigidities, etc., which would be difficult to do in a purely surface-type representation.

The representation used by [44] is geared towards analyzing geometrical symmetries for purposes such as the discovery of the mating features on the parts to be assembled, the determination of the symmetry of the assembled object from the symmetries of its parts, etc. While this representation, derived from group-theoretic considerations, is 'optimum' for its intended purposes, it is really not complete enough for assembly motion planning. However, given the importance of analyzing symmetries for mechanical assembly, we have embedded Popplestone et al.'s group-theoretic notions for describing parts in our representation. This makes it easier for us to calculate the destination pose of, say, a grasped part that will undergo a mating operation for assembly.

Our representation is somewhat similar to the one developed by Tilley [54]. Tilley's work concerns the automatic generation of assembly representations at design time, particularly when an assembly is created on a CAD workstation using graphical tools. In order for a user to specify an assembly operation with Tilley's system, the user must select two previously defined parts (or subassemblies) and input into the system the initial location and the assembled location of the part to be grasped, where these locations are relative to the stationary part. The

specification of these locations is facilitated by a graphical interface which also prompts the user for the type of operation required; e.g. fits, screw, etc., and an approach vector. Given this information Tilley's system automatically selects faces on each of the two parts and labels them as assembly features and then defines the spatial relationship between the two parts in terms of relationships between the assembly features, and also in terms of a 4×3 location matrix. The system then used the approach vector information to synthesize a motion plan for the assembly. However, such motion plans cannot always be used directly on a robot for a variety of reasons, such as when portions of the fixtured part impede motion along this vector or when the uncertainties involved are too large to ignore.

Our assembly representation differs from Tilley's with regard to (a) how the representation for a particular assembly is created; (b) the incorporation of various uncertainties; (c) automatic generation of assembly motion plans, as opposed to their being specified by the user; and (d) what it is that is considered to be an assembly feature -- our system uses volumetric primitives as assembly features, whereas in Tilley's system the contacting surfaces are called assembly features; etc. In our system, an assembly representation is not created by its graphical simulation, but by a declaration by the user of what volumetric primitives participate in assembly operations; such declarations can be made by the user at a purely symbolic level. Since our goal is the automatic creation of assembly motion plans, we cannot afford to ignore the various uncertainties that might be present in the system -- uncertainties regarding the geometry of the parts, as reflected by their tolerances, uncertainties in the initial poses of the parts, control uncertainties associated with the manipulator, etc.

The reader should note that, outside the context of assembly motion planning, much attention has recently been focussed on the use of volumetric primitives for feature based representations in part design, process planning, and visual part inspection systems [8, 24] and our work may be viewed as exploiting the power of these representations for assembly motion planning. Feature based representations are superior compared to the representations that use just geometry because features can be semantically significant. For example, in process planning we may have features such as slots and holes that correspond directly to machining operations.

3.4 An Example of Assembly Representation

To see how the user could create a representation of an assembly we will explain the process used to develop the assembly in Fig. 6. First the user must create an instantiation for each of the features making up each part. For instance, the features on the handle are created with the following three statements, presented for the sake of clarity in a pseudo-lisp form (those words beginning with a ':' are attribute names and are followed by their value).

```
bridge = make-instance(stock, :length 1.475, :width .25, :height .25))
```

```
peg1 = make-instance(round-peg, :radius .125, :height 1, :bevel-radius .125, :bevel-height 0)
```

```
peg2 = make-instance(round-peg, :radius .125, :height 1, :bevel-radius .125, :bevel-height 0)
```

Fig 6 The assembly tree showing how the subassembly on the left is assembled with the subassembly on the right. The leaves of the tree are parts.

Note that the user may specify a tolerance on say, the radius of the peg by stating *:radius (.125,.01)*. The representation for the handle part can be created with the statement:

```
handle = make-instance(part, :feature-list ((bridge, (0 0 .75 0 0 0)), (peg1, (0 -.7375 0 0 0 0)), (peg2, (0 .7375 0 0 0 0)))).
```

Similarly the lidpart is built with the following statements:

```

plate= make-instance(arbitrary-peg :partlist ((block, length=7.475, width=7.6, height=.25)))
hole1= make-instance(round-hole :radius .125, :height .25, :chamfer-radius .125, :chamfer-
height 0)
hole2= make-instance(round-hole :radius .125, :height .25, :chamfer-radius .125, :chamfer-
height 0)
lidpart= make-instance(part, :feature-list ((plate), (hole1, (3.8 3.0 0.125 0 0 0)), (hole2, (3.8
4.475 .125 0 0 0)))).

```

Note that if a transformation is not indicated for a feature, the identity transformation is assumed. The lid assembly then can be created by stating:

```

lid= make-instance(assembly, :grasped-part handle, :fixtured-part lidpart, :description ((peg1
(into) hole1), (peg2 (into) hole2))).

```

This creates an *assembly* data structure pointed to by the variable *lid*. The variables *hole1* and *hole2* point to *round-hole* data structures, *peg1* and *peg2* point to *round-peg* data structures. The variables *lidpart* and *handle* point to *part* data structures.

The right half of the tree can be created similarly by calling the *make-instance* function first for the features *gear1*, *hole3*, *housing*, *recess*, *opening*, *gear2*, and *shaft*, and then for the parts *gearpart* and *boxpart*, and finally for the assembly *gearbox*. The root node in the tree is created with the following statement:

```

gearbox= make-instance(assembly, :grasped-part lid, :fixtured-part box, :description ((plate
(into) opening))).

```

Many details of a real gear box assembly have been left out to make an assembly tree that could easily be depicted in a single figure.

While the above process might seem to be rather tedious, a user-interface could be easily developed to prompt the user for the required information. This would save the user from having to keep track of the syntax of the system and if the interface could take graphical input, the specification could be made even easier using methodology such as that presented in [54].

3.5 Calculation of the Mating Pose of the Grasped Part

One of the most useful aspects of our feature based assembly representation system is that it automatically calculates the position a grasped part must be brought to in order to be assembled with a fixtured part. To show how this is done we will first consider a two part assembly. Assume this assembly involves connecting part1 with part2 by mating feature1 of part1 with feature2 of part2. Then a valid mating transformation of part1 is simply:

$$T_{p1}^O = T_{p2}^O T_{f2}^p M(T_{f1}^p)^{-1} \quad (1)$$

where T_{p1}^O is the homogeneous transform defining the position of part1 with respect to the origin, T_{p2}^O is a similar transform for part2, T_{f2}^p is the homogeneous transform defining the position of feature2 with respect to part2, T_{f1}^p is the homogeneous transform defining the position of

feature1 on part1, and finally M is the transformation discussed earlier that relates the position of feature1 to that of feature2 when feature1 is attached to feature2 in the described manner. For those not conversant with homogeneous transforms see [28].

When the mating features on both parts possess symmetries, as is often the case, there will be mating transformations, other than the one obtained from Eq. (1), that will give the pose of part1 in the assembled object. Taking into account the symmetries, all the different possible poses for part1 can be found by calculating the following expression for all possible values of i

$$T(f1, i) = T_{p1}^0 T_{f1}^p R(f1, i) (T_{f1}^p)^{-1} \quad (2)$$

where $R(f1, i)$ is a member of the symmetry group of feature1. A symmetry group is defined as follows: let feature1 be the set of points S in R^3 , then the symmetry group of feature1 is the group of rigid transformations $\Sigma(S)$ defined by

$$\Sigma(S) = R \mid RS = S$$

where RS signifies premultiplication of the points in S by the transformation matrix R . Hence $R(f1, i)$ is used to denote the i^{th} member of the group $\Sigma(S)$ associated with feature1.

When the two parts to be assembled have only one mating feature each, it is pointless to compute the different transformations for part1 since all these poses would lead to equivalent results. If, however, it is desired that two or more features on part1 mate with their correspondents on part2, one must compute the set of feasible mating transformations for each pair of features on the parts and then find the intersection of the computed sets [44].*

For example, consider Fig. 7 and assume that feature1 of part1 must mate with feature2 of part2 and at the same time feature1' of part1 must mate with feature2' of part2. Mathematically, if a mating transformation is feasible for both feature1 and feature1' then

$$T(f1, i) = T(f1', j). \quad (3)$$

Here $T(f1, i)$ is the transformation found by first calculating T_{p1}^0 in Eq. (1) for feature1 and feature2 and then substituting this value into Eq. (2). Similarly $T(f1', j)$ is the transformation found by first calculating Eq. (1) with respect to feature1' and feature2' and then using the j^{th} member of the symmetry group of feature1', $R(f1', j)$, to calculate Eq. (2). If feature1 and feature1' have n symmetries each, then the complexity of computing the set of possible mating poses is on the order of n^2 . We will now discuss how the complexity can be reduced.

Combining Eq. (2) with Eq. (3) and rearranging we get

$$R(f1, i) = \left[T_{p1}^0 T_{f1}^p \right]^{-1} (T_{p1}^0)' T_{f1}^p R(f1', j) (T_{f1}^p)^{-1} T_{f1}^p \quad (4)$$

where $(T_{p1}^0)'$ is calculated with respect to feature1' and feature2', and T_{p1}^0 is calculated with respect to feature1 and feature2.

Substituting Eq. (1) into Eq. (4) where applicable and using properties of homogeneous transforms we can further simplify Eq. (4) to be

* Note that the mating transformation for a mesh attachment cannot be computed independently of other attachments, if there are any, because there is an infinite number of ways to mesh two gears, and it may be necessary to constrain the meshing transformation further. Therefore, the method that computes M in Eq. (1) for a mesh attachment automatically examines the other attachments specified.

Fig 8: (a) Four parts to be assembled. (b) One possible subassembly of pin1, pin2 and stock. (c) The subassembly of pin1, pin2 and stock when global constraints have been applied to feature symmetries.

and then the number of valid symmetries, in this case four.

These constraints are found as follows. After an entire assembly tree is created, a function is invoked to trace down the tree and find the first assembly operation at which two or more assembly feature pairs must be mated and the fixtured part features are not rigidly connected to the same part. It is for these type of mating operations that global symmetries must be examined. For these operations all of the valid mating positions are found for the assembly assuming the

grasped part is fixed and the fixtured part is free to move. All symmetries of each fixtured feature that yield a valid mating pose are found and recorded in the corresponding feature data structure. Then the symmetry groups of the other mating features on the part containing the fixtured feature under examination must be constrained as well in accordance with the fixtured mating feature's valid symmetries.

For the example in Fig. 8a, the valid symmetries for the head of **pin1** and the head of **pin2** are found. Then the part **pin1** is examined and the valid symmetries of the head feature of **pin1** are used to find the valid symmetries of the cylindrical peg feature on **pin1**. The valid symmetries of the cylindrical peg feature of **pin2** are similarly found. When the mating pose of **pin1** and **pin2** are found only the valid symmetries are considered. The results of mating **pin1** and **pin2** with **stock** by constraining the symmetries of **pin1** and **pin2** in such a manner is depicted in Fig. 8c. The function that calculates the constraints on the assembly features is referred to as *constraint-loc* in the source code listed in the Appendix 2.

An alternative approach would be to rely on backtracking to resolve the problem. After the entire assembly tree is created, the function for calculating the final mating position is called to find the mating position of the grasped part in the assembly. The *final-mating-position* function will be recursively called for the subassemblies of the fixtured part and of the grasped part. The *final-mating-position* function returns one possible position although, due to symmetries, other positions might be possible. Therefore if it is not possible to find a final mating position for the top level assembly, then the *final-mating-position* function could backtrack and select other configurations for subassemblies, until a feasible top level mating position is found.

4. ARTIFICIAL POTENTIAL FIELDS

In the previous section, we discussed a feature-based representation for parts and assemblies. The main advantage of that representation is that it allows humans to specify assemblies at a high level of description and it contains automatic mechanisms for inferring final mating poses and incorporating uncertainty. However, in and of itself this representation is not sufficient for assembly path and motion planning since it does not directly support any means for quick calculation of relative distances between parts. For path and motion planning purposes, as we will show in this section, one can take recourse to artificial potential fields that are easily derived from the geometric representations of the previous section.

The idea of artificial potential fields, as first promulgated by Khatib [27], consists essentially of associating repulsive potentials with obstacles and the moving object and then using the resulting artificial potential field distributions to determine the nearness of the object to the obstacles. The important phrase here is *artificial potential field distribution*. These potential field distributions are artificial in the sense that they do not necessarily satisfy the Laplace's equation everywhere, because they are not obtained by solving that equation subject to the appropriate boundary conditions. The only attribute they share with real potential field distributions is that the field decay for an obstacle is proportional to the inverse distance from the obstacle. This attribute suffices if the purpose is limited to the detection of collisions between an obstacle and

an object, since when there is a collision the potential of the object would go to infinity.

This inverse dependence on distance also implies that the potential at a point may be used as a measure of the nearness of that point to an obstacle and, given a set of obstacles, the valleys of the combined potential may be used to locate paths that avoid all the obstacles in some equal sense. Of course, in practice, the moving object will not be a single point, meaning that for collision avoidance we must consider both the position and the orientation of the object. Therefore, following the valleys of the potential field, while keeping the orientation fixed, may not suffice. The notion of finding valleys in a potential field must therefore be generalized to finding pathways that minimize a measure of potential integrated over the surface of the entire moving object, as was done by Hwang and Ahuja [21, 22].

Therefore, in the formalism of artificial potential fields, path planning for obstacle avoidance involves computation of minimum potential valleys for point objects and appropriate deviations thereof for extended objects. For on-line obstacle avoidance, it is of course not possible to pre-compute these valleys. However, in this case, especially for the case of a point object, local computation of the negative gradient of the potential at any location would point the way towards the nearest minimum potential valley and then track the valley itself to the desired destination, particularly so if a suitable attractive potential is associated with the destination [27]. As pointed out by Khatib himself, the problem with this approach is that the point object would get trapped in a local minimum, since there the value of the gradient will be zero.* Various approaches have been suggested for circumventing the problems posed by points such as local minima. As suggested by Rimón and Koditschek [47], for some geometries of obstacles, as for example when the obstacles can be considered to be a union of star-shaped regions, it is possible to define the potential function in such a manner so as to avoid local minima. Another solution to the problem, suggested by Barraquand and Latombe, lies in executing a series of random motions whenever one encounters zero gradients, with the hope that one of these motions will be a way out of the local minima [1].

On the other hand, getting trapped at a local minimum is less of a problem for off-line path planning with the artificial potential field formalism. As demonstrated by us here, and by other researchers before us [21, 22, 31], in most cases of practical interest it is possible to avoid getting trapped in local minima by employing some form of heuristic search.

In the rest of this section, we will first give our definition of the potential field function. This function, which describes the decay of artificial potential as a point object moves away from an obstacle, is different from those used by others before us. We believe that our definition, although slightly more cumbersome from a computational standpoint, is better suited for the calculation of assembly paths under tight clearances. Our definition requires that the minimum distance to an obstacle be calculated as precisely as possible. In contrast to some other computational procedures [21, 22], our definition allows us to compute potential fields for obstacles that

* It is interesting to note that a true potential field -- a field that satisfies Laplace's equation -- can only have a local minimum at a boundary. But then, as our discussion implies, artificial potential fields are not true potential fields.

are not strictly convex -- a frequent occurrence in assembly. We conclude this section with a brief survey of alternative approaches to path planning.

4.1 Potential Field Definitions

In AMP-CAD, the potential of a point in space represented by the position vector $\mathbf{x}$ is characterized by the following non-negative function, defined everywhere:

$$p(\mathbf{x}) = \frac{1}{\delta + \min_dist(\mathbf{x})} \quad (6)$$

where δ is a small positive constant, δ is included for computational reasons to keep the value of the potential bounded at the boundaries and in the interiors of the fixtured part and $\min_dist(\mathbf{x})$ is the smallest Euclidean distance between the point $\mathbf{x}$ and the fixtured part. As desired, when $\mathbf{x}$ is inside or on the surface of the fixtured part, $\min_dist(\mathbf{x})$ takes on a value of zero and therefore the potential function takes on a maximal value of δ^{-1} . As the point $\mathbf{x}$ moves away from the fixtured part, the value of $\min_dist(\mathbf{x})$ increases, and the potential function correspondingly decreases. From this point on we will refer to the value of the function $\min_dist(\mathbf{x})$ simply as $\min_dist$. We will describe an algorithm for calculating $\min_dist$ shortly. The reader is referred to [21, 27, 30, 55] for examples of other possible definitions for artificial potentials.

For an object to move collision-free, what is really of interest is the potential associated with the points occupied by the boundary of the object. Clearly, if any point on the object boundary is going to occupy the same space as any of the obstacles, the object potential should approach infinity as δ approaches zero. Therefore, the object potential is defined to be

$$P(\mathbf{X}) = \int_{\mathbf{x} \in B(\mathbf{X})} p(\mathbf{x}) d\mathbf{x} \quad (7)$$

where $\mathbf{X}$ is a six-vector representing the translation and orientation of the object, and $B(\mathbf{X})$ the set of points on the boundary of the object when the object is translated and rotated so as to occupy the pose corresponding to $\mathbf{X}$. $P(\mathbf{X})$ takes on its maximal value when there is a collision between the object and one or more obstacles, and decreases as the object moves away from the obstacles.* Unfortunately because $P(\mathbf{X})$ is dependent on both the shape of the obstacles and the shape of the moving object, in general an analytical form cannot be found for it. Therefore, it must be approximated using a method that will be described later on in this section.

Unfortunately, no matter how closely $P(\mathbf{X})$ is approximated, there will always be the possibility that its value might not become large enough to reliably detect a collision, especially if the point (or region) of contact between the two parts falls between the sampling points used for a digital computation of the integration above. Therefore, it becomes necessary to also calculate precisely the clearance between the object and the obstacles. For this reason we have defined a third function, we call the clearance function:

$$C(\mathbf{X}) = \min_{\mathbf{x} \in B(\mathbf{X})} [\min_dist(\mathbf{x})] \quad (8)$$

* It is important to note that, in the presence of a collision, the value of $P(\mathbf{X})$ is a measure of the "degree" of collision, meaning the extent of the intersection between the object and the obstacles. This statement is only true when δ is non-zero in Eq. (6), as will always be the case for computer calculations of $P(\mathbf{X})$.

Clearly, $C(X)=0$ when a collision between the object and obstacles takes place, and otherwise is equal to the smallest distance between a point on the object and the nearest obstacle. We have developed a method for computing $C(X)$ precisely so that, for instance, if it is necessary to know definitely that there exists no collision between the object in a given pose and the obstacles, we can calculate the value of $C(X)$.

It is necessary to be able to compute both $P(X)$ and $C(X)$. The gradient of the former yields the direction in which the object should be moved to get away from collision situations, and the value of the latter gives us a reliable indication of a collision. As shown in Fig. 9, the value of $C(X)$ will not change -- the value will be zero -- for incremental deviations of the object from the position shown. However, the value of $P(X)$ will change for incremental deviations.

Fig 9: If the object were perturbed slightly in any direction, from the position depicted, the value of $P(X)$ would change but the value of $C(X)$ would remain the same.

We will explain in greater detail how $C(X)$ can be computed precisely later on in this section after first discussing an algorithm for calculating the value of *min_dist*. Our discussion begins with a detailed explanation of the assumptions made by the *min_dist* algorithm. The interested reader can find routines for calculating *min_dist*, $p(x)$ (called pointpot), $P(X)$ (called objectpot), and $C(X)$ (called clearance) detailed in the path planning portion of the AMP-CAD source code given in Appendix 2.

4.1.1 Calculation of *min_dist*

How to calculate the minimum distance between two objects has been a problem of much interest during the last few years [5, 17, 35, 41]. However, all the solutions provided so far assume that the objects can be represented by polyhedral approximations; a further assumption is that nonconvex polyhedral objects can be expressed as a union of convex polyhedra. The polyhedral assumption makes such methods unsuitable for assembly motion and path planning because of the tight clearances involved. For example, the cylindrical surfaces of a 3 inch piston and a 3.001 inch cylinder would need to be represented by at least 122 facets to avoid any possibility of an erroneous collision being declared when the piston is precisely centered in the cylinder, to say nothing of how many facets would be needed if the piston was just slightly off-center but still not actually colliding. The need for such precision precludes the use of polyhedron-based methods.

In this section, we will therefore make a departure from these more traditional methods. Instead of expressing the fixtured part as a union of convex polyhedra we will assume that an arbitrary part can be expressed as a sum and difference of convex regions that are not necessarily polyhedral. Fig. 10 should make clear the distinction between the two approaches. Fig. 10a shows a block with a hole in it as a union of convex polyhedra -- as mentioned before, the number of such convex polyhedra will be very large for assembly under tight clearances. On the other hand, Fig. 10b shows the same block as a difference of two convex regions -- this is the decomposition on which our algorithms to be presented in this section are founded.

Fig 10: (a) A fixtured part represented as the union of convex polyhedral obstacles. (b) The same part represented precisely by the difference of two convex obstacles.

We will now list all the assumptions on which our calculation of $\min_dist$ is based and present more fully the assumption that the fixtured part be expressible as a union and difference of convex obstacles:

Assumption 1:

We assume that the fixtured part can be expressed in terms of union and difference operations on a set of convex obstacles. An obstacle whose material is added to the material of other obstacles is referred to as a positive obstacle and an obstacle whose material is subtracted from the material of a positive obstacle is referred to as a negative obstacle. Each negative obstacle is assumed to be contained within a single positive obstacle. Moreover, the negative obstacles contained in any given positive obstacle can overlap, but one negative obstacle cannot be completely contained within another. For example, the fixtured part shown on the left in Fig. 11a is the positive obstacle A, subtracted from which are two negative obstacles B and C. The ordering of the union and difference operations can be important. This is illustrated by the example in Fig. 11b, where if we first added the

positive obstacles C and D and then subtracted the negative obstacle B, the result would not be the gear box shown on the left. Fig. 11c shows an example of a part that does not lend itself to being easily expressed as a union and difference of convex obstacles.

Fig. 11: (a) This part can be considered to be a single positive convex obstacle containing two negative convex obstacles. (b) This gear box is made up of a positive convex obstacle (the box) containing a negative convex obstacle (the interior of the box) which in turn contains two positive convex obstacles (two gear shafts). The material of the negative obstacle B is subtracted from the material of the first positive obstacle A (c) An example of a part that could not easily be represented by a set of convex obstacles.

Assumption 2:

Given a decomposition of a fixtured part into positive and negative obstacles, it is assumed that the boundary representation (BRep) for each such obstacle is available. For example, for a block with a hole in it, we assume that the BReps are separately available for the convex block and the convex hole as shown in Fig. 12. (In contrast with the case of fixtured parts, we assume that for grasped parts we have available to us the BRep for the entire part. The reasons for this will be made clear later).

Assumption 3:

We assume that there exists a differentiable surface equation associated with each face of each obstacle in the boundary representation; the equations describe the infinite or closed surface in which the face is contained. By convention, the surface equation is such that if a point x is on the surface, $g(x) = 0$, and if x is on the material side of the surface $g(x) < 0$, otherwise $g(x) > 0$. In this way, as shown in [11] a point is inside an obstacle if $g(x) < 0$ for all faces on the obstacle.

Fig 12: A pictorial view of the BReps needed for the obstacles used to express the fixtured part depicted in Fig. 10. Face 3 through 6 are not shown for obstacle p.

Assumption 4:

As stated in Appendix 1, the problem of calculating the minimum distance from a point x to an infinite or closed surface described by the implicit equation $g(x)=0$ is really not that hard and can always be solved through techniques of nonlinear optimization. Usually, for a curved surface there will be more than one point on the surface whose distance from x will be locally minimum with respect to the neighboring points on the surface. (This point, which is not immediately intuitive, is illustrated in Appendix 1 with the help of a parabolic surface. For a point x shown there, there are three points on the surface whose distances from x are locally minimum with respect to their neighbors on the surface). The set of such points will be denoted by Y . We assume that available to us is an appropriate procedure, along the lines stated in Appendix 1, for finding Y for a given implicit surface equation $g(x)=0$.

A similar problem arises when we try to compute the minimum distance from a point x to

an infinite or closed space curve. There can be more than one point on the curve whose distance from x will be locally minimum. The set of these points will be denoted by Z . We assume, again in accordance with the discussion in Appendix 1, that we have available to us procedures for finding Z for a given space curve.

The set Y will be called the surface projection set and the set Z the curve projection set. Note that, for a given surface, each element of Y is a point on the surface such that a line joining the point with x is an orthogonal projection from x to the surface. The elements of the set Z may be interpreted in a similar manner.

We will now show, in procedures of increasing complexity, how $\min_dist$ is calculated. Four procedures used for this are summarized in Table 2. We will discuss in greater detail each of these procedures and justify their complexity measures.

Name of procedure	Complexity	Description of the function performed by this procedure
POS_OBSTACLE	$O(F^2)$	Compute minimum distance from x to a positive obstacle of F faces not containing any negative obstacles
SINGLE_NEG_OBSTACLE	$O(F^3)$	Compute minimum distance from x to a single negative obstacle (of F faces) contained in a positive obstacle.
NEG_OBSTACLES	$O(N \cdot F^3)$	Compute minimum distance from x to a member of a set Q of negative obstacles contained in a positive obstacle. Here N is the cardinality of Q and F is the maximum number of faces on any negative obstacle.
POS_OBS_CONTAINING_NEG_OBS	$O(N \cdot F^3)$	Compute distance from x to a positive obstacle p that contains a set Q of negative obstacles.

Table 2: Procedures needed for the computation of $\min_dist$.

PROCEDURE POS_OBSTACLE

This procedure is based on the rationale that given the implicit equations $g_1(x) = 0, g_2(x) = 0, \dots, g_F(x) = 0$ for the F surfaces on a positive convex obstacle, if

$$g_i(x) \leq 0 \text{ for } i=1, \dots, F \quad (9)$$

then the point x is inside or on the surface of the obstacle and the minimum distance to the obstacle is zero. This corresponds to the situation depicted in Fig. 13a. On the other hand, if there is only one face such that $g_a(x) > 0$, then we will have a situation depicted in Fig. 13b. In this event, in the general case the minimum distance to the obstacle is the minimum distance between x and a point in Y computed for this face using methods explained in Appendix 1. (See Assumption 4 for definition of Y .)

Fig 13: (a) In this situation $g_i(x)$ is less than zero for all faces so x is contained within the obstacle. (b) Here $g_a(x)$ is greater than zero, but for all other faces on the obstacle $g(x)$ is less than zero. Therefore the minimum distance is found with respect to the surface a . (c) In this case, $g_a(x)$ and $g_b(x)$ are both greater than zero, so that we need to first find the edge at the intersection of face a and face b , and then finding the minimum distance between this edge and x . (d) For this obstacle we have $g_a(x)$, $g_b(x)$, and $g_c(x)$ all greater than zero and the closest point to x is on the face b .

Finally, if there is a set of faces

$$\Gamma = \{i \mid g_i(x) > 0\}$$

then the relationship of x to the obstacle corresponds to one of the two depicted in Figs. 13c and 13d. For the case of Fig. 13c, the closest obstacle point to x is on the edge formed by the intersection of the faces a and b . And, for the case shown in Fig. 13d, the closest obstacle point is on the face labeled b .

To catch the condition depicted in Fig. 13c, we must find the edges shared by every pair of faces in Γ . For each such edge, we must then compute the set Z of points -- see Assumption 4 for definition of Z -- using methods also explained in Appendix 1. Now if a given point z in Z is not actually on the edge then we must replace z with the edge vertex to which it is closest. Recall that the points in Z are on the infinite or closed curve in which the edge is contained but not necessarily on the edge itself.

To deal with the situation shown in Fig. 13d, in addition to computing the sets Z for each pair of surfaces in Γ we must also compute the sets Y for each surface in Γ . We must then eliminate from all such Y 's the points that are not in the interior of the respective faces. A point y is in the interior of a face i if

$$g_j(y) < 0 \quad \forall j \in \Gamma \text{ \& } j \neq i$$

This necessary condition follows from the convexity assumption. Clearly, if a surface point is in the interior of a given face, then that point must be on the material side of surface equations for all the other faces. (Note that if there are no points left in Y after we have eliminated those points not in the interior, then we have a situation corresponding to that depicted in Fig. 13c.) From the remaining points in all the sets Y and Z , we now find that obstacle point which is closest to x . A pseudocode description of this procedure is as follows.

```

procedure POS_OBSTACLE(x)
begin
1.   $\Gamma \leftarrow \{ g_a(\mathbf{x}), g_b(\mathbf{x}), \dots \} \mid g_i(\mathbf{x}) > 0.$ 
2.  If  $(\Gamma = \emptyset)$  then return(0).
3.  If  $(\text{sizeof}(\Gamma) = 1)$  then
    begin
4.    Find Y for only face in  $\Gamma$ .
5.     $\mathbf{y}^* \leftarrow$  Closest point in Y to  $\mathbf{x}$ .
6.    return(euclidean_distance( $\mathbf{x}, \mathbf{y}^*$ ));
    end
7.  For each Face i in  $\Gamma$ 
8.    For each Face h  $\neq$  i in  $\Gamma$ 
    begin
9.      Find  $Z_e$  for non-null edge e at intersection of face h and i.
10.     If any point  $\mathbf{z}_{e,j}$  in  $Z_e$  not on edge e then, replace with closest vertex
    end
11.   $\mathbf{z}^* \leftarrow$  closest point to  $\mathbf{x}$  in any of the Z's.
12.  For each Face i in  $\Gamma$  find  $Y_i$ 
13.  For each face h  $\neq$  i in  $\Gamma$ 
14.    For each  $\mathbf{y}_{i,j}$  in  $Y_i$  found for face i
15.      If  $(g_h(\mathbf{y}_{i,j}) > 0)$  then, remove  $\mathbf{y}_{i,j}$  from  $Y_i$ .
16.   $\mathbf{y}^* \leftarrow$  closest point to  $\mathbf{x}$  in any of the Y's.
17.  return(min(euclidean_distance( $\mathbf{x}, \mathbf{y}^*$ ), euclidean_distance( $\mathbf{x}, \mathbf{z}^*$ )));
end

```

The main contributors to the time complexity of this procedure are the nested for-loops in lines 7-10 and the nested for-loops in lines 12-15. Since $\text{sizeof}(\Gamma)$ is bounded from above by $\bar{F}$, the number of faces on the obstacle, and since the computational effort in the two nested for-loops is additive, the overall complexity measure is $O(F^2)$. This measure is predicated on 1) our being able to compute the sets Y (see lines 4 and 14) and Z (see line 9) in constant time; and 2) the sizes of the set Y (see line 15) and of set Z (see line 10) being of order unity. Clearly, the effort required to compute Y for a given surface equation $g(\mathbf{x})$ is independent of the other surface equations. So we may upper-bound this effort by a constant, the size of the constant depending on the analytical form of the function g . Most of the assembly surfaces we have dealt with are first and second order surfaces and the computational effort to extract Y for such surfaces would contribute a small factor $O(1)$. The same argument applies regarding the computation of the set Z. The size of the set Y depends on the form of the surface involved. As our example in Appendix 1 illustrates, for second-order surfaces the size of Y will never be greater than three.*

* Note that it is possible for the set Y to be infinite for certain types of surfaces such as cylinders and spheres. In these situations Y consists of the points on a curve, or in the extreme case, Y consists of the points on the surface of a sphere when $\mathbf{x}$ is at the exact center of the sphere. When this occurs the entire set Y should be replaced by a single arbitrary point from the set. Since the distance to $\mathbf{x}$ does not decrease anywhere in the neighborhood of this point and since there are no points on the surface that are closer than this point (the distance to $\mathbf{x}$ is the same for every point in the original set) the computations to be presented will not fail to find the true distance to the obstacle. The same holds true for circular curves where the set Z could feasibly contain every point on the curve. Of course, the routines that calculate Y and Z should have some special cases set up to handle these situations so that their complexity is bounded.

Given that for this procedure a surface is constrained to be not undulating -- since otherwise the assumption of convexity would be violated -- we believe that the size of Y would not exceed a small number $O(1)$. The same conclusion can be extended to the size of Z .

PROCEDURE SINGLE_NEG_OBSTACLE

This procedure will find the minimum distance between x and a negative convex obstacle q contained within a positive obstacle p . Although it may seem that a negative convex obstacle is a region of space in much the same way as a positive convex obstacle and that therefore the algorithm for finding the minimum distance should be the same as described before, there are two important differences: 1) A necessary condition for a point to be a collision point with respect to a negative obstacle is that the point lie on one of the faces, therefore points inside a negative obstacle will not be collision points; and 2) Since some points on a negative obstacle may not correspond to points on the part, such points should not enter minimum distance calculations. To illustrate, note that none of the points shown in the three cases in Fig. 14 is a collision point with regard to the negative obstacle defined by the faces 1-4. Moreover, Fig. 14b shows a situation where, with regard to the negative obstacle, the closest point to x , on face 2, is not valid and should not be considered; the next closest point to x is on face 4 and that then defines the minimum distance in this case.

Fig 14: Shown here is a fixtured part with a cavity. The part may be considered to be a positive obstacle containing a negative obstacle, the latter defined by the faces 1-4. With regard to the negative obstacle, the minimum distance from x in (a) is to some point on the edge formed by the intersection of the faces 1 and 4; in (b) the minimum distance is to face 4 because the projection on face 2 would be invalid; in (c) the minimum distance is to face 3.

This procedure starts with the reasoning that the minimum distance from x to q is the minimum of the distances to any of the faces of q . So we first find the surface projection set Y_i for each face i of q . Let $y_{i,v}$ denote an element of Y_i . Every point $y_{i,v}$ in Y_i is examined for the following two conditions:

Condition 1:

This condition requires that $y_{i,v}$ be contained within the interior of the corresponding face i . If this is not true the point $y_{i,v}$ is replaced with a point that is on the boundary of face i .

Condition 2:

This condition requires that $y_{i,v}$ be a valid point on the part. By valid point we mean that it is a point on the surface of the positive obstacle p once the material corresponding to the

negative obstacle q is removed. If $y_{i,v}$ is not valid it is removed from Y_i .

To test a surface projection point $y_{i,v}$ for Condition 1, we first construct a set $\Gamma_{i,v}$ as follows:

$$\Gamma_{i,v} = \{j \mid g_j(y_{i,v}) \geq 0, \text{ face } j \text{ neighbors face } i\}$$

For instance in Fig. 15a for the point $y_{1,1}$ on face 1, $\Gamma_{1,v} = \{2\}$. This is because $y_{1,1}$ is on the non-material side of the surface containing face 2, implying $g_2(y_{1,1}) > 0$. In contrast, for the point $y_{1,1}$ in Fig. 15b, given that the neighboring faces of face 1 are face 2 and face 4, $g_4(y_{1,1}) < 0$ and $g_2(y_{1,1}) < 0$, so $\Gamma_{1,v} = \emptyset$.

Fig 15: (a) The point $y_{1,1}$ is on the exterior side of face 2 and therefore is not contained within the bounds of face 1. (b) The point $y_{1,1}$ is on the interior side of all faces neighboring face 1.

If $\Gamma_{i,v} = \emptyset$ then Condition 1 is satisfied. Otherwise we would like to find a point on the boundary of the face i to replace $y_{i,v}$. To replace $y_{i,v}$, for each face j in $\Gamma_{i,v}$, we find the edge $e_{i,j}$ formed by the intersection of face i with face j and perform the following operations. If $g_j(y_{i,v}) > 0$, we first find the curve projection set $Z_{i,v,j}$ for the edge $e_{i,j}$, otherwise when $g_j(y_{i,v}) = 0$ $y_{i,v}$ is already a boundary point so we set $Z_{i,v,j} = \{y_{i,v}\}$. Now since not every point in $Z_{i,v,j}$ is necessarily on the edge (recall that the points in $Z_{i,v,j}$ can be anywhere on the curve containing the edge $e_{i,j}$ formed by the surfaces i and j), we must replace any points in $Z_{i,v,j}$ that are not on the edge with the nearest vertex of edge $e_{i,j}$. Of course, not every point in $Z_{i,v,j}$ will be valid. For example, Fig. 16a shows a point z that is on the edge of the negative obstacle. But since this point is also on the edge of the positive obstacle, it is not a real point on the part. On the other hand, the point z shown in Fig. 16b is also on the edge of a negative obstacle. But, not being at the same time on the edge of the positive obstacle, that point is valid. By the same token, if a point z in $Z_{i,v,j}$ is a vertex of the edge $e_{i,j}$, it will be invalid if it is also a vertex on obstacle p . Therefore, we must next remove all such invalid points from $Z_{i,v,j}$.

For a given $y_{i,v}$, once we have removed all of the invalid points from the set $Z_{i,v,j}$ and we have considered each face in $\Gamma_{i,v}$, we then replace the point $y_{i,v}$ in Y_i by the closest point to x in any of the curve projection sets, meaning we select the closest point from $Z_{i,v,j}$ taking into account all possible indices j .

If Condition 1 is not true for $y_{i,v}$, then the point $y_{i,v}$ will necessarily have been replaced with an edge point that is valid, so there is no need to test Condition 2 for this point. Otherwise, to test Condition 2 when Condition 1 is true, we need to test every surface of p and if $y_{i,v}$ is on

Fig 16: (a) The edge point z on the negative obstacle is also an edge point on the positive obstacle and is invalid. (b) The edge point z on the negative obstacle is not on an edge of the positive obstacle so it is valid. (c) The face point y on the negative obstacle is also a face point on the positive obstacle and is invalid. (d) The face point y on the negative obstacle is not on a face of the positive obstacle so it is valid.

any of the surfaces it is invalid. In the event that $y_{i,v}$ is invalid we discard it from the set Y_i . An invalid point y and a valid point y are depicted in Fig. 16c and d respectively; in both these cases y passes Condition 1 and is therefore interior to the face i . A pseudocode implementation of this procedure is given here.

```

procedure SINGLE_NEG_OBSTACLE(p,q,x)
begin
1.  For each face  $i$  on  $q$ 
2.     $(dist_i, y_i^*) \leftarrow \text{DIST\_TO\_FACE}(p,q,x,i)$ .
3.  Return smallest  $dist_i$  and associated  $y_i^*$ .
end

procedure DIST_TO_FACE(p,q,x,i)
begin
1.  For each point in  $y_{i,v}$  in  $Y_i$  Comment-  $Y_i$  is surface projection set for face  $i$ 
begin
2.    For each face  $j$  that is a neighbor of  $i$  Comment- Test for Condition 1.
begin
3.    If  $g_j(y_{i,v}) \geq 0$  then
begin
4.    Condition 1 is false.
5.    If  $g_j(y_{i,v}) > 0$  then
6.     $Z_{i,v,j} \leftarrow$  curve projection set for edge  $e_{i,j}$ .
7.    Else  $Z_{i,v,j} \leftarrow \{y_{i,v}\}$ .
8.    For each  $z$  in  $Z_{i,v,j}$ 
begin
9.    If  $z$  not on edge  $e_{i,j}$ , then  $z \leftarrow$  nearest edge vertex.
10.   If  $z$  is a vertex of  $e_{i,j}$  then
11.   For each vertex  $w$  of  $p$ 
12.   If  $z=w$   $z$  is invalid, then discard  $z$  from  $Z_{i,v,j}$ .
13.   Else for each edge  $e'$  of  $p$ 
14.   If  $g_{e'}(z) = 0$ , then discard  $z$  from  $Z_{i,v,j}$ .
end
end
end
end
end

```

```

15.    $y_{i,v} \leftarrow$  closest point to  $x$  in any of the  $Z$ 's.
16.   If Condition 1 is true then    Comment- Test for Condition 2.
17.     For each face  $j$  of  $p$ 
18.       If  $g_j(y_{i,v}) = 0$  then  $y_{i,v}$  is on a face of  $p$  and invalid so discard  $y_{i,v}$ .
       end
19.    $y^* \leftarrow$  closest point to  $x$  in  $Y_i$ .
20.   return(euclidean_distance( $x, y^*$ ),  $y^*$ );
end

```

From the implementation of DIST_TO_FACE we can see that its complexity is dependent on the test for Condition 1 and the test for Condition 2. In assessing the complexity we will assume that F is the maximum of the following: the number of vertices on p , the number of edges on p , the number of faces on p , and the number of faces on q . The complexity of testing for Condition 2 then is $O(F)$, this complexity being derived from the for-loop executed on lines 17-18 of the procedure. The complexity of testing for Condition 1 is $O(F^2)$ due to the nested for-loops on lines 2-14. Since the complexity of Condition 1 clearly dominates, the complexity of DIST_TO_FACE is $O(F^2)$. Therefore, because the SINGLE_NEG_OBSTACLE procedure must call the DIST_TO_FACE procedure for each face on the obstacle q , its complexity is $O(F^3)$.

PROCEDURE NEG_OBSTACLES

When there is a set Q of negative obstacles contained in a positive obstacle p , the test for Condition 1 stated in the previous procedure remains unchanged. However, the test for Condition 2 is now slightly more complex due the fact that a surface projection point $y_{i,v}$ on a surface of q might be invalid if it is contained in the boundary or the interior of another negative obstacle in Q . This is the case for the point labeled $y_{i,v}$ in Fig. 17a. To determine the validity of a surface projection point $y_{i,v}$, in addition to the testing the surfaces of obstacle p as explained above, we perform the following operations. First we form the set of obstacles R which consists of any obstacles from the set $Q-q$ that contain the point $y_{i,v}$ on the boundary or within their interior. Note that R will be empty if the intersection of q with any of the other obstacles in Q is null.

We next eliminate from R those negative obstacles whose interaction with q does not invalidate $y_{i,v}$. In Fig. 17b, we have shown two negative obstacles with a portion of their bottom faces in common. If suppose a point y is being considered in connection with an obstacle q , this point will not be rendered invalid by the presence of the obstacle r . We therefore remove from R any obstacle r that meets the following condition: $y_{i,v}$ is on a face j of r (i.e. $g_j(y_{i,v})=0$) and the point x is on the same side of face i and face j (i.e. either $g_i(x)<0$ and $g_j(x)<0$ or $g_i(x)>0$ and $g_j(x)>0$)

As before, if $y_{i,v}$ is on a surface of p it is invalid. Otherwise $y_{i,v}$ is valid when $R = \emptyset$ and is invalid when $R \neq \emptyset$. When $R \neq \emptyset$, $y_{i,v}$ must be replaced with a valid point, to do this we recursively invoke the procedure NEG_OBSTACLES for each obstacle in R and replace $y_{i,v}$ with the closest point to x found on any of the obstacles in R .

Fig 17: (a) In this situation, the point marked " $y_{i,v}$ " is the nearest point to x on the face i , but $y_{i,v}$ is not a valid point. Therefore the distance to the face i must be found by recursively calculating the distance to the negative obstacle r . Unless care is taken, this call to r can recursive call q again, leading to an endless recursion. (b) Shown here are two negative obstacles, q and r , sharing a portion of one of their faces.

A situation might arise during the execution of the NEG_OBSTACLES procedure that would lead to an endless recursion unless it is trapped and the recursion is halted. This situation occurs when we make a call to NEG_OBSTACLES for an obstacle q , this makes a recursive call to NEG_OBSTACLES for an obstacle r , and this makes a recursive call to NEG_OBSTACLES for the obstacle q again. For example, this is the case for the two negative obstacles, both spheres, depicted in Fig. 18, where we seek to find the closest point to x on the obstacle q . This closest point, marked $y_{i,1}$, will turn out to be invalid because of the presence of the interfering obstacle r . Therefore, the procedure NEG_OBSTACLES will be called recursively with arguments x and obstacle r . This call will generate a surface projection set $\{y_{j,1}, y_{j,2}\}$, as shown in the figure. Now, since, the point $y_{j,1}$ is itself invalidated by the presence of q , one would want to make a recursive call to NEG_OBSTACLES with arguments x and q ; however, that would lead to an endless recursion, from q to r , and then back from r to q . To get around this difficulty, rather than call NEG_OBSTACLES again, we find the curve projection set $Z_{i,j}$ for the edge $e_{i,j}$ that is formed by the intersection of faces i and j . We then replace $y_{j,1}$ by the closest point to x from the set $Z_{i,j}$; let's denote the replacement point by $y_{j,1'}$. Now our last call to NEG_OBSTACLES, which had obstacle r as one of its arguments, will return the closest point to x from the set $\{y_{j,1'}, y_{j,2}\}$.

The pseudocode implementation of NEG_OBSTACLES is similar to the implementation of SINGLE_NEG_OBSTACLE only it invokes a different DIST_TO_FACE procedure, the pseudocode for which is given here.

Fig 18: In this situation the point $y_{j,w}$ is invalid and must be replaced with the point closest to x on the edge $e_{i,j}$.

```

procedure DIST_TO_FACE(p,q,x,i,last,Q)
begin
1.  For each point in  $y_{i,j}$  in  $Y_i$  Comment  $Y_i$  is the surface projection set for face i.
    begin
2.    Test for Condition 1 as before.
3.     $R \leftarrow \text{valid\_point}(y_{i,v}, p, q, Q, \text{edge\_point})$ ;
       Comment  $R = \text{True}$  if  $y_{i,v}$  is valid,  $p$  if  $y_{i,v}$  is invalid & on  $p$ , or the set  $R$  as defined.
4.    For ( $r \in R$ )
5.      If ( $r \in \text{last}$ ) then Comment Trap endless recursion.
        begin
6.          Find  $Z_e$  for the edge  $e$  at the intersection of face  $h$  and  $i$ .
7.          Replace points in  $Z_e$  not on edge with closest vertex.
8.           $y_{i,v} \leftarrow$  closest point to  $x$  in  $Z_e$ .
        end
9.      Else if ( $r = p$ ), then discard  $y_{i,v}$  from  $Y_i$ .
10.     Else If closest point to  $x$  on  $r$  has already been found, then  $y_{i,v} \leftarrow$  closest point on  $r$ .
11.     Else  $\text{dist}, y_{i,v} \leftarrow \text{NEG\_OBSTACLE}(p, r, x, \text{append}(\text{last}, q), Q)$ .
    end
12.   $y_{i,v}^* \leftarrow$  closest point to  $x$  left in  $Y_i$ .
13.  return( $\text{euclidean\_distance}(x, y_{i,v}^*, y_{i,v}^*)$ );
end

```

Note that the calling parameter *last* in the DIST_TO_FACE procedure is a list that is initially set to $\emptyset$ and is used to avoid an endless recursion. When NEG_OBSTACLES is recursively called, the current obstacle under consideration is appended to *last* so that this list contains a history of the obstacles in Q that have already been examined.

To find the complexity of NEG_OBSTACLES let us first assume no recursive calls need to be made. In this case the complexity is F times the complexity of DIST_TO_FACE, which is again dominated by the test for Condition 1 (called for on line 2 of DIST_TO_FACE), which was previously shown to be $O(F^2)$, making the complexity of NEG_OBSTACLES $O(F^3)$. Here we assume that F is the maximum of the following: the number of vertices on p , the

number of edges on p , the number of faces on p , and the number of faces on any obstacle in Q .

Now to account for the recursion, we need to note that it should never be necessary to invoke NEG_OBSTACLES on a member q of Q more than once because after the distance and closest point to x have been calculated for q the first time, they can be stored in obstacle q 's data structure and retrieved when necessary. Therefore, the depth of recursion caused by the call in line 11 will never exceed the cardinality of the set Q , which we will denote N . Moreover, note that since there is no backtracking allowed in the recursion and that since NEG_OBSTACLES might in the worst case be recursively invoked $N-1$ times, the overall complexity of the recursive NEG_OBSTACLES is $O(N \times F^3)$.

PROCEDURE POS_OBS_CONTAINING_NEG_OBS

The procedure for calculating the distance to a positive obstacle p containing a set Q of negative obstacles is similar to the procedure POS_OBSTACLE explained at the beginning of this section with the following exception. When the closest point to x with respect to p is on the boundary or in the interior of a negative obstacle q in Q , the point is invalid and we must replace it with the closest valid point to x with respect to q . This replacement point can be found using the NEG_OBSTACLES procedure.

To show how this is incorporated into the POS_OBS_CONTAINING_NEG_OBS procedure note that if

$$\forall_{j \in \text{Faces}(p)} g_j(x) \leq 0 \quad (10)$$

and

$$\neg \left[\exists_{q \in Q} \forall_{j \in \text{Faces}(q)} g_j(x) \leq 0 \right] \quad (11)$$

then the point x is inside or on the boundary of p but is not inside or on the boundary of any of the negative obstacles in Q , so the distance to p is zero. If, on the other hand, only Eq. (10) is satisfied then x is inside or on the boundary of some obstacle q , so we must find the distance to x with respect to the negative obstacle q by invoking the NEG_OBSTACLES procedure. Clearly, such a call will automatically take care of any interactions between the various negative obstacles in Q .

The only other possibility is that both Eqs. (10) and (11) are not true. This means that the point x is neither in the positive obstacle p nor in any of the negative obstacles in Q . In this case, the distance to the boundary of the positive obstacle p is found as we have shown already at the beginning of this section, except that, when examining a point in a surface projection set or an edge projection set of p , we must check to see if this point is valid. As stated before, such a point will not be valid if it is on the boundary or in the interior of a negative obstacle q in Q . If the point is invalid, we must replace it with the point returned from a call to the NEG_OBSTACLES procedure with arguments x and q .

A pseudocode implementation for this procedure is given here.

```

procedure POS_OBS_CONTAINING_NEG_OBS(p,Q,x)
begin
1.   $\Gamma \leftarrow \{ g_a(x), g_b(x), \dots \} \mid g_i(x) > 0 \text{ for obstacle } p.$ 
2.  For (q  $\in$  Q) do
3.    If x inside obstacle q, then  $q' \leftarrow q.$ 
4.    If (( $\Gamma = \emptyset$ ) & ( $q' = \emptyset$ )) then return(0).
5.    If (( $\Gamma = \emptyset$ ) & ( $q' \neq \emptyset$ )) then return(NEG_OBSTACLES(p,q',Q,x, $\emptyset$ )).
6.    return(POS_OBSTACLE(p,Q,x, $\Gamma$ )).
end

```

The procedure POS_OBSTACLE is similar to the one given already except for the small modification described. A new implementation is given below.

```

procedure POS_OBSTACLE(p,Q,x, $\Gamma$ )
begin
1.  If (sizeof( $\Gamma$ ) = 1) then
2.    begin
3.      For each point  $y_v$  in Y found for only face in  $\Gamma$ 
4.        If (( $q \leftarrow \text{valid\_point}(Q,y_v)$ )  $\neq$  True) then
5.          If closest point to q has been found, then  $y_v \leftarrow$  closest point to q.
6.          Else  $y_v \leftarrow$  NEG_OBSTACLES(p,Q,x,q, $\emptyset$ ).
7.           $y_v^* \leftarrow$  closest point to x in Y.
8.          return(euclidean_distance(x, $y_v^*$ )).
9.        end
10.   For each face i in  $\Gamma$ 
11.     For each face h  $\neq$  i in  $\Gamma$ 
12.       begin
13.         Find  $Z_e$  for non-null edge e at intersection of face h and i.
14.         If any point  $z_{e,j}$  in  $Z_e$  not on edge e, then replace with closest vertex.
15.         If (( $q \leftarrow \text{valid\_point}(Q,z_{e,j})$ )  $\neq$  True), then for any  $z_{e,j}$  in  $Z_e$ .
16.         If closest point to q has been found, then  $z_{e,j} \leftarrow$  closest point to q.
17.         Else  $z_{e,j} \leftarrow$  NEG_OBSTACLES(p,Q,x,q, $\emptyset$ ).
18.       end
19.        $z^* \leftarrow$  closest point to x in any of the Z's.
20.     For each face i in  $\Gamma$ 
21.       For each face j  $\neq$  i in  $\Gamma_{i,v}$ 
22.         For each  $y_{i,v}$  in  $Y_i$  Comment  $Y_i$  is the surface projection set for face i.
23.         begin
24.           If ( $g_j(y_{i,v}) > 0$ ), then remove  $y_{i,v}$  from  $Y_i$ .
25.           If (( $q \leftarrow \text{valid\_point}(Q,y_{i,v})$ )  $\neq$  True) then
26.             If closest point to q has been found then  $y_{i,v} \leftarrow$  closest point to q.
27.              $y_{i,v} \leftarrow$  NEG_OBSTACLES(p,Q,x,q, $\emptyset$ ).
28.           end
29.          $y^* \leftarrow$  closest point to x in any of the Y's.
30.       return(min(euclidean_distance(x, $y^*$ ), euclidean_distance(x, $z^*$ ))).
31.     end
end

```

Here the procedure for deciding if a point is valid or not is similar to the procedure valid_point used before only we do not need to consider p but we do need to consider every obstacle in Q.

Now the complexity of the procedure POS_OBS_CONTAINING_NEG_OBS will clearly be dominated by either the call to NEG_OBSTACLES in line 5 or the call to POS_OBSTACLE in line 6. As shown before, the complexity of NEG_OBSTACLES is $O(N \times F^3)$. The complexity of POS_OBSTACLE is dominated by the nested for-loops in lines 8-14 or 15-21. If we ignore the call to the NEG_OBSTACLES procedure then the complexity of either of these loops would be $O(N \times F^3)$ where the definitions of F and N were given previously and where $O(N \times F)$ is the complexity of the valid_point procedure invoked on either line 12 or line 19. To see that this is the complexity for the valid_point procedure, note that this procedure must examine every face on every obstacle in Q to decide if a point is valid or not.

It would seem that the complexity of the computations in lines 8-14 or 15-21 would be F^2 times the complexity of the procedure NEG_OBSTACLES, which has already been shown to be $O(N \times F^3)$. However, for reasons identical to those mentioned in the derivation of the time complexity function for NEG_OBSTACLES procedure, when the distance from x to any negative obstacle q is computed, it is stored for later use if necessary via the "then" clauses in lines 13 or 20. Therefore, it is not possible to really make F^2 invocations of the NEG_OBSTACLES procedure and the total computational effort for all the NEG_OBSTACLES will not exceed $O(N \times F^3)$. Therefore the overall complexity of POS_OBSTACLE is $O(N \times F^3)$ and the overall complexity of the POS_OBS_CONTAINING_NEG_OBS procedure is also $O(N \times F^3)$. Of course, if Q is null the complexity of the POS_OBS_CONTAINING_NEG_OBS procedure would be the complexity of the POS_OBSTACLE procedure which was shown to be $O(F^2)$.

The computation of *min_dist* entails invoking POS_OBS_CONTAINING_NEG_OBS for each positive obstacle and taking the minimum value returned from all calls. Hence the overall complexity of computing *min_dist* is $O(M \times N \times F^3)$ where M is the total number of positive obstacles.

4.1.2 Potential and Clearance of the Grasped Part

In an assembly, the grasped part is the object that will be moved during the mating operation and therefore the functions defined in Eqs. (7) and (8), $P(X)$ and $C(X)$, must be computed for the grasped part. In order to approximate $P(X)$ and to compute $C(X)$, we first find a polyhedral approximation of the boundary of the grasped part. To compute $C(X)$ precisely, we assume that we also have a parametric surface equation for each face of the grasped part, this equation describing the infinite or closed surface containing the face. As we will show in this section, the polyhedral approximation, while yielding directly an estimate for $P(X)$, yields a set of regions on the boundary of the grasped part that must be searched for the exact value of $C(X)$.

In AMP-CAD, a boundary representation of the grasped part is created by performing a boundary evaluation [50] on a constructive solid geometry (CSG) representation of the part; as the reader will recall a CSG representation forms a tree with the leaves of the tree being CSG shape primitives and the nodes of the tree denoting regularized Boolean operations. To create a boundary representation of a polyhedral approximation of the grasped part, we perform a boundary evaluation on a similar tree where the CSG primitives have been replaced by their

polyhedral approximations.

Of course, if the CSG primitive is polyhedral, there will be no reason to approximate it. If the CSG primitive is a conic, there are well known methods for approximating its shape with a faceted polyhedra. When the primitive is a NURBS surface or some other type of spline surface, we can form a planar faceted approximation of the surface by finding a set of sampling points on the surface and then connecting the sampling points to form planar facets. A set of sampling points can be found by sampling the surface in parameter space, by this we mean that if $r(u, v)$ is the parametric equation of the surface, then we can find sampling points on the surface by evaluating $r(u, v)$ for different values of the parameters u and v . The reader is referred to [19] for a further discussion of NURBS and spline surfaces.

To make easier the calculations of $P(X)$ and $C(X)$ using polyhedral representations, the following two-step procedure can be used. First, we can subdivide any planar facet that has more than four vertices by connecting non-adjacent vertices on the facet with edges to form two new facets in the place of the one. Next, we can make sure that the shortest distance, as computed on the actual boundary of the grasped part, between any two adjacent vertices of the approximating polyhedron is less than some constant ϵ . (We will explain the choice of ϵ shortly.) Whether or not two adjacent vertices satisfy this condition can be tested by computing the length of the shortest curve segment joining the two vertices along the boundary of the primitive. Methods for computing this length, which involve calculating the local curvature on the boundary of the primitive, are given in [18]. When the length of the curve is greater than ϵ , the two vertices are spaced too far apart on the boundary of the primitive, so we can add a new vertex between them that is also on the boundary of the primitive and define two new planar facets, each of which includes the new vertex, in place of the old planar facet.

In the current implementation of AMP-CAD, we assume that we already have a sufficiently dense polyhedral approximation to grasped parts. For simple shape primitives such as cylinders, spheres, etc., AMP-CAD has the capability to generate polyhedral approximations at any desired level of accuracy by using well known parametrizations of the surfaces. For a cylindrical surface, AMP-CAD uses the discretization of the usual (z, θ) parametrization; and for a spherical surface, tessellations are generated by geodesic divisions of the faces of regular solids like icosahedron and dodecahedron (more on this in Section 5). Given a polyhedral approximation to the grasped part, $P(X)$ is estimated by evaluating the following equation

$$\tilde{P}(X) = \sum_{i=1}^{i=no. \text{ facets}} p(x_i) s_i$$

where x_i is the center point on each facet, and s_i is the area of each facet.

For the purposes of calculating the value of $C(X)$, we define η to be the maximum distance between any point on the boundary of the grasped part and the nearest vertex of the approximating polyhedron, the distance being the shortest distance along the boundary of the grasped part. To see how η is calculated, consider a planar facet on a planar portion of the boundary of the grasped part. Now the farthest any point on this facet can be from one of the vertices is $\epsilon/\sqrt{2}$, and this occurs in a square planar facet at the center point of the square (see Fig. 19a). Now to

see what happens for the same square planar facet when it is used to approximate a curved portion of the boundary of the grasped part, imagine this square planar facet is flexible and we stretched the surface to create a bump in the center of the square so that it fits an arbitrary portion of the boundary precisely (see Fig. 19b). Now the distance from the center point of the stretched square to any of the vertices is based on the local curvature on the bump.

Fig 19: (a) The distance from the point p to the vertex a here is $\epsilon/\sqrt{2}$ where the sides of the square have a length of ϵ . The point p is further away from the vertices a, b, c, and d than any other point on the square. (b) The distance from the point p to any of the vertices of the square will depend on the curvature of the curve passing through p.

In general the value of η is relative to the maximal curvature on the faces of the grasped part. If c is the maximum absolute value of the curvature of any curve on any face of the grasped part, then $\eta < (\sin^{-1}(\sqrt{2}\epsilon c))/c$ when $\epsilon < 1/(\sqrt{2}c)$ because the distance between any two nonadjacent vertices of a planar facet must be less than $(2\sin^{-1}(\sqrt{2}\epsilon c))/c$. Therefore ϵ must be chosen so that it is less than $1/(\sqrt{2}c)$ so that we can approximate the value of η with the upper bound given above. In the case shown in Fig 19b, ϵ is too large, and this approximation will not hold.

To find η , we must find the maximal curvature on any of the faces of the grasped part. For simply shaped faces, such as planar faces, cylindrical faces, and elliptical faces, we can find a mathematical expression for the maximal curvature. For a complicated surface we might need to compute the local curvature for every point on the face to find the maximal curvature. If this computation is considered to be prohibitively expensive then the user would need to compute the maximal curvature manually and specify this value to the program.

To calculate precisely the value of $C(X)$, we start by computing the value of $\min_dist$ for each vertex on the polyhedral approximation. Assuming that d^* is the smallest value of $\min_dist$ for any vertex, those vertices whose computed value of $\min_dist$ is smaller than $d^* + \eta$ are considered further, we will refer to these vertices as near-minimal points. The reason we find these near-minimal points is that the closest point on the grasped part to the fixtured part must be in a neighborhood of one of these near-minimal points. The neighborhood of each near-minimal

point will have a radius of η .

We next perform a gradient search starting at each near-minimal point. Each gradient search is confined to the infinite or closed surface containing the face of the near minimal point, and the gradients are computed with respect to the *min_dist* function. (Of course we must halt the gradient search any time it reaches the boundary of a face.) Therefore, we take the smallest value of *min_dist* found by any of the gradient searches as the value of $C(X)$.

An assumption we are making here is that we will be sampling the boundary of the grasped part at a high enough resolution that there will be only one point that has a locally minimum value of *min_dist* in any η -neighborhood of a near-minimal point. For this to be the case, any two points of locally maximum curvature on the boundary of the fixtured part must be separated by a distance greater than $2*\eta$, since the projection of these two points onto the boundary of the grasped part might give rise to two local minimum points. (Here we define a point to have a local maximum curvature if a curve passing through this point on the surface of the fixtured part has a locally maximum curvature at this point). Therefore, if the η computed above is greater than half of the smallest distance between any two local maxima curvature points on the fixtured part we must decrease the value of ϵ until the corresponding value of η is sufficiently small.

Because of the first constraint placed on the value of ϵ , there will be only one point of maximum curvature on a face of the grasped part in any η -neighborhood. Unfortunately, a local curvature maximum on a face of the grasped part might correspond to a local minimum of *min_dist* on the face while a local curvature maximum on the boundary of the fixtured part might correspond to a different local minimum on the face, and the distance between these two local minima can become arbitrarily small depending on the placement of the grasped part with respect to the fixtured part. Therefore it is possible, but not likely, that we may have two local minima in an η -neighborhood. To test for this situation, in addition to performing a search for a local minimum, we also perform a search for a local maximum. If we travel a distance greater than η from the starting point, there is no maximum in this neighborhood so there can only be one local minimum. If we come across a local maximum then we start a new gradient search for a local minimum at a point just slightly farther away from the starting point but in the same direction as the local maximum point. The idea here is to get out of the valley formed by the first local minimum and into the valley formed by the other local minimum. A search from this point should yield the second local minimum, unless we travel a distance greater than η from the starting point, in which case we halt the search because there is no second local minimum within this neighborhood.

To compute the minimal distance between local curvature maxima on the fixtured part, we not only have to worry about local maxima on continuous faces of the fixtured part, as shown in Fig. 20a, but also vertices of the fixtured part (as shown in Fig. 20b) and edges (as shown in Fig. 20c), which also give rise to local curvature maxima. Since the computational procedure required to locate all of these local curvature maxima would be prohibitively expensive for general surfaces, it would usually be best if the user examined the boundary of the fixtured part and specified the minimum distance between local curvature maxima on the fixtured to the program.

To help the user specify values of both this distance and the maximal curvature on the grasped part, it might be possible to create an interactive graphical routine for this purpose.

Fig 20: (a) Two local curvature maxima on a face of the fixtured part. (b) Two local curvature maxima formed by vertices on the fixtured part. (c) Two local curvature maxima formed by edges on the fixtured part.

A pseudocode implementation of the procedure for calculating $C(X)$ is given here.

```

procedure clearance(X,samples)
begin
1. Transform polyhedral approximation to position and orientation defined by X.
2. Find  $min\_dist$  for each vertex on polyhedral approximation.
3. Identify near minimal points on polyhedral approximation.
4. For each near minimal point  $v_i$ 
5.    $x_i \leftarrow$  result of gradient search in neighborhood of  $v_i$ .
   Comment Gradient search performed on boundary of grasped part, minimizing  $min\_dist$ .
6.  $v^* \leftarrow$  the point  $v_i$  with smallest  $min\_dist$  value.
7. return( $min\_dist(v^*)$ );
end

```

The complexity of this procedure is dominated by the process of finding the near minimal points from the polyhedron vertex set since here we must calculate the value of min_dist for every vertex on the polyhedral approximation. Therefore since the complexity of the min_dist calculation was shown to be $O(M \times N \times F^3)$ then the overall complexity of the clearance procedure is $O(S \times M \times N \times F^3)$ where S is the cardinality of the boundary sample set for the grasped part and M , N , and F have previously been defined.

4.2 Alternative Methods of Path Planning

The potential field based-approach is only one approach commonly used for path planning. Assembly motion planners based on two competing path planning schemes were discussed in Section 2. Here we will discuss these path planning schemes and others, but we would like to note that this discussion is very brief and a much better survey of the work that has been done in path planning can be found in [35].

In the configuration space approach for path planning used by Donald and others [9, 15, 17, 40] the *configuration* of the object is defined by three parameters specifying the position of the object and three parameters specifying the orientation of the object. A configuration

space representation is simply a map of all of the configurations of the object; if a configuration of the object would result in a collision with an obstacle in the work space, the point corresponding to this configuration in the configuration space would be inside of a *configuration space obstacle*. Alternatively if a configuration of the object would not result in a collision with a physical obstacle, the corresponding point in configuration space would not be contained in any obstacles. Hence the "free space" in the configuration space represents all non-collision configurations of the object with respect to the obstacles. By using a configuration space representation, the problem of path planning is reduced to that of finding a path for a single point in the configuration space map, referred to as the C-space.

Several methods have been proposed for this task. O'Dunlaing and Yap [44] originated the retraction method that retracts the free space in the C-space onto its Voronoi diagram. Brooks [4] introduced the freeway method which extracts freeways in the work space consisting of generalized cylinders. Both the retraction method and the freeway method work only for low-dimensional C-spaces.

Canny [8] developed a roadmap method that works in on a C-spaces of arbitrary dimensionality. This method however is of exponential complexity in the dimension of the C-space. Donald [14] presented a similar approach that employs heuristics he refers to as "local experts" to reduce the complexity for the expected case.

While theoretically a configuration space representation can be used to precisely represent any set of obstacles and an object, most practical algorithms for computing this representation approximate the shape of the the object or obstacles and produce a quantized map. Both of these factors may lead to imprecision in path planning which, as we will show, is highly undesirable for assembly motion planning problems where very tight fitting parts might be involved.

The contact space approach for path planning used by Laugier and others [33, 36, 37] converts a configuration space map into a graph where classes of equivalent robot states are represented by the nodes of the graph. A robot state is defined by the configuration of the object and the set of contacting surfaces, edges, and vertices on the object and the obstacles; those states that have equivalent contact relationships associated with them are considered to be equivalent states.

This representation is very good for finding the relationship between the parts for any given position and orientation of the object, but does not lend itself to planning motions between two configurations represented by the same node. For instance, since the representation is based on contact, all poses of the object that do not result in a contact with the obstacles are grouped into the same node. So if it is necessary to maneuver the object into a reasonable position before beginning to physically mate the parts, for example if the object must first be brought into a complicated housing, a motion planner based on a contact space representation would not necessarily be able to find the best possible path. Furthermore the methods used to synthesize and analyze motions between nodes in a contact space representation are currently very limited and oftentimes heuristics must be employed to find feasible motions.

Schwartz and Sharir [54] have presented a path planning algorithm, often referred to as a cell decomposition method, based on partitioning free space into non-critical regions and critical curves. From this straight-line path segments are drawn through subregions created by the curves, and heuristics are employed to join segments across subregions into a connected collision-free path. A similar approach was independently derived and presented in [12]. Herman [22] presented an approximate cell decomposition method that uses an octree representation of free space and is fast but produces approximate paths.

We have found that for the purposes of planning paths for assembly mating operations, there are many benefits to using a potential field representation. For instance, this representation can be obtained easily from the CAD models of the parts and in our system this is done automatically with the assembly representation discussed in Section 3. Also, as we have shown, potential fields can be used to represent obstacles exactly so long as they can be decomposed into a set of convex pieces possibly containing convex holes. A third advantage is that the potential field representation can be used not only to tell if a contact occurs, but also the precise direction to move in to avoid a collision. Finally, we have found it easy to incorporate into potential field representations uncertainty information, and although we will not give further details in this manuscript, information about articulations of parts and flexibilities of parts.

5. PHASE 1 PLANNING: IGNORING UNCERTAINTY

In Phase 1 planning, the uncertainties are ignored and a path is developed that will be collision free in the ideal case. In other words, the job of Phase 1 is to generate a path plan. The purpose of a path planning in general is to find a path from a starting point to a goal point such that when an *object* of interest is moved along this path, it does not collide with any *obstacles* in the work space. In path planning for assembly, the grasped part is the *object* and the fixtured part or subassembly is an obstacle, or, in some cases, a set of obstacles if it becomes necessary, for algorithmic reasons, to decompose the fixtured part into a set of obstacles. By posing the problem in this manner, any standard path planner could, in principle, be used to find the underlying path that coarse and fine motions in an assembly motion plan must follow.

As listed in Table 3, there are, however, some significant differences between the standard domains of path planners and the assembly domain. Our purpose in showing Table 3 is to point out that it might be preferable to use a path planner that is specially designed for assembly applications. We will present such a path planner that uses the method of potential fields to represent obstacles which was outlined in the preceding section. Our particular usage of the method here is an extension of the work done by Hwang and Ahuja [24, 25], the major extensions being related to those issues outlined in Table 3. As our discussion proceeds through this Section we will show how the first three issues in Table 3 are addressed by our path planner. We showed in the last section how the fourth issue in the Table can be addressed by allowing for "negative obstacles." We know of no other path planning system intended specifically for assembly mating operations. Hence the fundamental advantage of using our path planner over others is that only ours will not suffer from any of the problems outlined in Table 3.

Path Planning Used in Standard Domains	Path Planning Required for Assembly
The resolution with which a path is computed is usually constant along the path.	Small clearances necessitate that the resolution with which a path is computed be a function of the clearance between the object and the obstacles, so that the path is adequately represented without an excessive number of points.
Small deviations from a collision free path are still collision free.	Small clearances imply that small deviations from a collision free path may result in a collision.
The <i>goal point</i> of the object (the position of the centroid of the object when in its final pose) will be outside of the obstacles.	The <i>goal point</i> of an object with a hole in its center may lie within an obstacle, as, for example, when mounting a gear onto a shaft.
Obstacles may be required to be convex; obstacles that would otherwise be concave are decomposed into convex pieces.	Assembly parts often contain features such as cylindrical holes which cannot be decomposed into convex pieces.

Table 3: A comparison of path planning in standard domains and the path planning required for assembly.

A more traditional use of potential field representations than that employed by AMP-CAD is based on real time path planning. In these approaches, developed by Khatib and others [31], obstacles are defined to have repulsive potentials, and the goal is defined to have an attractive potential. These algorithms operate by always moving the object in the locally optimal direction; where an optimal direction is one that tends to point toward attractive potentials and away from repulsive potentials.

The path planning done by Hwang and Ahuja [24,25] is a good example of the use of potential field representations in off-line path planning. They develop a path (that is unfortunately approximate) by first finding a good estimate of a collision free path and then performing a parallel optimization on the points in this path. Following this, if necessary, they perform a serial optimization on the collision points in this path. Finally, if no valid path could be found in these steps they consider the possibility of at some point in the path, moving the object to the side, reorienting it, and resuming the motion along the path. They refer to this as *sidetracking*. The reader might note that while our path planning algorithm follows that of Hwang and Ahuja, we have only found it necessary to employ the first and third stages of their method, but we have modified those stages significantly to make them more amenable for assembly problems.

In our path planning algorithm the collision free path is similarly represented by a sequence of "points" which correspond to the position and orientation of the object along the path where straight line segments between adjacent points are assumed. Hence the starting point, the goal point, and all intermediate points will be 6-vectors with the first three components of the vector representing the position of the center of mass of the object and the remaining three components the orientation of the object about the center of mass.

As alluded to, our algorithm consists of two stages: In the first stage, it is assumed that the object orientations vary linearly from the starting point to the goal point and an initial estimation of a collision-free path is found by "tracing out" minimum potential valleys. On account of the assumptions regarding object orientations, the path found will not, in general, be collision-free everywhere. So, in the second stage, this path is analyzed and those segments identified where collisions cannot be avoided under the assumption of linearly varying object orientations. Subsequently, in the second stage, for each such segment -- let's denote the end points of one such segment by A and B and the center point of the segment by C where A and B are themselves collision-free-- a steepest descent procedure is used to find a minimum-potential (and collision free) position and orientation of the object in the neighborhood of C; this steepest descent method operates in the six-dimensional space spanned by the three translational components and the three orientational components of the object. Ideally, the steepest descent method yields a collision-free position and orientation C'. Two new path segments, AC' and C'B, are formed assuming that the object orientations vary linearly from the computed orientation at C' to the orientations at A and B. The second-stage procedure is then recursively invoked on the new path segments AC' and C'B. In the event that the steepest descent procedure cannot locate a collision-free position and orientation of the object between A and B, we resort to a using a brute-force search for a collision-free path.

5.1 Stage 1:

As mentioned above, an estimation of the path is found by following minimum potential valleys under the assumption of linearly varying object orientations from the starting point to the goal point. An efficient algorithm for finding minimum potential valleys was given by Hwang and Ahuja [24,25]. This algorithm finds the minimum potential valleys extending from the starting point to the goal point by finding a set of connected spheres linking the two points, each sphere's radius being the maximal possible in the free space. The set of connected spheres are represented in the computer by a graph, as will be explained in what follows.

Suppose we want to find one or more minimum potential valleys from the point marked START to the point marked GOAL in Fig. 21a where we have depicted an L-shaped object that is to be moved from its position at START to its final position at GOAL while avoiding the obstacles. We start out by finding the maximum possible radius of the sphere, with center at the point START, which would be contained entirely in free space. This radius, denoted by *min_dist*, is equal to the minimum distance from START to any of the obstacle surfaces. As demonstrated in the preceding section, when the obstacle can be considered to be an agglomeration of positive or negative convex volumes, it is possible to use a reasonably efficient algorithm

for computing min_dist . Therefore, we will not dwell any further on the computation min_dist .

Fig 21: (a) An assembly operation which is to put the L-shaped object into the L-shaped hole. (b) A sphere is drawn about the starting point of the path, where the radius of the sphere is equal to the value of min_dist for the starting point. (c) The spheres attached to the starting sphere correspond to some of the points of local minimum potential on the surface of the starting sphere.

A sphere whose radius is equal to the computed value of min_dist is now drawn about the starting point, as shown by the circle in Fig. 21b. In addition to the obstacles that are actually present in the work space, the path planner also considers as obstacles the surfaces of an imaginary bounding parallelepiped -- depicted by the rectangle ABCD in Fig. 21b -- in order to bound the free space. That is why in Fig. 21b the size of the sphere is limited by a min_dist that corresponds to the side CD of bounding parallelepiped.

Sampling the Potential Field on a Sphere

In order to locate the potential minima we must come up with a computer representation of the potential distribution on a sphere.* Therefore, the sphere is tessellated in some fashion and the vertices of the tessels are used as sampling points for the potential distribution. An ideal tessellation scheme would create cells, or *tessels*, that are symmetric, identical in shape, and possess equal areas for any given resolution of cells. Unfortunately, it is well known that such a scheme does not exist [23]. The closest one can come is to project a regular solid, such as the icosahedron shown in Fig. 22a, onto the sphere to be sampled, and then divide the triangular faces of the projected solid into smaller triangles. This is done by dividing each edge of a triangular face into Q each parts and joining the additional points to form Q^2 smaller triangles; the division is referred to as a geodesic tessellation of the sphere. Fig. 22b shows a geodesic tessellation of a sphere for $Q=4$. The number Q , referred to as the frequency of geodesic division, can be used to vary the resolution of cells and is customarily taken to be a power of two. A more complete description of geodesic tessellation techniques can be found in [30]. For our algorithm, we have used the vertices of tessellations, such as those shown in Fig. 22b, for sampling the potential distribution on the surface of a sphere. Later, we will discuss the choice of Q with respect to the parameters of an assembly.

To access the vertices of the tessellation we use the 3D-POLY indexing strategy, reported on in [11], which allows the neighbors of a vertex to be found in constant time. The 3D-POLY indexing is best explained by imagining that the icosahedron can be “flattened out” into five parallelograms, each consisting of four triangular faces, as shown in Fig. 22c. The vertices obtained by geodesic tessellation can then be displayed by a finer division of the triangles, as depicted in the leftmost parallelogram in Fig. 22c, where we have used indices labeled j and k to locate points in this finer division. Hence, each vertex on the sphere of Fig. 22b can be given a unique triple index $[i,j,k]$, where i identifies a parallelogram, and j and k locate the point in the i^{th} parallelogram. The reader is referred to [11] for further details, especially if the reader is concerned about those vertices in Fig. 22b that are on the “tear” edges in the flattened version of the icosahedron in Fig. 22c, which it turns out do not cause complications. Also presented in [11] are simple formulas for computing the neighbors of a given vertex.

Choosing a Sampling Density

Note that for a given frequency Q of geodesic tessellation, there will be $10 \times Q^2 + 2$ sampling points on the *min_dist* sphere, and the radial angle between any two adjacent sampling points will be roughly $\text{atan}(2)/Q$. The total number of sampling points, and therefore the choice of Q , is determined by the following two conditions:

SamplingCondition# 1: The density of the sampling points must be such that at least one point is located between any two adjacent local maxima that confine the assembly path. With

*The manner in which we find the potential minima on the surface of a sphere is different from the approach used in [24, 25]. We believe our method is more robust and particularly suited to path planning for assembly where the sphere may have to be sampled densely.

Fig 22: (a) An icosahedron, (b) A geodesic polyhedron derived from subdividing the triangular faces on an icosahedron with a frequency of four. (c) A flattened-out icosahedron depicting an indexing scheme for the polyhedron of (b).

reference to Fig. 23, the potential distribution on the surface of the bottom half of the sphere will be roughly as shown in Fig. 23b. Clearly, we can ignore the fluctuations in the potential caused by small features such as the screw holes. However, we do want that at least one sampling point be within the maximas caused by the structure containing the hole. Since the object must pass through the hole, we may state this sampling condition in terms of the relevant dimension of the object -- we will call it *obj_size**. This condition then states that the maximum distance between two adjacent sampling points on the sphere be less than *obj_size*. This implies that Q should obey $\min_dist \times (\tan(2)/Q) < obj_size$.

* For a convex object the value *obj_size* simply corresponds to the minimum extent of the object. For concave objects, consider a line drawn through the center of mass the object. If the center of mass of the object lies within the object, we find the shortest line segment on this line contained completely within the object. If the center of mass lies within a hole of the object we consider the shortest line segment contained completely within the object or the hole containing the center of mass. Now, we find the shortest such line segment for any line drawn through the center of mass of the object, *obj_size* is taken to be the length of this line segment.

Fig 23: (a) Here the desired path is into the larger hole on the right. Because the screw hole is so small that the peg could not fit through it, there is no reason to consider a path into this hole. (b) The potential is shown for a set of points near the two holes along the surface of the sphere.

SamplingCondition#2: As we will discuss later, those sampling points on the sphere that correspond to potential minima are candidates for paths. If the sphere is sampled too coarsely, the resulting path could be jagged; this jaggedness might lead to an excessively large number of collisions when the clearance between the object and the obstacles is small. The idea of the sampling condition #2 is to ensure that the number of collision points discovered along an initial path is as close as possible to what the number would be for the path along the true potential valleys.

Strictly speaking, for the sampling rate on the sphere one should take the minimum of the two conditions above. We have empirically found that for clearances on the order of 0.001 in. and *obj_sizes* on the order of one or two inches, the sampling conditions are met when $Q=32$, which

translates into the use of 10,242 sampling points and a radial angle of roughly 0.0346 radians between adjacent sampling points. Of course if $Q=32$ does not satisfy Condition #1 we would have to increase Q . While Condition #2 can only be tested empirically and therefore it cannot be known during path planning if this condition is met, it is not a necessary condition but only a desirable condition, so the planner will succeed even if this condition is not met.

Locating Local Minima on a Sphere

As should become clear to the reader shortly, an assembly path may be computed either from the start position to the goal position, or from the goal position to the start position. In most cases, the latter approach will prove to be more efficient since goal positions tend to be more confined, leaving the potential distributions with only a few -- in many cases only one -- local minima. Of course, the existence of a single potential minimum is never guaranteed, so an algorithm that finds potential minima on the sphere must be able to locate them all.

There can of course be multiplicity of paths, some more optimum than others from the standpoint of manipulation effort required, for carrying out an assembly, and supposedly these paths can be discovered by appropriate deformations of the minimum potential valley paths. The minimum potential valley paths may be found by locating all the potential minima on the surface of the sphere, constructing new spheres by centering them at each such minimum, and continuing the process. Since not all the minimum potential valleys found in this manner will lead from the start point to the goal point, or vice versa, it becomes necessary that the local minima found on a sphere be organized in some manner that would be conducive to search for desired paths. In the next subsection, we will describe a graph data structure for this organization, but here we will focus on how to find the local potential minima on the sphere.

To find the local potential minima on the surface of the sphere, we first calculate the potential at every sampling point on the sphere using Eq. (6). We then test each sampling point to see if its potential is less than the potential of all of its neighbors; the sampling points where this condition is satisfied are the points of local minima. The computational effort required for this procedure is mitigated by the fact that, with 3D-POLY indexing, the neighbors of a sampling point can be identified in constant time. For the 10,242 sampling points used in our system, the algorithm must make 61,440 comparisons, with six comparisons at 10,230 sampling points and five at the remaining twelve points; this number of comparisons is not inordinately large.

Because the potential field calculation is based on exact models of the obstacles as opposed to some quantized measures, the potential of a point can be calculated to the accuracy of the computer's numerical precision. To account for the limited precision, we actually require that in order that a point be considered to be a local minimum the potential of all of its neighbors must exceed its potential by a small threshold.

However, if we take into account only this set of points we would be ignoring points that are local minimums but not strict local minimums. That is, we would exclude from further consideration those points, located on a "local minimum plateau," whose potential is less than or equal to that of its neighbors. To see how this can happen, imagine a sphere located between two parallel planar obstacle faces. If the area of each face is relatively large compared to the size of

the sphere, the only points of local minimum potential on the sphere will be those located on the great circle of the sphere that is parallel to the planar obstacle faces, as in Fig. 24. Furthermore, none of these points will be strict local minima. Because mating faces of fixtured parts might easily take on such a configuration it is essential to consider local minimum plateaus as well as strict local minima. (There may also be points that are strict local minima but because of limited numerical precision would not be recognized as such, and the following procedure will find these points as well).

Fig. 24: (a) A local minimum plateau is formed about the great circle of the sampling sphere that is parallel to the obstacle surfaces. (b) In reality the local minimum plateau will have some width, as shown above, due to the fact that points whose potential does not exceed its neighbors by more than a small threshold are accepted as local minima. This is done to account for limited numerical precision.

Therefore we create a list of all of the points on the surface of the sphere whose potential (minus a small threshold to account for the limited numerical precision) does not *exceed* the potential of any of its neighbors. After all of the points on the sphere have been considered, we segment this list into clusters of neighboring points, each cluster corresponds to a region of local minimum potential. If a cluster spans a large longitudinal and/or latitudinal distance, as is the case for the cluster of points shown in Fig. 24, the cluster is divided into a number of smaller clusters. This is done so that each connected cluster represents only a small interval in both the longitude and latitude angles. Along with the strict local minimas, we choose for further consideration the center point of each of cluster.

To avoid consideration of extraneous points, it is possible to establish a threshold so that clusters that do not contain a minimal number of points are discarded. However there is always the danger that if this threshold is too large, small clusters that might actually correspond to a true plateau will be discarded. Similarly, if the threshold used for the selection of a strict potential minimum is too high, a strict potential minimum point located in a region of slowly varying potential may be rejected. To get around these difficulties, we set the second threshold to some reasonable level derived empirically, but accept any cluster regardless of size. Therefore, if a point is a strict potential minimum in a slowly varying region, it will at the very least be

accepted as a cluster of size one. However, if there were points very near to it that appear as though they may be local minimas, they will be grouped with the first point and only one point in this region will be further considered. In the event that points are falsely accepted as local minimas, the graph search might take longer than necessary, but this would usually not be the case because we have a powerful heuristic for pruning such points, and because we have a powerful heuristic for guiding the graph search. Of course, in the unlikely event that a non-minimum point became part of the derived minimum potential valley then either a collision free path would be found in the second stage, at which point it is not important whether or not the point was a local minimum, or the second stage would fail and the planner would backtrack to the first stage.

Graph Search for Finding Minimum Potential Valley Paths

Given that only some -- in many cases only one -- of the minimum potential valleys would correspond to an assembly path, we need a representational facility for all the different minimum potential valleys that would allow the desired valley(s) to be identified. The potential valleys found by concatenating spheres in the manner described previously are conveniently represented by an attributed graph data structure, the nodes of the graph corresponding to the centers of the spheres and the arcs the straight-line segments of the potential valleys between the centers of the spheres. The attribute list stored at each node is a record consisting of the following attribute-value pairs:

$$\left\{ \begin{array}{l} \text{node_coordinates : value} \\ \text{min_dist : value} \\ \text{object_orientation : ?} \end{array} \right\}$$

where the attribute *object_orientation* remains uninstantiated until a complete path has been discovered. As we mentioned earlier, it is assumed that in the path calculated during Stage 1 the object orientations are linearly varying along the path. Therefore, it is not possible to instantiate the last attribute until the entire path has been found.

Of course, it is not necessary to discover the entire graph that would represent all the minimum potential valleys for a given configuration of parts for assembly, unless of course we want to find all possible assembly paths, that being unnecessary since we use a best-first search strategy to grow a tree of paths in the underlying graph. If we start the path computation from the goal node, our best-first strategy finds all the potential field minima on the sphere centered at the goal point, evaluates each of these using a heuristic function, selects the point to be used for construction of the next sphere, and so on. In this manner, the algorithm 'grows' a tree of paths in the underlying graph of minimum potential valleys. The tree growing process comes to a halt when the start node is discovered. The heuristic function for node evaluation has two components: One is the computation of the 'air-distance' heuristic, meaning we select that node for expansion which has the shortest distance to the START node. (The reader may note that there is rich tradition to the use of air-distance functions in heuristic search [43,46]). Second, the node selected on the basis of minimum air-distance is then subject to a validity test that compares the value of *min_dist* for that node with the *obj_size*; roughly speaking, the node is

accepted if the new value of min_dist computed at that node is larger than obj_size . The validity test, which we will now explain in greater detail, ensures that certain necessary conditions are satisfied for a potential minimum to lie on an assembly path. Since the test is easy, it is capable of quickly eliminating the impossible potential minima. Of course, since this test does not constitute a sufficient condition, an inapplicable potential minimum may still be accepted, but then such points would be eliminated either by subsequent truncation of the corresponding paths as the best first algorithm tries to extend them, or by the procedure described in Stage 2.

Now for the validity test itself, a two-step computation is carried out: In the first step, the value of min_dist is computed at the node; this min_dist is equal to the distance to the nearest obstacle surface from the node in question. Then, in the second step, the following inequality is tested $2 \times min_dist(1 + (\tan(2)/Q)) > obj_size$; the node is accepted if the inequality is satisfied. Basically, the test is a comparison of $2 \times min_dist$ with obj_size ; the factor $min_dist(\tan(2)/Q)$ is included in the inequality to account for the sampling interval between the adjacent sampling points on the sphere.

To further clarify the scope of this validation test, we present Fig. 25 where, in parts (a) and (b), we have shown the mating of a long and oversized piston with a cylinder -- a physically impossible operation; and where in part (c) we have depicted the same with a short and oversized cylinder, again an impossible assembly. We will also assume that Q is arbitrarily large, making sampling a non-issue. When a node corresponds to the point marked as O in part (a) of the figure -- in other words, for points on the assembly path in the interior of the cylinder -- the parameter min_dist will be equal to the radius of the cylinder, as shown in the figure. Therefore, $2 \times min_dist$ will be smaller than obj_size , implying that, as we would want to be the case, the node will fail the validation test. Next, assume that, as shown in part (b), a node corresponds to the center O of the sphere drawn there; in other words, the piston is close to the mouth of the cylinder. Now min_dist will equal the distance OA, assuming for the sake of explanation that the point O is symmetrically placed with respect to the edge points A and B. In this case, depending on how much larger the piston diameter is compared to the diameter of the cylinder, it could easily happen that obj_size would be less than $2 \times min_dist$, meaning that the node would pass the validation test, even though the assembly is clearly impossible. This possible conclusion should support our earlier statement that the validation test is a test of only a necessary condition, and not a test of a sufficient condition -- the validation test will on occasion accept nodes that assembly would not be physically possible. The validation test would also be successful for the assembly in part (c) because the parameter obj_size , which for convex objects is the minimal extent of the object, now corresponds to the thickness of the piston.

A special procedure must be used to find a graph from which an initial path can be extracted when the goal point is located within an obstacle as is often the case for mating operations that involve putting a part containing a hole onto a peg-like mating feature. For example, for the assembly depicted in Fig. 26 involving a gear, the center of mass of the gear lies in the hole in the gear. The goal point corresponds to the position of the center of mass of the grasped part when it is in its assembled position; so in this case the goal point would be inside the gear shaft. Obviously there is no minimum potential valley extending into the gear shaft since points

Fig 25: (a) A situation where obviously the value of *min_dist* for the node is too small to warrant further consideration of the node. (b) A situation where the validity test fails to prune a node even though it can not be on any valid assembly path. (c) Another situation where the validity test would fail to prune a node, if it is possible to reorient the object at this point, it will fit through the hole easily.

inside of any obstacle will theoretically have infinite potential.

Therefore whenever the goal point lies within an obstacle this obstacle and the other obstacles on this same mating feature are not considered when computing the potential function and the value of *min_dist* for the purposes of finding a graph representing the minimum potential valley. In this way the planner will be able to connect the minimum potential valley to the goal point, but the resulting graph will not represent a minimum potential valley; in the following we will refer to it merely as a valley.

The ultimate goal of Stage 1 planning, however, is not to find a minimum potential valley but instead to develop an initial path that avoids obstacles as much as possible on a global scale; problematic areas of this path will be locally refined in Stage 2. For example, imagine that we are developing a plan to mate a gear with a gear shaft but that in order to do this we must first put the gear into a housing where there are other parts already in place within the housing. A

Fig 26: A gear, whose center of mass is not contained in the gear, to be put onto a shaft.

path must be found that avoids the sides of the housing and the other parts within the housing but does bring the gear onto the shaft. The importance of Stage 1 is that it discovers the best way to bring the gear into the housing and to the shaft while avoiding other obstacles. The portion of the path that involves the actual mating operation, which necessarily brings the gear very close to the shaft, will often not be collision free at the end of Stage 1 and will therefore be revised in the second stage of planning.

The valley found in the way we have proposed will produce a good initial path in this sense. The portion of the initial path that brings the center of mass of the gear into the gear shaft is clearly a haphazard choice. However, due to the tightness of most mating operations and the fact that the shape of the object is ignored during Stage 1, the portion of the initial path that involves the actual mating operation is somewhat of a haphazard choice in any case.

For complicated mating operations such as the one depicted in the top of Fig. 27a, the initial path found will be very poor. However, the initial path found by considering a true minimum potential valley for the mating operation depicted in the bottom of Fig. 27a, where the problem is reversed, will not be good either. In contrast the initial path found for the simple mating operation depicted the top of Fig. 27b will be collision-free, as will the initial path for the mating operation shown in the bottom of Fig. 27b.

Constructing a Path along a Minimum Potential Valley

Previously we presented a best-first procedure for the extraction of a minimum potential valley from the start node to the goal node. This procedure is contained within the while loop in the main function of the path planner, as indicated in the source code given in Appendix 2. This minimal potential valley will be represented by a sequence of sampling points corresponding to a set of nodes in the aforementioned graph search. In order to construct a path along the minimum potential valley an orientation vector must be associated with each of the nodes in this sequence.

Fig 27: (a) A complicated mating operation where in the top part the problem is to put the shoe onto the foot and in the bottom part the problem is to put the foot into the shoe. (b) A simple mating operation where in the top part the problem is to put the cap onto the shaft and in the bottom part the problem is to put the shaft into the cap.

Therefore a path can be constructed by listing the 6-vectors associated with each node in the sequence, the 6-vector corresponding to the 3-vector representing the position of the node appended with the orientation vector assigned to the node.

However, the sampled path thus obtained may not adequately represent the true path for the purposes of Stage 2 computations. The reason being that the path may be undersampled; two adjacent points on the path may represent perfectly reasonable positions and orientations of the grasped part, however there may be positions and orientations that will be achieved by the grasped part if moved between the two points that result in a collision. Such a collision cannot be sensed by Stage 2 since there is no sampling point on the path corresponding to any of the collision positions and orientations. For this reason it is important that we add enough points between any two existing points on the path, by interpolating between them, so that the true path is adequately sampled. In other words, the resolution of the sampled path must be fine enough that all collisions can be discovered.

The resolution of the path in any one particular region is dependent on the smallest distance between the object and the obstacle in that region, this distance is often equivalent to the clearance of the mating operation, and can be computed using the clearance function $C(X)$ discussed in the last section.

Clearly the object can be moved a distance of *clearance* in any direction without becoming involved in a collision before the motion ends. Therefore, if there is no change in orientation along the portion of the path under scrutiny, it is safe to put the next sampling point at a distance of *clearance* from the current sampling point. If there is a change in orientation between the two points then it is necessary to make sure that the orientational variation is adequately sampled. To explain how this is done we first note that the maximum distance a point on the object can move from its original position when the object is rotated by an amount described by the 3 parameters ϕ, θ, ψ (corresponding to rotations about the X, Y and Z axes respectively) is certainly less than $m(|\phi| + |\theta| + |\psi|)$. Here m is the maximal extent of the object if it is convex and is the maximal extent of the smallest convex shell completely containing the object otherwise. If the distance between two adjacent sampling points is ΔS then we must have $\Delta S + m(|\phi| + |\theta| + |\psi|) < \text{clearance}$, assuming *clearance* is greater than zero.

To summarize the procedure for finding an initial path, which is implemented by the function *initpath* in the *pathplan* module listed in Appendix 2, first an orientation vector is associated with each node by linearly interpolating between the starting orientation and the goal orientation. Then, beginning with the start node, we insert the 6-vector associated with the node into the path and then find the *clearance* associated with the node. If the value of *clearance* is zero, we move on to the next node because the point corresponding to this node will be flagged as a collision point by Stage 2 and an appropriate path through this region of the path will be constructed when the collision point is modified.

Otherwise the largest ΔS is found such that $\Delta S + m(|\theta_x| + |\theta_y| + |\theta_z|) < \text{clearance}$, where $(\theta_x, \theta_y, \theta_z)$ describes the change in the orientation of the object as it is moved a distance of ΔS along the path. If ΔS is less than the distance between the current point and the point corresponding to the next node in the sequence, the point corresponding to the position and orientation of the object when moved ΔS units along the path is added to the path and it is considered in the same manner as the current point. If ΔS is larger than the distance to the next node in the path, we move on and examine this next node in the sequence.

This process continues until every node in the sequence has been considered. The outcome is that there will be many points in an area of tight constriction and fewer points where there is more free space surrounding the path. Fig. 28a illustrates the minimum potential valley found for the example in Fig. 21, Fig. 28b depicts the points along the initial path found for this example, and Fig. 28c depicts the object at each of these points. (Note that the parts depicted in this figure are three dimensional, so they have six degrees of translational and rotational freedom. However, due to symmetries, the position along the Y axis and the orientation about the X and Y axes remain constant. Therefore the object is depicted as a two dimensional object with three degrees of freedom. Similarly a two-dimensional slice through the center of the obstacles is shown).

5.2 Stage 2:

The purpose of the second stage of path planning is to adjust any collision points on the initial path so that collisions no longer occur at these points. To do this, Stage 2 examines the initial

Fig 28: (a) The sequence of nodes representing the minimum potential valley that connects the starting node to the goal node. (b) Points chosen to form an initial path through free space. (c) The object placed in the position and orientation dictated by the initial path. (d) The collision free path produced by Stage 2. (e) The final path where unneeded points have been removed by the function slp listed in Appendix 2.

path and groups together successive collision points into collision regions, the collision regions being padded with one or two non-collision points on each end of the region. For each collision region, a collision free replacement point is found by using the center point of the region as the

initial estimate for a gradient descent optimization procedure (to be discussed shortly) that iteratively minimizes the the object potential function given by Eq. (10). This optimization procedure is implemented by the fbfc routine listed in Appendix 2.

A new path segment is generated passing through this region starting at the first point in the collision region, going through the replacement point, and ending with the last point in the region. This path is generated in the same manner that the initial path passing through the minimum potential valley was generated. This new path segment is recursively revised by Stage 2 until no collision points remain. Each successive collision region in the initial path is treated in the same manner as the first collision region. Fig. 28d depicts the output of the second stage for the example under consideration.

The optimization procedure used by the second stage is of central importance in that its ability to find collision free replacement points will determine the ability of the second stage to find a collision free path quickly. The procedure we chose to use is based on a gradient descent search that follows the negative gradient of the object potential function. Other, possibly more efficient, optimization procedures place strict restrictions on the cost function to be minimized, here taken to be the object potential function, and if these restrictions are not met, certain steps in the iteration may not be computable. For instance, when using Newton's method, one needs to compute a Hessian matrix of the cost function and this matrix cannot be singular. Because of the highly nonlinear nature of object potential function this can not be guaranteed.

Of course, as is well known, a perennial shortcoming of the gradient descent method is that the solution will always get trapped in the first local minimum found for the potential function; this local minimum will not be acceptable if the resulting position and orientation of the object is not collision free. For the example shown in Fig. 29a, if the orientation of the object when gradient descent method is invoked is as shown in the figure, the closest local minimum will corresponding to the orientation shown by dashed lines -- clearly an unacceptable solution. There is one more difficulty with the application of the gradient descent method: the solution found may correspond to an earlier location and/or orientation of the object. Let's say that the orientation of the object when the gradient descent method is invoked is as shown in Fig. 29b, the minimum potential solution may get trapped in the location shown by the dashed lines -- which may correspond to an end point on the path segment under examination. When the gradient descent method fails to produce a solution, a brute force search strategy, called the bumble strategy and suggested originally by Donald [14] is used. As the name implies, this strategy consists of marching outward in twelve different cardinal directions from a given point in the six dimensional search space and carrying out best-first search. Fortunately, for the types of geometries we have analyzed, the exhaustive search method is seldom invoked. Usually, exhaustive search becomes necessary only if the orientation assigned to the object by the stage 1 algorithm is at great variance from the true collision free orientation; however, for most practical assemblies that is almost never the case. The bumble strategy is performed by the path planner function bumblepath whose source code is listed in Appendix 2.

Fig 29: (a) A situation where the nearest local minimum may not be collision free. (b) A situation where the nearest local minimum may be an endpoint of the path segment.

6. PHASE 2 PLANNING: INCORPORATION OF UNCERTAINTY

Once a collision-free path has been derived for an assembly operation by the Phase 1 module, AMP-CAD enters Phase 2 where uncertainties are incorporated into the potential field representation of the parts, their effects are analyzed, and the results of this analysis are used to produce a final motion plan. Before beginning a discussion on Phase 2 processing we would first like to introduce an assembly example, pictured in Fig. 30, to be used to illustrate the process of formulating an assembly plan given a collision-free path for the assembly operation to be performed.

Fig 30: The goal here is to mount the compound gear A on the shaft B and the simple gear C on the shaft D. We will assume the former has already been accomplished.

In Fig. 30 we have shown a housing H containing a compound gear A that is already mounted on shaft B and a simple gear C that needs to be mounted on the shaft D and, at the same time, meshed with gear A. Phase 1 will have found a path for gear C similar to the one shown by the dotted line in the figure. Because the clearances between the shaft D and the hole in the gear C is tight and due to the fact that gear C must be meshed with gear A, some parts of the motions will have to be guided by force/torque feedback. In the following we will show how this information is obtained using the CAD models of the parts and how a motion plan is generated.

6.1 Representing Uncertainty

A key factor in assembly motion planning, then, is how to go about representing uncertainties and analyzing their effects. We consider three types of uncertainties:

- 1 Initial pose uncertainty- The uncertainty in the initial pose (position and orientation) of the grasped part with respect to the fixtured part at the start of the assembly mating operation. This is sometimes referred to this as “sensing uncertainty” because it is usually due to uncertainties in the sensors used to initially locate parts in the workspace.
- 2 Control uncertainty- The uncertainty in the motion of the robot that increases the uncertainty of the pose of the grasped part as it is moved away from the starting point.
- 3 Model uncertainty- The uncertainty in the geometric (CAD) model of a part usually due to the manufacturing tolerances allowed on the dimensions of the part.

The initial pose uncertainty and control uncertainty are specified by instantiating the respective attributes in the frame representation of the assembly. As shown as explained in Section 3, both these uncertainties are 6-vectors. The 6-vector for initial pose uncertainty represents the amount of uncertainty in each component of the initial position and orientation of the grasped part with respect to the fixtured part. Similarly, the 6-vector for the control uncertainty represents the amount of uncertainty incurred per unit distance traveled in each component of the grasped part position-orientation vector.

Model uncertainty is specified by tolerances on the dimensions of features of the parts when part models are created via the method discussed in Section 3. To expand upon this a little further, note that the hole in gear C may suffer from three deformities that generally arise as a result of the imperfect manufacturing processes used to produce it. That is, it may not be located in the center of the gear and perfectly aligned with the gear's Z axis as the model indicates, it may not be perfectly round, and its radius may be smaller or larger than the desired radius for the hole. To represent these deformities a single tolerance on the radius of the gear is defined, such that, for the worst case imperfections, there will be no material inside a cylinder of nominal radius minus the tolerance collocated with the hole, and there will be no lack of material due to the hole outside a similarly located cylinder whose radius is the nominal hole radius plus the tolerance.

Before we discuss the mechanisms available to AMP-CAD for dealing with the model uncertainties, we must further explain what is meant by the ‘mmc’ attribute in the frames discussed in Section 3. As stated earlier, the attribute ‘mmc’ stands for the *most material condition* form of the object represented by the frame. The value of this attribute is a pointer to the BRep (boundary representation) of the object in its most material condition, a condition that is obtained by enlarging the dimensions of all the additive features by their tolerance values and shrinking the dimensions of all the subtractive features, such as holes, again by their tolerance values. If we had to check whether model uncertainties alone would demand that a certain motion be executed under force/torque guidance, we would look for possible collisions between the mmc representation of the grasped part with the mmc representation of the fixtured part.

Using potential fields, such a check would be computationally simple, since all we would have to do would be to measure the integrated potential on the boundary of the grasped part, the potentials given rise to by the fixtured parts in their mmc forms.

The initial pose and control uncertainties lend themselves to a unified treatment since the former is represented by absolute values for the deviation vector $(\delta x, \delta y, \delta z, \delta \phi, \delta \theta, \delta \psi)$ and the latter by per unit distance values for the same deviation vector. So, at each point along a path, we can compute a total deviation vector, represented by $(\delta x_T, \delta y_T, \delta z_T, \delta \phi_T, \delta \theta_T, \delta \psi_T)$, by adding to the initial pose uncertainty a distance-integrated value of the control uncertainty.

To determine whether or not a given total deviation vector would demand that the motion to a certain point on an assembly path be conducted under force/torque guidance, the translational components, $(\delta x_T, \delta y_T, \delta z_T)$, and the rotational components, $(\delta \phi_T, \delta \theta_T, \delta \psi_T)$, are considered separately. For the translational components, the additive features in the mmc form of the grasped part is enlarged by δx_T , δy_T , and δz_T along the three coordinate axes, while the subtractive features are shrunk by similar amounts. Note that it is not possible to combine the mmc enlargement and shrinkage with the deviation-vector enlargement and shrinkage into a single step since the latter is a function of distance traveled along the path. Moreover, the mmc enlargement and shrinkage occurs on a feature by feature basis, since each feature will have its own tolerance, while the deviation-vector enlargement and shrinkage is applied uniformly to the entire grasped part.

These modified boundary representations for the grasped part are then tested for collisions with the fixtured part by integrating the potentials due to the latter on the former. This testing for collisions takes place for a set of orientations of the grasped part, the set being a sample set drawn from a three-dimensional region defined by the intervals $(\pm \delta \phi_T, \pm \delta \theta_T, \pm \delta \psi_T)$. If a collision is detected for any of the samples, the point along the assembly path is declared to be one where fine motions must be used. The three dimensional region is sampled uniformly.

In case the reader is wondering why we don't deal with the orientational components of the total pose uncertainty by finding the volume swept by the grasped part as it is rotated through all possible orientations in a manner similar to what was done for the translational components, the answer is that the BRep for the volume swept by rotations would in general require transcendental functions which are not easily computed or represented.

In contrast to our approach, the configuration space based approach, used in [5, 9, 15, 17, 40], and the contact-space based approach, used in [33, 36, 37], represent initial pose uncertainty by a three dimensional sphere in the xyz space, the sphere being centered at the ideal starting position. We believe using a sphere representation for positional uncertainties is unrealistic, since it implies equal uncertainties along all three axes. In practice, that is rarely the case. For the example of Fig. 30, there may be no uncertainty in the Z direction for gear C if it is known to be originally resting on a particular work surface.

The reader should also note the differences between how we represent the control uncertainty and how it is represented in [5, 9, 15, 17, 33, 36, 37, 40]. While we represent control

uncertainties by pose deviations per unit distance introduced by the motions of the grasped part, in these other systems control uncertainties are represented by velocity cones. A shortcoming of the velocity cones is once again it forces the uncertainty in all components of the velocity vector to be equal to one another.

Also to be noted are the differences in how we treat model uncertainties vis-a-vis how they are treated in [15]. Donald has to introduce additional dimensions in the configuration space representation of a part. In fact, in his system every nonzero tolerance increases the dimensionality of the problem by one. Our method for representing model uncertainty has no effect on the dimensionality of the problem. In the other systems [5, 9, 17, 33, 36, 37, 40] there is no mechanism at present for representing model uncertainty.

While we have contrasted our representation of uncertainty with those representations used by other assembly motion planners, we feel that it is important to discuss uncertainty representation in a broader sense. In particular, there have been efforts to produce a useful description of positional uncertainty due to imperfect sensing and control. Examples of this work can be found in [3, 16, 57, 60].

There are two general approaches to uncertainty representation, the set-theoretic approach that keeps track of only worst-case bounds on the uncertainty, and the probabilistic approach keeps track of the distribution of uncertainty. The representations used by Donald, Lagier, and us are all set-theoretic, but it was shown in [60] how a set-theoretic representation can be obtained from a probabilistic representation.

In describing our assembly representation, it was inferred that the user will specify the uncertainty in the initial pose of the object. However, the user may not always be aware of this uncertainty for each assembly operation since many other tasks might need to be performed to bring the parts and the robot to the start of the operation. For instance, the fixtured part must often be picked up and moved into a fixture and the grasped part must somehow be picked up and brought to the starting point. Any of the coarse motion operations, grasping operations, or sensing operations used to perform this task may have an effect on the positional uncertainties of the parts; as described in [60] sensing and grasping operations usually reduce positional uncertainty while coarse motion operations usually increase positional uncertainty. Hence, it may be desirable to use a system such as the one described in [60] to calculate the initial pose uncertainty for each assembly operation. Here a probabilistic representation of uncertainties is used and it is shown how uncertainties can be propagated across actions and how uncertainties associated with two or more sensors can be fused to form an overall uncertainty estimate if more than one sensor is used to locate the parts in the work space.

At this point in time we have not pursued incorporating any such strategy into AMP-CAD. We feel that uncertainty manipulation is best performed by an assembly task planner since the assembly task planner is aware of the other operations necessary to perform an assembly operation. When we eventually integrate AMP-CAD with such a planner we will incorporate an uncertainty manipulation system to calculate the initial pose uncertainty.

6.2 Converting the Assembly Path into a Sequence of Motion States

So far we have shown how points on a possible assembly path can be analyzed for whether the motions through those points should be free (purely position controlled) or force/torque guided. Next, path segments are created between adjacent points and are marked for free motions or fine motions depending on the condition of the terminal point of the segment. Each segment is represented by an attribute-value frame we call the motion state, instantiated examples of which are given in Fig. 31 for the segments mp, pq and qr in the assembly. Instantiations for the attributes are generated by further processing in Stage 1 of Phase 2 on the basis of following considerations.

Fig 31: (a) A motion state attribute-value frame created for the segment op. (b) A motion state attribute-value frame created for the segment pq. (c) A motion state attribute-value frame created for the segment qr.

The instantiation for the attribute "constraints" tells us along which of the X, Y, Z, Φ , Θ , and Ψ dimensions the grasped part is constrained and thus can not be assumed to be free for assembly manipulations such as tilting, etc, if such manipulations are called for during the final synthesis of the motion plan by Stage 2 of Phase 2. The constrained dimensions are determined by translating or rotating the grasped part along the respective dimension and checking for collisions via the potential field representation of the fixtured part. The instantiations for the "unc_red_axis" and "misc" fields are generated for only those path segments that are marked for force/torque guided motions. The instantiations for the "misc" attribute are the pointers to the features that are directly involved in the mating operations at the end point of the segment. These features are identified essentially by calculating the integrated potential of each of the features of the grasped part separately with respect to each of the features of the fixtured parts; the feature combinations that yield boundary integrated potentials exceeding a certain threshold become the instantiation of the "misc" field. The clearances between the features that are the instantiations for "misc" are used to update the uncertainties in the representation of the overall assembly, a point that will be discussed further in the next subsection.

The instantiation for the attribute "unc_red_axes" are the axes along which the uncertainties must be reduced before the termination of the segment. For example, for the segment pq in Fig. 30, subsequent stages of planning must be aware of the fact that in order for the gear to be mated with the shaft, which is what must happen at point q, the uncertainties along the X and Y axes at p or the X and Y uncertainties that might be introduced by the travel from p to q must be smaller than the clearance between the gear and the shaft. To determine these axes, the potential function is recalculated, first assuming there is no uncertainty in the translation of the grasped part at the termination point q and then assuming there is no uncertainty in the orientation of the grasped part at point q. If, by eradicating the translational uncertainty, the possibility of collision is eliminated, then the fine motion plan that will eventually be developed for this point needs to be one that reduces the translational uncertainty of the part. On the other hand, if no collision occurs when there is assumed to be no orientational uncertainty, then a fine motion plan to reduce the orientational uncertainty should be devised.

If translational uncertainties must be reduced, collision tests are then made ignoring all but the X component of uncertainty, and similarly for the Y and Z components of the uncertainty. If rotational uncertainties must be reduced, collision tests are made ignoring all but the Ψ component (the rotation about the Z axis) of the uncertainty, and then ignoring the Φ and Θ components of the uncertainty. In this way the exact components of uncertainty that must be reduced to avoid collision are found.

6.3 Final Motion Plan Generation

The previous two sections discussed the operation of Stage 1 of Phase 2; as mentioned before, this stage generates a list of motion states, like the ones shown in Fig. 31, each state corresponding to a segment of the assembly path. The source code for Stage 1 is given in Appendix 2 and is there referred to as the uncertainty analyzer. In this section, we will show how Stage 2 of Phase 2 generates the final motion plan by reasoning over these motion states. The function that performs this is called build-plan in the source code listing for the plan supervisor also in Appendix 2.

Stage 2 of Phase 2 has available to it a library of strategies, examples of which are shown in Fig. 32. Each strategy has preconditions, a motion specification function, and a post-condition function. The motion state for a segment must match the precondition part of a strategy frame in order to invoke the corresponding motion function. For example, for the motion state in Fig. 32(b), the segment pq matches the precondition of the strategy frame in Fig. 32(b) and will thus invoke the motion function *torqvecsearch*. This motion strategy, explained in detail in [21], will allow the gear C to mate onto the shaft D even under tight clearances.

Currently, Stage 2 of Phase 2 has available to it a dozen strategy frames. We have displayed in Table 4 the relationship between the instantiation for the "unc_red_axes" attribute of a motion state and the motion strategies invoked for a few cases. The references where the individual strategies were proposed are also given.

Fig 32: (a) A strategy for point to point motion through free space. (b) A strategy for reducing the X,Y uncertainty by moving along the torque vector in the X-Y plane. (c) A strategy for reducing Ψ uncertainty by rotating about Ψ until certain termination conditions are sensed.

So far we have left unexplained the need for the post-condition part of the strategy frames. To explain why such post-conditions may be needed, we first need to explain in fuller detail the workings of the Phase 2 of the planner. Until now, for ease of explaining the other aspects of the planner, we have assumed that control flows sequentially from Stage 1 to Stage 2 of this Phase. However, that is not true, as will be evident from the explanation to follow.

As was mentioned before, the initial assembly path is calculated by Phase 1 by ignoring all uncertainties. This path is then processed by the two stages of Phase 2 in the manner indicated in Fig. 33. Stage 1 of Phase 2 first performs uncertainty analysis on as much of the path as possible, ending either when the entire path has been examined or when a point on the path has been discovered that requires force/torque guided motions. The path points analyzed so far for uncertainty are grouped into point-to-point straight line segments and the last point where the need for force/torque guided motions was discovered is made a part of the last straight line segment. That

Uncertainties to be reduced	Strategy
Z	Guarded motion [27]
X and Z	Biased search in X direction [27]
X, Y, and Z	Move along torque vector until change in contact force[21]
X, Y, Z, rotX, and rotY	Tilt and slide [27, 57] (if at start of mating operation)
X, Y, Z, rotX, and rotY	RCC (if during mating operation) [57, 64]
rotZ	Rotational search (rotate about Z axis until no contact force along Z axis)
X, Y, Z, rotX, rotY, rotZ	Convex peg in hole [59] (if there is a convex peg involved)
X, Y, Z, rotX, rotY, rotZ	Tilt and slide dependent on geometry (if multiple peg insertion) [22]

Table 4: Some strategies currently in use by AMP-CAD.

is how the segment pq would be constructed for the example of Fig. 30.

As shown in Fig. 33, these segments are then passed on to the Stage 2. Now, depending on whether or not force/torque guided motions are needed in a segment, Stage 2 does different things. For the example of Fig. 30, all the segments until the segment pq will take the right branch and, through the invocation of strategies as explained before, will result in the creation of the motion specifications for the respective segments.

On the other hand, when a segment calls for force/torque guided motions, as would be the case for segment pq, the left branch would similarly invoke the appropriate strategy frame and output the corresponding motion specifications. Additionally, the left branch will execute the post condition of the strategy frame. For the segment pq, as was mentioned before, the strategy frame *search-along-torque-vector* is invoked. The post condition of this frame says *update_uncertainty()*. What that means is that subsequent to the gear mating with the shaft, the relative uncertainties between the two may be determined by their clearances, as opposed to by the initial pose and the path-integrated control uncertainties; the smaller of the two are taken to be the dominating uncertainties. Before any further points on the assembly path are analyzed, the initial pose uncertainty of the grasped part must therefore be updated. This updating is done by the post condition function and the updated information resides in the frame representation of

Fig. 33: The Phase 2 flow of control.

the assembly. Now, as shown in Fig. 33, the control shifts back to Stage 1.

In one of the competing approaches for assembly motion planning discussed, [15] uncertainty reduction is never considered. Therefore, this system is really limited to being able to, in essence, plan for assembly operations where uncertainties must be dealt with only once during the operation. For example, for the assembly depicted in Fig. 30, gear C must be mounted onto shaft D while at the same time meshed with gear A. Because of the geometry of the gears and the shaft, it is actually necessary to begin to mount gear C onto shaft D before it is necessary to mesh the two gears. Hence at one point in the assembly operation, the translational uncertainties in the position of the grasped gear have to be dealt with, and at another point in the operation, the rotational uncertainties of the grasped gear have to be handled. Without taking into account uncertainty reduction, a plan for this assembly operation could not be developed given our currently available strategies. In Laugier's approach the uncertainty model is somehow updated although it not entirely clear in [37] how this is done.

At this point we should note that our planner selects the first strategy in the library for which the preconditions are met by the current state. It is possible that there might be two strategies that can be employed on the current state so in our system we order the strategies in the library from most preferred to least preferred, the preference being based on simplicity, in order

to make sure that the best strategy is selected. A preferable method might be to consider all possible strategies and rank them according to some measure. For instance some strategies tend to work well when there is only a small amount of uncertainty regardless of the clearance and others tend to work well when the clearance is not incredibly small but large amounts of uncertainty can be tolerated. We have not, at this time, considered taking such an approach mostly because we do not have a large library of strategies where there would be many possibilities to consider for a given situation, and therefore we feel that performing a reasonable study on the best way to rank strategies using our current library would not be possible.

In many ways, the largest limiting factor of our planner is the small number of available strategies from which it has to choose. While our current library contains strategies that in theory cover almost all possible situations, some of these may be highly prone to errors in certain situations, so that even when executed under error detection and recovery, these strategies may not accomplish their goal but instead iterate indefinitely through error detection and recovery cycles. Also there are situations for which it is possible that a strategy could not be found. For these reasons, further work needs to be done in extending this library.

The focus of our work so far, however, has been more on the theoretical issue of how to analyze geometric and uncertainty constraints to decide when fine motion must be used and to characterize the type of fine motion strategy required and less on the actual implementation of strategies. We have discussed the further development of strategies in Section 7.

Each individual motion specified by a strategy in AMP-CAD is represented by a special motion data structure that we will describe in the next subsection. The execution unit that we will also describe in that subsection will be responsible for converting this representation into a robot command that is in the language understood by the robot being used. Hence, the available strategies specify motions that can be executed on a variety of robots, so long as the robots have the motion capabilities required by the strategies. The execution unit that we will discuss is limited to dealing with only force/torque guided motions and point-to-point motions to be executed by the robot. Therefore, if a motion strategy was added to the library that, for instance, relied upon using additional sensors or a dexterous gripper, the execution unit would likewise need to be extended to handle these types of motions.

6.4 Motion Plan Execution

To make certain that those segments of the motion plan that are marked for force/torque guided motions are carried out with the aid of automatic error detection and recovery (EDR), we have implemented a special execution unit to execute the motion plan and perform EDR along the lines discussed in [21]. The source code for the execution unit is listed in Appendix 2; the main routine of the execution unit is there called `execute`. In order to cope with the discrete nature of the manipulator control, the noise and other distortions in the signals produced by the force/torque sensor on the manipulator, etc., it is not possible to carry out force/torque guided motions without automatic error detection and recovery (EDR). EDR requires that the termination of each motion be verified and, upon failure, a recovery motion be carried out. In this implementation, it is assumed that the force/torque guided motions specified will account for

uncertainties, and only the errors induced by the discrete nature of force control, etc. are considered for EDR.

The execution unit also performs other functions in the execution of each step in the motion plan. In some cases it might be necessary to skip a step. For instance, if a fine motion subplan has been devised to align a grasped peg with a mating hole but this alignment is serendipitously achieved before the execution of the fine motion subplan, then some of the steps in this fine motion subplan do not need to be executed. Also, it is sometimes necessary to perform some functions before or after the execution of a motion, such as altering the value of a variable.

A step or "motion" is specified to the execution unit via a move data structure, as is depicted in Table 6.2. In order to execute a step in the motion plan, the execution unit first makes sure that the step should be executed by examining the *skip-conditions*. If all of these conditions fail, the step is to be executed and each of the *initial-functions* is invoked. The *motion* is then executed and the status of the motion is determined by testing the *goal-terminating-conditions*, the *position-bound*, the *force-bound*, and by executing a verification motion if necessary. If any of these tests reveals a failure of the motion just executed, an error state is declared and recovery actions are taken if necessary. The motion is then repeated and the process continues until it is determined that the motion successfully met its goal while remaining within the specified bounds.

Table 6.2 The move data structure used to describe a motion.

(move
motion
skip-conditions
initial-functions
final-functions
position-bound
force-bound
vector
goal-terminating-conditions
protectional-terminating-conditions)

Of course, an error might occur if the actual geometry of the parts was not within the bounds specified by the nominal geometric representations. For instance, say we are trying to put a peg into a hole where the radius of the hole turns out to be smaller than the radius of the peg. Upon attempting to put the peg into the hole, an error condition will result and since it is not possible to ever put the peg into the hole, error conditions will continue to result from every recovery effort. Therefore, theoretically the system will continue to cycle forever, so to account for such situations an upper limit could be set on the number or *EDR* cycles allowed. If this limit is reached without successful completion of a motion, an unrecoverable error condition would be

declared, and the execution would halt.

As mentioned earlier, some fine motion strategies used by AMP-CAD can be very error prone if excessively large uncertainties or excessively tight clearances are involved. By setting the upper limit on the number of allowable cycles, the execution may also halt on motions that are executable in theory, but prove to be unexecutable in practice.

To explain the scope of our error detection and recovery abilities, we must first explain the control scheme we used to implement fine motion on our robots. We define a fine motion to be a motion made by a robot toward a specified goal position while maintaining a given force constraint. A fine motion terminates when one of a given set of position or force conditions is satisfied. We implement the fine motions via an inner/outer servo loop system, where the inner servo loop controls the position of the robot and the outer servo loop controls the force being exerted by the robot. Thus the manner in which the outer servo loop zeros out the force error is by converting it into an incremental position command that is added to the commanded position input to the inner servo loop. The force error is converted into an incremental position by modeling the force exerted by the robot with a first-order spring approximation. Fig. 34 depicts our control scheme.

Fig 34: Control diagram of our inner/outer loop system that issues position commands to adjust forces.

One particularly robust way to do error detection and recovery of fine motions is by assuming a particular type of fine motion will be used for a specific application and building a control scheme that automatically accounts for the types of errors that tend to occur in that type of motion for the given application. An example of such a system used for grinding applications was discussed in [28]. Unfortunately a wide variety fine motions are in general required for assembly motion plans and therefore it would not be easy for us to use this type of approach.

Instead, we have developed an execution unit that makes assumptions about the local geometry from the given motion specification (which is assumed to be a sensible specification as was defined by Mason in [42]) and uses additional information given in the plan (e.g. the *force-bound* and *goal-terminating-conditions*) about the motion to be made as a basis to decide whether or not an error has occurred and if so what type of recovery motion should be executed. The intent is that the EDR strategies used by our execution unit should be able to handle errors that occur in the outer force servo loop of our control system and to be able to handle

errors that result when a motion is specified to follow a trajectory that is calculated based on the current force and/or torque exerted by the robot. In short, we assume that the position sensors on the robot are accurate enough and that the position-servoing is done well enough that they will not be the cause of an error in a motion. Recall, of course, that the motion plans are developed assuming control uncertainty which is the result of predictable uncertainty in the position sensors and in the position servo loop. In contrast, we assume, however, that the force/torque sensors and the force-servoing will not necessarily be accurate enough to avoid errors. The reason for this is that the inaccuracies in the force sensor and the force-servo loop tend to be unpredictable and non-linear and thus are not easily modeled as uncertainties to the planning system.

Given the limited application of assembly plans, there is a small number of types of servoing constraints that will be used on an assembly motion and a small number of types of terminating conditions that will be used, and finally the servoing constraints will be used in conjunction with the terminating constraints in a limited number of ways. So, in a sense, motions can be classified into different types and specific recovery motions can be associated with each type of motion. Upon unsuccessful termination of a motion, our execution classifies the type of motion that was executed and uses this in conjunction with the type of failure triggered (e.g. force out of bounds, position out of bounds, etc) to select an appropriate recovery motion.

Therefore we can detect errors of the type that we will be looking for and we will be able to execute a recovery motion (assuming one is called for) each time one of these errors occur. There is no guarantee, however, that the recovery motion will completely eradicate the problem at hand as it is executed under similar conditions as for the original motion, and whatever caused it to terminate unsuccessfully might have the same effect on the recovery motion. For instance, if someone is using a hand drill in the next work cell, thus inducing much electronic noise into the signal sent from the force/torque transducer to the force/torque sensor controller and causing a motion to terminate in an error condition, there is no guarantee that the recovery motion used will be successful when it is executed under the same noise conditions. However if the recovery motion ends unsuccessfully then when the fine motion is repeated it most likely will end unsuccessfully thereby forcing another EDR cycle and giving the robot another chance to recover. At the very worst, the robot would cycle on this motion indefinitely.

In closing we should note that while our error detection and recovery might seem very limited, we have found that, given assembly motion plans that account for predictable uncertainties, our EDR is almost always successful. We have found that in practice, errors other than the ones that we look out for can be avoided by using adequate fixturing, etc. and when an undetectable error occurs it often results in a detectable error, which can then be handled by our EDR. Of course, our EDR is extensible if different types of sensors become available for sensing different types of errors or if different types of motions are required by the assembly motion planner.

7. FINE MOTION STRATEGIES

For many assembly mating operations, orientational uncertainty about the mating axis does not need to be accounted for because these operations mate features that are rotationally

symmetric about the mating axis, as is the case for a round peg and hole. The problem of mating such parts is well studied and a variety of mating strategies have been proposed in [21, 26, 56, 64] and other works. These strategies realistically encompass the range of possibilities for this type of mating operation, which generalizes to the problem of fitting a round peg in a round hole. Therefore the problem of performing insertions when orientational uncertainty can be ignored has essentially been solved.

There are, however, situations in which orientational uncertainty becomes a significant factor; for instance when an assembly operation involves simultaneously mating two or more pairs of features whose mating axes are not aligned, or when the mating features are not rotationally symmetric. This problem remains largely unsolved. The purpose of this section is to describe the strategies that currently exist for dealing with this type of assembly operation and to characterize the limits of these strategies. An approach for developing additional strategies along these lines will also be proposed.

The fine motion strategies that have been presented so far in the literature rely on the use of force/torque guided robotic motion. The idea behind any force/torque guided fine-motion strategy is that parts are brought into contact in such a way that the force and torque exerted on the grasped part can be measured and through some mechanism be converted into a motion that will align the grasped part with the fixtured part. These motions often terminate when some activity in the force/torque signal is sensed. In a sense then, the information contained in the force/torque signal is used in combination with motion to reduce the uncertainty so that the operation may proceed. Thus two major factors that limit the applicability and robustness of any force/torque guided fine-motion strategy are the amount of information contained in an ideal force/torque signal and the signal to noise ratio of an actual force/torque signal.

We believe that when there is orientational uncertainty about the mating axis to contend with, there may be situations in which there is no way to obtain enough information from even a noiseless force/torque signal to reduce the uncertainty by the amount required. Even if there are strategies that work in the ideal case, they may be such that they will not be robust due to the noise in and the limited resolution of the force/torque signal. Of course, such strategies could be executed under error detection and recovery, but if a noise-sensitive strategy does not work on the first attempt, there is no guarantee it will work on the second or any other attempt since it is being executed under the same noise and resolution conditions.

For these reasons we believe that it may be ultimately necessary to bring in other types of sensors, such as vision and tactile sensors, in order to be able to handle all assembly operations robustly. Alternatively it might be feasible to use a micromanipulator end-effector for performing noise-sensitive strategies since it seems that fine-motions would be easier to execute using a micromanipulator than a robot manipulator, and therefore noise may not be as much of an issue. Unfortunately there has been very little research done in either of these areas. In the following we will focus entirely on force/torque guided fine-motion strategies that are executed with a standard six degree of freedom robot manipulator.

7.1 Strategies for a Rotationally-Asymmetric Feature Pair

As mentioned in Chapter 2, Caine and Lozano-Perez [6] have developed and thoroughly analyzed a mating strategy that handles orientational uncertainty. However, this strategy is limited to cases where the mating features are rectangular solids. A more general strategy, developed by Strip [58], can handle convex solids, that is, mating features whose cross-sections are convex. Both strategies are similar, but because the latter strategy is the more general of the two, the following discussion will focus on it.

Strip's strategy is carried out in three phases: the initial orientation phase, the final orientation phase, and the insertion phase. In order to describe these phases it is necessary to first define a coordinate frame with respect to the mating feature of the grasped part that is to be inserted into the corresponding feature of the fixtured part. The mating feature of the grasped part is referred to as the *peg* and the mating feature of the fixtured part is referred to as the *hole*.

The point of greatest curvature on the curve or polygon bounding the bottom face of the peg is defined to be the *target point*. The point about which forces and torques are measured and applied is referred to as the *point of support*. This point is usually, although not necessarily, near the center of mass of the bottom face of the peg. The Y axis of the coordinate reference frame extends from the point of support through the target point. The Z axis is defined to lie coincident to the outward pointing normal of the bottom face. The X axis is defined so as to create a right-handed orthonormal coordinate frame, with the origin being at the point of support. For an illustration of this see Fig. 35.

Fig 35 Definition of coordinate frame, target point, and point of support.

In the first phase of this strategy, the peg is positioned directly above the hole so that the Z axes of the peg and hole are aligned and the X and Y axes are coincident, to the best of the available knowledge. The peg is then tilted about its X axis so that the target point is the lowest point on the peg. The target point of the peg is then brought to the corresponding target point of the hole with a compliant motion that attempts to maintain zero torques about the Z axis and zero

forces along the X axis of the peg. Of course, there is some uncertainty in the position of the peg with respect to the hole, so in general the peg will be initially brought in contact with the hole at points other than the corresponding target points. This motion will continue, applying force along the Y axis, until a stable equilibrium is reached. By stable equilibrium is meant that the forces and torques being exerted by the peg are perfectly balanced by the forces and torques exerted on the peg by the sides of the hole so there is no net motion of the peg.

During the second or final orientation phase of the strategy, the peg is uprighted so that the Z axes of the peg and of the hole are once again aligned as well as possible given the uncertainties. During this motion a positive force in Y is maintained so that contact between the peg and hole is not broken, a positive force in Z is maintained so as to be pushing the peg further into the hole as the motion continues, and also the zero X force and Z torque are maintained as before. This phase ends, ideally, with the bottom face of the peg completely contained in the hole.

The final phase of the strategy inserts the peg the rest of the way into the hole via the compliant motion proposed and justified in [64]. That is, the peg is pushed in the positive Z direction while attempting to maintain zero forces along and zero torques about the X and Y axes. In this way wedging and jamming situations are avoided.

The success of the overall strategy depends on the success of the first phase. It is possible that the peg might become wedged in the hole before a reasonable alignment may be achieved by the first phase motion. Also if the initial uncertainty is too large the motion could feasibly terminate with one-point contact between the peg and the hole where the peg is stuck in an unaligned position due to friction. However, both these situations will be avoided if the initial uncertainty is relatively small. An exact measure on the allowable uncertainty would depend on the geometry of the hole, the friction between the peg and the hole, the clearance between the peg and the hole, and the size of the peg and hole. A complete analysis of the allowable uncertainty has not been performed but general guidelines of such an analysis are outlined in [58]. In this paper, Strip reports that the strategy was found to be robust when used for inserting a two inch square peg into a corresponding hole with the clearance between the peg and hole being 0.0015 inches and the uncertainty being couple of hundredths of an inch in translation directions and a degree or two in rotational directions. Strip also reported on a passive device that can be used to perform such insertions in [59].

Unfortunately, the applicability of this strategy is rather limited. For instance, imagine a convex but not round plug and socket connector. The plug might be reasonably modeled as a peg and the socket as a hole, only because this strategy requires the peg to be tilted into the hole and the peg only interact with the edges of the hole, it could most likely not be applied in this case because the pins in the socket might interfere with the plug during the mating operation.

As mentioned earlier, Peshkin [47] has developed a general methodology for deriving strategies for arbitrary mating feature geometries. It might prove to be feasible to apply Peshkin's method to those feature geometries where there are no currently known fine-motion strategies, although, as we will explain, we feel that it is probably not possible to do this without further development of the methodology. Peshkin's method is presently limited to the development of

strategies consisting of a single, decision-free compliant motion; his method develops a damping matrix to be used to control this motion. It does not seem likely that a single motion strategy could be used to mate certain feature geometries given normal uncertainties.

For instance, assume the strategy to be developed is to move the rectangular peg depicted in Fig. 36a from the start position to the goal position. Peshkin's method relies on first finding all of the qualitatively distinct contact configurations that could occur while inserting the peg into the hole with a single straight line motion given the uncertainties. From this a compliance matrix is developed that, when used to control the robot, would deflect the insertion motion of the peg in such a way that alignment will be achieved and the forces and torques will remain bounded. An input to the equations used to find this matrix is the force/torque vector that would be exerted on the peg in each qualitatively-distinct realizable contact configurations. For the cases shown in Fig. 36b and Fig. 36c, the force exerted on the peg in both cases is the same but the desired motion is different. Therefore there is no way that the robot could differentiate between the two situations (using force/torque information alone) and thus deduce the correct direction in which to rotate the peg. For this reason it does not seem feasible that Peshkin's method could be used to develop a single-motion strategy for this situation.

However, it might be possible to expand Peshkin's methodology so as to be able to produce multi-step strategies where motions may be added to bring the parts into distinguishable contact configurations. For instance the use of a tilting motion can often lead to distinguishable contact configurations. Of course it may not always be feasible to use tilting motions for this purpose as we explained in the socket and plug connector example given earlier in this chapter.

Of course, while there are feature geometries for which there are no known strategies, it is questionable how important these geometries are in a practical sense. For instance, while no strategy is known for inserting a "U" shaped peg in a hole, it is not clear that such a task would need to be done in any industrial assembly. Also, while most strategies that do exist can handle only a limited amount of uncertainty, it is usually feasible that sensing operations and grasping operations can be used to reduce the uncertainty, particularly in part placement, to a large degree.

7.2 Strategies for Multiple Feature Pairs

Little work has been done toward finding robust strategies for mating multiple pairs of assembly features simultaneously. While it would be nice if there were a general approach, such as Peshkin's, that could be used to develop strategies for these situations, since the limits of such approaches are currently not known it cannot be assumed that the multiple mating feature problem can be solved in this way. Instead a strategy will be discussed that might be used in some of these situations and present experimental evidence and a theoretical justification of the viability of this strategy. The limitations of this strategy will then be discussed an approach to the development and analysis of related strategies will be informally proposed.

When multiple feature pairs must be mated, a general approach may be to tilt the grasped part so that the number of mating feature pairs that come in contact is reduced and/or the areas

Fig 36

A situation where Peshkin's method would fail. In this situation, depicted in panel (a), Peshkin's method would attempt to derive a one-step decisionless fine-motion strategy for the assembly operation depicted. Here the peg is a rectangular solid and the hole is a rectangular solid void. Peshkin's method ideally will produce a control that can differentiate between all contact configurations and always move in the correct direction for each configuration. Panel (b) and (c) depict two contact configurations that would require rotational motions in opposite directions, but the force exerted on the peg is the same in both cases; hence there is no way for the control to differentiate between the two situations using force/torque information alone.

of the potential contact regions are reduced. For example strategy, assume that a grasped part has two round pegs and a fixtured part contains two round holes and the task is to simultaneously insert the two pegs into the two holes.

We will define a coordinate reference frame as depicted in Fig. 37 for the grasped part where the X axis extends through the centers of the bottom faces of the two pegs. The Z axis is parallel to the outward pointing normal to the bottom faces of the two round pegs, and the Y axis is chosen to form a right-handed orthonormal coordinate frame. A coordinate frame is defined similarly for the fixtured part where the X axis extends through the centers of the bottom faces of the two round holes on the fixtured part.

The first step of the proposed strategy begins with the grasped part positioned above the fixtured part with their Z axes aligned and the X and Y axes coincident, to the best of the available knowledge. The grasped part is tilted slightly about its -X axis and then moved along the positive Z axis of the fixtured part until contact between the grasped part and the fixtured part is made.

Fig 37 Definition of the coordinate reference frame of the grasped part.

A compliant motion is then made that attempts to move in the +Y direction while maintaining a constant force along the X axis and a constant torque about the Z axis of the grasped part until a stable equilibrium is reached. By stable equilibrium is meant that the forces and torques exerted by the pegs are balanced by the forces and torques exerted by the sides of the holes. If this motion is successful, it will end with the bottom tip of each peg within their corresponding holes. If there are other mating features on the grasped and fixtured parts this strategy can be used as long as the two round pegs contain the lowest points on the grasped part (the points with the most positive Z values with respect to the hole features).

The next step is quite similar to the second phase of Strip's strategy [58], it is to perform a compliant motion to upright the grasped part so that the Z axes of the parts are again aligned while maintaining constant forces along the X, Y, and Z axes and constant torque about the Z axis. This motion should terminate with the bottom faces of each of the two pegs completely

contained within their respective holes.

The final step is to insert the part using the same compliant motion as was used by Strip in his insertion phase. Note that when there are more than two pegs on the grasped part and if the grasped part could be tilted in the first step in such a way that the lowest points on the part were on the bottom faces of two round pegs (with the coordinate reference frame being defined with respect to these two pegs), then this strategy could be also be used. At the end of the second step all of the bottom faces of each of the pegs should be within their respective holes.

To see why this strategy works, first consider the case where the orientation about the Y and Z axes and the position along the X axis are exactly correct. Then the compliant motion executed in the first step will begin in a contact configuration similar to the one depicted in Fig. 38a. Let us assume that the force exerted by the peg located along the -X axis exerts a force of F^- and that the peg located along the +X axis exerts a force of F^+ . Further assume that the forces are measured relative to the origin of the reference coordinate frame depicted in Fig. 37. Then we note that in Fig. 38a we have

$$F^- \cos \theta^- = F^+ \cos \theta^+ \quad (12)$$

$$F^- \sin \theta^- + F^+ \sin \theta^+ = F_o \quad (13)$$

$$l^- |\sin(\pi - \theta^-)| F^- = l^+ |\sin(\pi - \theta^+)| F^+ \quad (14)$$

where F_o is the force exerted by the robot. Therefore in this case we have a stable equilibrium and the motion terminates immediately.

If this is not the case then either Eq. 12 or Eq. 14 are not satisfied; Eq. 13 is necessarily satisfied because the manipulator is exerting a force on the grasped part along the Y axis. In the event that the pegs are not aligned with the holes along the X direction, then the initial contact between the grasped part and the fixtured part will be similar to that depicted in Fig. 38b and Eq. 12 will not be satisfied. However the manipulator will tend to move in the direction necessary to zero out the net force along the X axis and as a consequence the pegs will come into alignment with the holes along the X axis. The pegs may momentarily loose contacts with the sides of the holes but they should regain contact in a configuration that is closer to having perfect alignment along the X axis.

In the case that the initial orientation about Z is not correct then the motion will initially establish only one point of contact between one or the other peg and hole as depicted in Fig. 38c. Therefore a net moment about the Z axis of the grasped part will exist and the compliant motion will tend to rotate the part in the correct direction necessary to zero out this torque until two point contact is achieved with a point of contact for each peg and hole pair and equilibrium is reached. Hence at the end of this motion the grasped part will be positioned as shown in Fig. 38a.

As the grasped part is uprighted with respect to the fixtured, the contact points will be maintained because the compliant motion attempts to maintain a constant force along the Y direction, and the pegs will stay in the holes because the compliant motion attempts to maintain a constant force in the Z direction. Any minuscule misalignment along the X axis or about the Z axis will be countered as the compliant motion attempts maintain zero X force and Z torque.

Fig 38

Relative placement of contact points. In panel (a), two point contact achieved along the edge of the round holes when the pegs are aligned with the holes along the grasped part's X axis and about its Z axis. In panel (b), two point contact achieved along the edge of the round holes when the pegs are not aligned with the holes along the grasped part's X axis.. In panel (c), one point contact achieved when the grasped part is not initially aligned with the fixtured part about the grasped part's Z axis.

Hence, this motion should terminate with the bottom face of each peg in the corresponding hole. Note that if the grasped part has more than two pegs, then due to the initial tilt imposed upon the grasped part, the remaining pegs do not come in contact with the fixtured part until the very last portion of this move whereupon the alignment along the X axis and about the Z axis should already be achieved so the bottom faces of the additional pegs should land within their respective holes.

There may still be some misalignment about the X and Y axes of the grasped part, but the final motion of the strategy will attempt to zero out the X forces and torques and the Y forces and torques created by such a misalignment so the X and Y orientation will be corrected as the insertion motion proceeds.

Experimental verification of this strategy was conducted in our lab where a grasped part with four round pegs was inserted into a fixtured part with four round holes. This was done by first deriving a motion plan for the operation and then executing the motion plan on a Puma 760 robot using AMP-CAD's execution unit. The clearance between each peg and hole was 0.001 inches. We are able to mate these parts robustly when the initial uncertainty is relatively small (on the order of hundredths of an inch of translational uncertainty and a few degrees of rotational uncertainty).

For a clarification of how the steps discussed above apply to the case where there are four pegs to be mated, we present Fig. 39. Here we show a grasped part and a fixtured part with a geometry similar to the ones we used in the assembly. In the figure, the top frame of each panel depicts the assembly as viewed along the X axis of the fixtured part, while the bottom frame depicts a view along the Z axis. In the bottom frame, only the lowest points on the peg are shown, that is the points with the largest Z value with respect to the fixtured part coordinate frame.

Of course this strategy is applicable only in situations where the underlying assumptions are met; namely that the grasped part could be tilted such that the lowest point on the part are on the bottom faces of two round holes. Secondly, it is assumed the grasped part contains only pegs and the fixtured part contains only holes. It is also assumed that the two pegs dealt with are of the same length and that the orientational uncertainty about the Y axis is such that the first step ends with the tips of both pegs within their holes. Finally it is assumed that all initial uncertainties are small enough that initial contact between either peg and the fixtured part will occur along the top edge of the peg's mating hole and nowhere else.

While the range of parts this particular strategy could be used for might be very limited, it is easy to see how similar strategies could be developed for parts with different geometries. For instance, if one peg is sufficiently longer than the others, it might be possible to use the strategy discussed in [21] if the peg is rotationally symmetric about its mating axis or the strategy discussed in [58] if the peg is convex in order to insert the longest peg partially into the corresponding hole. If the peg is nonsymmetric, then the grasped part should already be aligned rotationally about the mating axis at the end of this step, else the grasped part could be rotated about the mating axis until the remaining pegs are aligned with their holes. In either case a compliant motion that maintains constant forces and torques along and about the X and Y axes could be used to complete the insertion.

Alternatively if one of the pegs is convex, it might be feasible to tilt the part such that the lowest point on the part is a suitable target point for the application of Strip's strategy. If so it should be possible to carry out Strip's strategy in order to mate the parts in these situations. Similar strategies can be developed for other geometries using the strategies defined for single mating feature pairs as a guide.

Fig 39

Steps involved in mating multiple pegs and holes. In panel (a), the grasped part is aligned with and positioned above the fixtured part to the best of the available knowledge. In panel (b), the grasped part is tilted about the X axis. In panel (c), the grasped part is brought to the fixtured part until contact is established. In panel (d), the motion of the grasped part continues toward the target point until a stable equilibrium has been established. In panel (e), The grasped part is untilted so that the Z axes of the grasped and fixtured part is aligned to the best of the available knowledge. In panel (f), the grasped part is inserted the rest of the way into the fixtured part.

Therefore, an overall approach to dealing with multiple mating feature pairs is to first find strategies that are applicable to different part geometries, then determine the conditions under which each strategy is applicable, and finally develop a set of rules for the selection of an appropriate strategy for a given part geometry. Ultimately it might be necessary to rely on additional forms of sensor input in order to find strategies for every situation that might occur.

8. GEAR BOX ASSEMBLY EXPERIMENTS

In the previous section, assembly operation experiments involving the simultaneous mating of four peg-hole feature pairs were briefly described. This section contains a more detailed description of some experiments that were conducted that involve a gear box assembly. Before discussing these experiments however, we will briefly outline the implementation of AMP-CAD. More details on the implementation can be found in Appendix 2, where we have included the source code for AMP-CAD. References to various modules in this source code will be made throughout the rest of this section, and have been made in the previous sections.

AMP-CAD is written in Common Lisp and C. The Common Lisp part, written in an object oriented style using CLOS (Common Lisp Object System), is used for orchestrating the overall flow of control in accordance with Figs. 1 and 33. CLOS is also used for the representation of parts and assemblies; this representation contains facilities for interfacing with the TWIN solid modeling package, which itself is written in C. Common Lisp is also used for the programming of Stage 2 of Phase 2 (see Fig. 1).

We have used the motion plans generated by AMP-CAD to carry out assemblies on both a Cincinnati Milacron T^3-726 robot and a Puma 762 robot. In the Robot Vision Lab we have a robot-independent interface, written in Common Lisp and C, which allows us to specify motion commands in a robot-independent format. The execution unit discussed in Chapter 6 translates the assembly motion plan into a sequence of motion commands that can be understood by this interface, causes the execution of these commands, and performs error detection and recovery (EDR).

In general, we have found that the plans produced by AMP-CAD are feasible roughly 85 percent of the time. That is to say if we made 100 attempts to execute on a robot the motion plan for a particular assembly, roughly 85 attempts would be successful. The rest of the attempts would fail because of the failure of one of the motions due to the unpredictable aspects of force control [21]. A motion under EDR fails when its execution gets stuck in the endless loop of error detection and recovery; this is more likely to occur when the uncertainties at the beginning of the motion are large in relation to the clearance between the parts.

In the following, the development of motion plans for the parts depicted in Fig. 40, and the subsequent execution of these motion plans will be described. The assembly involves two operations. In the first, the gear depicted in Fig. 40b is mounted onto the gear shaft in the box depicted in Fig. 40c and at the same time it is meshed with the gear in the gear box. The second operation involves putting the lid depicted in Fig. 40a into the gear box. The actual Common Lisp code used to develop a representation of this assembly is given in Table 6. The source code

for the routines invoked by the code listed in Table 6 is itself listed in Appendix 2 as part of the assembly representation code.

(a)

(b)

(c)

Fig 40 The assembly experiment to be conducted involves assembling the gear in (b) with the box in (c) and then putting the lid in (a) onto the box.

Table 6 The Common Lisp code used for creating an assembly representation for the parts involved in the experiment.

```
(setq gear1hole (make-instance 'round-hole :radius 0.63109375 :cheight 0.40625 :chradius

(setq gear1 (make-instance 'spur-gear :radius 1.40234575 :tooth-width1 0.1 :tooth-width2 0.2
:height 0.25 :h .40625 :num-teeth 18 :offset 0.372246))
(setq gear1part (make-instance 'part :features-list '((gear1) (h1))))
(setq gear2hole (make-instance 'round-hole :radius 0.72 :cheight 0.875 :chradius .72 :chheight 0))
(setq gear2 (make-instance 'spur-gear :radius 1.6875 :tooth-width1 0.05 :tooth-width2 0.15 height
0.28125 :h .875 :num-teeth 20 :offset 0.43565054376))
(setq shaft1 (make-instance 'round-peg :radius .62109375 :cheight 3.0 :bradius .62109375 :bheight
0))
(setq shaft2 (make-instance 'round-peg :radius .7 :cheight 1.0 :bradius .7 :bheight 0))
(setq housing (make-instance 'feature :csg '((ppiped 0 0 0 9.125 7.5 3.625)) :additivep '+' :sym 1)))
(setq recess (make-instance 'arb-hole :csg '((ppiped 0 0 0 8.9375 7.3125 3.25)))
(setq box (make-instance 'part :features-list '((housing) (recess (0.125 0.125 0.375 0 0 0)) (gear2
(3.0 3.75 0.375 0 0 0.0)) (gear2hole (3.0 3.75 0.375 0 0 0.0)) (shaft2 (3.0 3.75 0.375 0 0 0.0))
(shaft1 (5.977345 3.75 .375 0 0 0))))))
(setq plate (make-instance 'arb-peg :csg '((ppiped 0.03125 0.03125 0 8.875 7.25 .1)) :sym 4))
(setq hole1 (make-instance 'round-hole :radius .125 :cheight .1 :chradius .125 :chheight 0))
(setq hole2 (make-instance 'round-hole :radius .125 :cheight .1 :chradius .125 :chheight 0))
(setq peg1 (make-instance 'round-peg :radius .124 :cheight .15 :bradius .124 :bheight 0))
(setq peg2 (make-instance 'round-peg :radius .125 :cheight .15 :bradius .125 :bheight 0))
(setq hole3 (make-instance 'round-hole :radius .25 :cheight .75 :chradius .25 :chheight 0))
(setq lidpart (make-instance 'part :features-list '((plate) (hole1 (4.4375 3.125 0 0 0 0)) (hole2
(4.4375 4.125 0 0 0 0))))))
(setq bridge (make-instance 'feature :csg '((ppiped -.375 -.75 -.875 .75 1.5 1.75)) :additivep '+'
:sym 1))
(setq handle (make-instance 'part :features-list '((bridge (4.4375 3.625 1.025 0 0 0)) (hole3
(4.0625 3.625 .65 0 1.5707963268 0)) (peg1 (4.4375 3.125 0 0 0 0)) (peg2 (4.4375 4.125 0 0 0
0))))))
(setq lid (make-instance 'assembly :grasped-part handle :description '((peg1 (into) hole1) (peg2
(into) hole2)) :fixtured-part lidpart :done t))
(setq boxassem (make-instance 'assembly :grasped-part gear1part :fixtured-part box :pose-uncert
'(0.15 0.15 0.0 0.0 0.0 0.175) :description '((gear1 (mesh) gear2) (h1 (onto) shaft))))
(setq gearbox (make-instance 'assembly :grasped-part lid :description '((lid (into 3.125) recess))
:fixtured-part boxassem :pose-uncert '(0.0 0.15 0.0 0.0 0.0 0.0)))
```

After composing the assembly representation for the assembly depicted in Fig. 40, AMP-CAD is called upon to develop a motion plan for the first assembly operation. The assembly

representation is used by Phase 1 to create a path for the assembly operation to follow. It turns out that a simple straight-line path can be used to perform the assembly operation. The source code to develop this path plan is given in Appendix 2; as indicated there, the path is developed by the pathplan program. To develop the path plan the gear box was automatically decomposed into 46 obstacles. Two wedge shaped CSG primitives are used to form each tooth and there were 20 teeth on the gear in the box, so 40 obstacles were needed to represent the gear teeth alone.

Next, all of the uncertainties were taken into account. In this experiment there was .15" of uncertainty in X component and .15" of uncertainty in the Y component of the initial pose of the gear. There was also roughly .175 radians of uncertainty in the initial orientation about the axis of the gear. Due to the manner in which the gear was grasped, that is from a stable position, all other uncertainties were insignificant. These uncertainties were incorporated into the potential field representation of the parts in the first stage of Phase 2, which is implemented by the uncertaintyall program listed in Appendix 2. Using this augmented representation the first point of potential collision was found to be that labeled "1" in Fig. 41.

At this point it was further determined that it would be necessary to find a strategy for the path segment containing the point "1" that will be able to reduce the uncertainty in the X and Y directions in order for the mating operation to proceed. In the second stage of Phase 2 the strategy detailed in [21] (referred to as "peg and hole" in the code given in Appendix 2) was selected for this purpose and it was used to develop a fine motion subplan that effectively aligns the bottom face of the hole in the grasped gear with the top face of the gear shaft. This is done by first attempting to put the gear onto the shaft using a guarded motion that will terminate if the gear instead runs into the shaft. If the gear did indeed run into the shaft then the torque exerted on the gear is measured to find the direction vector in which to slide the gear along in order to align the gear-hole and the shaft axis. This direction vector is tangent to the torque vector in the X-Y plane. By sliding the gear along this vector until a contact force ceases to be sensed between the gear and the shaft, the gear and shaft will effectively be aligned so that a third motion can be made to the end of the path segment containing the point "1" following the successful termination of this sliding motion.

By using this strategy, the X and Y pose uncertainty are reduced to .01" by the time the gear is brought to the end of the path segment containing the point "1", corresponding to the clearance between the shaft and the hole in the grasped gear. Resuming the uncertainty analysis performed in Stage 1 of Phase 2 as indicated in Fig. 33, the next point of potential collision that was found corresponds to the point labeled "2" in Fig. 41. Analysis revealed that the uncertainty in the orientation about the Z axis needs to be reduced when the gear is being moved through the path segment containing the point "2" in order to reach this point.

Therefore a fine motion subplan was synthesized that first moves the gear down the gear shaft until it either reaches the end of the path segment containing "2" or until a contact force is

Fig 41 Two path segments where fine-motion strategies must be used.

created between the two gears. If no contact force is sensed then clearly the two gears are successfully meshed and the fine motion plan is finished. If, however, a contact force is sensed then the two gears are assumed to be clashing and therefore the fine motion subplan dictates that the gear be rotated back and forth about the Z axis until there is no longer any contact force sensed. When this occurs the grasped gear is aligned so that it can be successfully meshed with the fixtured gear, and the final motion in the subplan consists of a motion to the endpoint of the path segment containing "2". The strategy used to develop this subplan is called rotz-search and the source code for it is given in Appendix 2.

Upon further analysis by the first stage of Phase 1, it was determined that no more potential collision points could be found in the remainder of the path. Therefore the entire motion plan consists essentially of the first subplan described appended with the second subplan described

followed by a final point-to-point motion to reach the goal from the point labeled "2". The first motion in each subplan might actually consist of a point-to-point motion followed by a guarded motion depending on how the original path was broken into segments.

We have executed this motion plan many times on both of the robots. In accordance to what was said about our experiments previously, we were usually able to run the assembly operation successfully, but were less likely to be successful when the original uncertainty was large. For instance, a particular problem that arises when the original uncertainty is large is that the second motion in the first motion subplan, the one that slides the grasped gear along the tangent to the torque vector, is inclined to get stuck in an endless loop of error detection and recovery. The reason for this is that the resolution of the force/torque sensor may not be good enough to calculate the torque vector as precisely as necessary when the uncertainty in the X and Y direction at the beginning of this motion is relatively large and the center of the gear is far away from the center of the shaft. This phenomena was explained in detail in [21]. The amount of uncertainty that can be tolerated by this strategy is dependent on factors such as the clearance between the grasped part and the fixtured part and the resolution of the force/torque sensor. For the setup we used, where the clearance between the gear and the shaft was approximately .01" and the resolution of the force/torque sensor was 1/40 foot-pounds, we found that uncertainties under .1 in to .2 in would result in a successful operation.

Another point in the plan that required significant error detection and recovery efforts was the second motion in the second motion subplan, this is the motion that rotates the gear back and forth about the Z axis until contact between the two gears no longer exists so that it can be assumed that the two gears are meshed. This motion often needed to be repeated many times (10 to 20 iterations were not uncommon) because either the force terminating condition was missed and the motion hit a limit (the motion is only allowed to travel about .18 rads in one direction, and if the force constraint is not meant, rotation in the opposite direction is attempted) or a force terminating condition was falsely sensed.

The first situation occurs when the motion moves too rapidly and the force/torque sensor is not sampled during the small amount of time that the gears are meshed during the motion. This is especially common on the Puma robot because the rotational speed of motions can not be controlled by us directly and the result is the compliant rotations executed on that robot are done too fast. The second situation occurs when the force/torque sensor produces a noisy reading or when the grasped gear is raised slightly during the rotation and thus the otherwise clashing teeth loose contact with one another. A noisy reading can be detected by a static and stable reading, a loss of contact is determined by executing a verification motion. In all cases our execution unit was able to correctly detect errors and no recovery motions were necessary other than to repeat the motion. Unlike the prior motion subplan, this motion practically always terminated successfully after 20 cycles or so. Of course this motion is much simpler as it involves a rotation strictly about the Z axis as opposed to the earlier motion which involved a translation along a direction vector somewhere in the X-Y plane.

All other motions in the motion plan were practically always successful as they were all either point-to-point motions or simple guarded motions for which error detection and recovery is simple. Thus the most significant factor in the success of this assembly operation is whether or not the second motion in the motion plan can be executed successfully.

In order to develop an assembly motion plan for the lid into box assembly operation, the assembly representation was first developed as described in Table 6. Next a path was found by Phase 1 for placing the lid into the box; once again the path followed a straight line between the starting point and the ending point. To give the reader an idea of the time required to compose a path plan, first note that in this case the gear box with the second gear placed in it was automatically decomposed into 84 obstacles, 3 of which are negative obstacles; the additional obstacles were required to represent the second gear. It took about 19 minutes to compute the path on a Sun 4 server under a typical load.

While this might seem to be a long time, recall that the complexity of the potential field calculation is linearly related to the number of obstacles in the assembly, and 84 is a relatively large number of obstacles. For the first example discussed, the final path was found in about 41 minutes. The reason why this path plan took so long to develop is that the grasped gear must be moved along the gear shaft for most of the operation and there is only a small clearance between the gear and shaft. Therefore the path had to be computed at a very tight resolution which lead to a large number of points on the path which takes a large amount of compute time. Of course, AMP-CAD is a research tool and not a production system; the implementation has not been optimized to any great extent.

The majority of time it takes to develop a motion plan is usually spent developing the path plan, especially in more complicated situations such as the gear box. The implementation of Stage 1 of Phase 2 takes the next largest amount of time, but it usually only takes a small fraction of the time taken to develop a path plan. Generating the assembly representation and generating the motion plan take very little time in general compared to the time taken by the path-plan program and the uncertainty program.

Due to the manner in which the lid was grasped, the only significant uncertainty was found to be in the Y component of the initial pose. This uncertainty could potentially cause a collision just as the lid is being placed into the entrance of the gear box, as shown in Fig. 42.

Therefore a fine motion subplan must be used for the path segment containing this point; the strategy the subplan is based upon is the well known biased search strategy, which is referred to as biased motion along y axis in Appendix 2. To perform a biased search along the Y axis, the lid is brought close to the potential collision point, moved purposely in the positive Y direction by the amount of its uncertainty, brought down until it is contacting the gear box, and then slid in the negative Y direction until the contact force is no longer sensed. The uncertainty in the Y component of the pose at the end of this search corresponds to the clearance between the lid and box. A final motion to the end of the current path segment is then executed.

Fig 42 One path segment where a fine-motion strategy must be used.

No other potential collisions were found for this assembly operation. Thus the complete motion plan consists of this fine motion subplan followed by a point-to-point motion to move from the ending point of the subplan to the goal point. Of course, there may be a few point-to-point motions before the fine motion subplan depending on how the path was sampled which would determine the starting point of the motion subplan.

Execution of this assembly operation was nearly always successful. Perhaps the only real possibility for failure in the operation occurs if the biased search missed the terminating condition and overshot the entrance to the box, but we have not seen this occur. Fig. 43 shows snapshots of the motion plans being executed.

Fig 43

The first four snapshots show the robot assembling the gear with the gearbox. The last two snapshots show the robot assembling the lid with the box.

9. CONCLUSION AND FUTURE WORK

We have discussed a two-phase planning system that produces motion plans for assembly operations to be carried out by robots that must operate in uncertain environments. In Phase 1 planning the uncertainties are ignored and a path that will be collision free in the ideal case is found for the assembly operation. In Phase 2 the effects of uncertainties are analyzed and the results of this analysis are used to synthesize an assembly motion plan.

Much of our work in Phase 1 path planning lies in the development of methods for dealing with assembly situations. In the first stage of path planning a quick search of free space is performed to find an initial path. In the second stage, collision regions in the initial path are found and eliminated via localized optimization. Capabilities have been added to the potential field representation used by Phase 1 to make it easier to handle parts containing holes. Special measures were taken to avoid problems that occur when developing plans for tightly fitting parts.

To perform uncertainty analysis for Phase 2 planning, the uncertainties are first incorporated into the potential field representation in a manner that allows for accurate modeling of these uncertainties. The path produced by Phase 1 is then analyzed to find segments of the path where fine motions must be used to avoid problems due to uncertainties.

Given the path together with the knowledge of the effects of uncertainty and related information, a motion plan can be generated that will allow the grasped part to more or less follow along the derived path in order to be assembled with the fixtured part. This is done by choosing from a library of strategy one particular strategy for each segment of the path. Each such strategy is expanded into a sequence of motions capable of moving the grasped part through the path segment under the existing uncertainty and contact conditions.

In order to produce a system that does not rely on copious amounts of specialized input information to be produced for each assembly, we have developed a feature-based representation of assemblies and parts. To create a computer representation for a specific assembly, the user specifies only a high-level description of the assembly to the computer. Methods associated with the representation are able to take the high level descriptors supplied by the user and convert them into the input data required by the planner, the most important input being a list of the obstacles and their boundary representations to be used for the potential field representation. By using object-oriented programming techniques we have modularized the representation in such a way that it could easily be tailored to specific assembly domains.

We have used this planning system to produce plans for many assemblies and have carried out these plans using our execution unit. We have found that when executed under the error detection and recovery afforded by our execution unit the plans produced are robust.

Despite the advances we have made, much work remains to be done. Perhaps the most limiting factor in our work is the set of strategies currently available for use in Phase 2 of our system. This set is based primarily on previous work done in the development of fine motion strategies, but this work is by no means complete. For instance, there is no strategy that we know of for inserting a concave solid peg into a corresponding hole. Furthermore while our system is

currently able to deal with parts whose geometry can be modeled by taking the union or difference of convex solids, there are many industrial parts that cannot be represented exactly in this manner. Because preciseness is so important in assembly motion planning, work needs to be done in finding a means to deal with exact representations of these parts as well. Finally, there is much work that remains to be done in the integration of assembly motion planning with other areas of assembly planning before a fully automated assembly system can be developed.

ACKNOWLEDGMENTS

The authors would like to thank Matt Carroll for the many hours of tireless work he has spent helping out with assembly experiments. Thanks are also due to Professor Dave Anderson and the members of the CADLAB at Purdue for allowing us to use their TWIN solid modeling package in this research.

REFERENCES

1. *IMSL Library Reference Manual*, IMSL Inc., 1980.
2. J. Barraquand and J. C. Latombe, "Robot Motion Planning: A Distributed Representation Approach," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1712-1717, 1990.
3. R. A. Brooks, "Symbolic Error Analysis and Robot Planning," *The International Journal of Robotics Research*, vol. 1, no. 4, pp. 29-67, Winter 1982.
4. R. A. Brooks, "Solving the Find-Path Problem by a Good Representation of Free Space," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 13, no. 3, pp. 190-197, 1983.
5. S. J. Buckley, "Planning Compliant Motion Strategies," *The International Journal of Robotics Research*, vol. 8, no. 5, pp. 28-44, October 1989.
6. M. E. Caine, T. Lozano-Perez, and W. P. Seering, "Assembly Strategies for Chamferless Parts," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 472-477, 1989.
7. S. A. Cameron and R. K. Culley, "Determining the Minimum Translational Distance Between Two Convex Polyhedra," *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, pp. 591-596, 1986..
8. J. F. Canny, *The Complexity of Robot Motion Planning*, MIT Press, 1988.
9. J. F. Canny, "On Computability of Fine Motion Plans," *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, pp. 177-182, 1989.
10. T. C. Chang, D. C. Anderson, and O. R. Mitchell, "QTC- An Integrated Design/Manufacturing/Vision Inspection System for Prismatic Parts," *Proceedings of ASME Computers in Engineering Conference*, vol. 1, 1988.
11. C. H. Chen and A. C. Kak, "A Robot Vision System for Recognizing 3-D Objects in Low-Order Polynomial Time," *IEEE Transactions on Systems Man and Cybernetics*, vol. 19, no. 6, pp. 1535-1563, 1989.
12. R. T. Chien, L. Zhang, and B. Zhang, "Planning Collision-Free Paths for a Robotic Arm Amongst Obstacles," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, no. 1, pp. 91-96, 1984.
13. P. G. Comba, "A Procedure for Detecting Intersections of Three-Dimensional Objects," *Journal of the Association for Computing Machinery*, vol. 15, no. 3, pp. 354-366, July 1968.
14. B. R. Donald, "A Search Algorithm for Motion Planning with Six Degrees of Freedom," *Artificial Intelligence*, vol. 31, no. 3, pp. 295-353, 1987.
15. B. R. Donald, "Planning Multi-Step Error Detection and Recovery Strategies," *The International Journal of Robotics Research*, vol. 9, no. 1, pp. 3-60, 1990.
16. H. F. Durrant-Whyte, "Uncertain Geometry in Robotics," *The International Journal of Robotics Research*, vol. 4, no. 1, pp. 23-31, February 1988.

17. M. Erdmann, "Using Backprojections for Fine Motion Planning with Uncertainty," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 19-45, 1986.
18. I. D. Faux and M. J. Pratt, *Computational Geometry for Design and Manufacture*, John Wiley & Sons, 1979.
19. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*, Addison Wesley, 1990.
20. E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, "A Fast Procedure for Computing the Distance Between Objects in Three-Dimensional Space," *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193-203, 1988.
21. S. N. Gottschlich and A. C. Kak, "A Dynamic Approach to High-Precision Parts Mating," *IEEE Transactions on Systems Man and Cybernetics*, vol. 19, no. 4, 1989.
22. M. Herman, "Fast, Three-Dimensional, Collision-Free Motion Planning," *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1056-1069, 1986.
23. B. K. P. Horn, "Extended Gaussian Image," *Proceedings of IEEE*, vol. 72, pp. 1671-1686, 1984.
24. Y. K. Hwang and N. Ahuja, "Path Planning Using a Potential Field Representation," University of Illinois at Urbana-Champaign Tech. Report UILU-ENG-88-2251, 1988.
25. Y. K. Hwang and N. Ahuja, "Path Planning Using a Potential Field Representation," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pp. 569-575, 1989.
26. H. Inuoe, "Force Feedback in Precise Assembly Tasks," in *Artificial Intelligence: An MIT Perspective*, vol. 2, pp. 219-241, 1981.
27. R. R. Karinithi and D. S. Nau, "Using a Feature Algebra for Reasoning about Geometric Feature Interactions," *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1219-1224, 1989.
28. K. Kashiwagi, K. Ono, E. Izumi, T. Kurenma, and K. Yamada, "Force Controlled Robot for Grinding," *Proceedings of IEEE International Workshop on Intelligent Robots and Systems*, pp. 1001-1006, 1990.
29. S. E. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.
30. H. Kenner, *Geodesic Math and How to Use it*, Berkeley CA: University of California Press, 1976.
31. O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 90-98, 1986.
32. A. J. Koivo, *Fundamentals for Control of Robotic Manipulators*, Wiley, 1989.
33. A. Koutsou, "A Geometric Reasoning System for Moving an Object While Maintaining Contact with Others," *Proceedings of ACM Symposium on Computational Geometry*, 1985.
34. B. Krogh, *A Generalized Potential Field Approach to Obstacle Avoidance Control*, SME Conference Proceedings Robotics Research: The Next Five Years and Beyond, 1984.

35. J. C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, 1991.
36. C. Laugier and P. Theveneau, "Planning Sensor Based Motions for Part-mating using Geometric Reasoning Techniques," *Proceedings of European Conference on Artificial Intelligence*, 1986.
37. C. Laugier, "Planning Fine Motion Strategies by Reasoning in the Contact Space," *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 653-659, 1989.
38. K. Lee and G. Andrews, "Inference of the positions of components in an assembly," *Computer Aided Design*, vol. 17, no. 1, pp. 20-25, Jan/Feb 1985.
39. M. C. Lin and J. F. Canny, "A Fast Algorithm for Incremental Distance Calculation," *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1008-1014, 1991.
40. T. Lozano-Perez, M.T. Mason, and R.H. Taylor, "Automatic Synthesis of Fine-Motion Strategies for Robots," *The International J. of Robotics Research*, vol. 4, pp. 3-24, 1985.
41. D. G. Luenberger, *Linear and Nonlinear Programming*, Addison-Wesley, 1984.
42. M. T. Mason, "Compliant Motion," in *Robot Motion*, pp. 305-322, MIT Press, 1982.
43. N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., 1980.
44. C. O'Dunlaing and C. K. Yap, "A Retraction Method for Planning the Motion of a Disc," *Journal of Algorithms*, vol. 6, pp. 104-111, 1982.
45. M. Orlowski, "The Computation of the Distance Between Polyhedra in 3-Space," *Presented SIAM Conference on Geometric Modeling and Robotics*, no. 4, 1985.
46. J. Pearl, *Heuristics*, Addison-Wesley Publishing Co., 1984.
47. M. A. Peshkin, "Programmed Compliance for Error Corrective Assembly," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 4, pp. 473-482, 1990.
48. R. Popplestone, Y. Liu, and R. Weiss, "A Group Theoretic Approach to Assembly Planning," *AI Magazine*, vol. 11, no. 1, pp. 82-97, 1990.
49. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numeric Recipes in C*, Cambridge University Press, 1988.
50. A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *Computing Surveys*, vol. 12, no. 4, pp. 437-464, 1980.
51. A. A. G. Requicha, "Toward a Theory of Geometric Tolerancing," *The International Journal of Robotics Research*, vol. 2, no. 3, pp. 45-60, 1983.
52. E. Rimon and D. E. Koditschek, "Exact Robot Navigation in Geometrically Complicated but Topologically Simple Spaces," *Proceedings of IEEE International Conference on Robotics and Control*, pp. 1937-1942, 1990.
53. A. C. Sanderson, L. S. Homem de Mello, and H. Zhang, "Assembly Sequence Planning," *AI Magazine*, vol. 11, no. 1, pp. 62-81, 1990.

54. J. T. Schwartz and M. Sharir, and D. S. Seltzer, "Use of Sensory Information for Improved Robot Learning," *Communications of Pure and Applied Mathematics*, vol. 1/Fundamentals, pp. 424-431, 1981.
55. S. Simunovic, "Force Information in Assembly Processes," *Proceedings of Fifth International Symposium on Industrial Robotics*, pp. 415-427, 1975.
56. S. Simunovic, "An Information Approach to Parts Mating," Sc.D. thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, 1979.
57. R. C. Smith and P. Cheeseman, "On the representation and estimation of spatial uncertainty," *The International Journal of Robotics Research*, vol. 5, no. 4, pp. 56-68, Winter 1986.
58. D. R. Strip, "Technology for Robotic Mechanical Assembly: Force-Directed Insertions," *AT&T Technical Journal*, vol. 67, no. 2, pp. 23-34, 1988.
59. D. R. Strip, "A Passive Mechanism for Insertion of Convex Pegs," *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, pp. 242-248, 1989.
60. S. F. Su and C. S. G. Lee, "Uncertainty Manipulation and Propagation and Verification of Applicability of Actions in Assembly Tasks," *Proceedings of IEEE International Conference on Robotics and Automation*, 1991.
61. F. Thomas and C. Torras, "A Group-Theoretic Approach to the Computation of Symbolic Part Relations," *IEEE Transactions on Robotics and Automation*, vol. 4, no. 6, pp. 622-634, 1988.
62. C. M. Tilley, "The Representation of Mechanical Assemblies in a Computer-Aided Design Environment," Masters Thesis, Purdue University, 1988.
63. R. Volpe and P. Khosla, "Manipulator Control with Superquadric Artificial Potential Functions: Theory and Experiments," *IEEE Systems, Man, and Cybernetics*, vol. 20, no. 6, pp. 1423-1436, 1990.
64. D.E. Whitney, "Quasi-Static Assembly of Compliantly Supported Rigid Parts," in *Robot Motion*, pp. 439-471, MIT Press, 1981.

Appendix 1

FINDING LOCAL MINIMA ON A SURFACE OR CURVE

In order to calculate the value of the function $\min_dist$, defined in Section 4, we must be able to find the set of points Y , that are a locally minimum distance to a surface or a curve. In the following we will start out by assuming there is only one such point (as there is for many cases), and then consider the case where there are multiple points. If there is only one such point, this point is also a global minimum.

Therefore the task we will consider first is how to find the closest point, call it y^* , on either a surface or an edge to an arbitrary point in space x . In the following we will first treat the problem of finding the point y^* on an arbitrary surface and we will then treat the problem of finding the point y^* on an edge.

For some simple surfaces, such as a plane or a cylinder, the surface equation $g(x)$ can be defined in such a way that its absolute value is the distance between x and the surface. For instance, for a plane

$$g(x) = ax + by + cz + d \quad (12)$$

where (x, y, z) are the coordinates of the point x in Cartesian space. Furthermore, here the outward pointing unit normal vector to the plane has the coordinates (a, b, c) . Therefore, assuming $\alpha = g(x)$, $y^* = (x - \alpha a, y - \alpha b, z - \alpha c)^T$. Perhaps a more intuitive way to express Eq. (12) is in a standard vector form

$$g(\vec{r}) = \vec{r} \cdot \hat{n} - d \quad (12')$$

where $\vec{r} = x$, $\hat{n} = (a, b, c)$, and d in Eq. (12) is equal to $-d$ in Eq. (12'). For another example, the distance between x and a right circular cylinder aligned with the Z axis is

$$g(x) = \sqrt{x^2 + y^2} - r \quad (13)$$

where r is the radius of the cylinder. Here the outward pointing normal at y^* is $(x, y, 0)$, so that if $\alpha = g(x)$, $y^* = (\pm \alpha \hat{x}, \pm \alpha \hat{y}, z)^T$ where $(\hat{x}, \hat{y}, 0)$ is the normalized normal vector.

In a more general sense the distance between x and an arbitrary surface is

$$d(y^*) = \sqrt{(x-i)^2 + (y-j)^2 + (z-k)^2} \quad (14)$$

where (i, j, k) are the coordinates of y^* . The problem then is how to find y^* .

It turns out that this is a nonlinear constrained optimization problem where the idea is to minimize the objective function given by Eq. (14) for a point that is constrained to be on some arbitrary surface described by the differentiable equation $g(y)$. As has been proved in many texts, such as [41] the point y^* can be found by first forming the Lagrangian associated with the constrained problem $l(y, \lambda) = d(y) + \lambda g(y)$ and then solving the equations

$$\begin{aligned} \nabla_y l(y, \lambda) &= 0 \\ \nabla_\lambda l(y, \lambda) &= g(y) = 0 \end{aligned} \quad (15)$$

simultaneously for $y^* = y$ and λ . This will usually involve finding the solution of four nonlinear equations which cannot always be done algebraically. Therefore we may need to solve the four equations resulting from Eq. (15) numerically. There are practical methods for doing this, for instance the IMSL function NEQNF, which is explained in [1], can be used to find the solution

y^* . Note that while there is only one point that minimizes the distance function globally on the surface, there may be a set of points that minimize the distance function locally. If this is the case Eq. (15) will have multiple solutions, each solution corresponding to a local minimum. In this case we must find all of the solutions and return the set of points, Y , yielded by the solutions, to the *min_dist* procedure for further examination.

Of course, using such a general method to find an accurate solution can be computationally prohibitive. Therefore it is desirable to simplify Eq. (15) as much as possible and perhaps apply a more specific technique to find the solution. For instance, if $g(y)$ is polynomial, Eq. (15) can sometimes be rearranged and combined into a single polynomial equation of one variable. In this case, we can solve for the roots of the polynomial to find the solutions for the first variable, and this can be used to find the values of the other variables, finally yielding the set of points that are local minimums of the Euclidean distance to x on the given surface. There are many texts that deal with both the theory and implementation of various methods of solving for the roots of a polynomial, as for instance [49].

Now we will present an example of solving Eq. (15) numerically for the surface equation

$$g(y) = j^2 - k \quad (16)$$

where here (i,j,k) are the coordinates of the point y . Rearranging and combining Eq. (15) for this surface equation we get:

$$j^3 + (.5-z)j - .5y = 0 \quad (17a)$$

$$k = j^2 \quad (17b)$$

$$i = x \quad (17c)$$

where (i,j,k) are the coordinates of y^* and (x,y,z) are the coordinates of x . Hence, we can solve for the real roots of the (single variable) polynomial in Eq. (17a) for any given point x to get a value for j and from this solve Eq. (17b) for k . We can, in this case, solve Eq. (17c) trivially for the value of i .

Note that since Eq. (17a) is a third order polynomial, it will have either one or three real solutions. If there is only one real solution, this solution will correspond to the point (i,j,k) that minimizes the distance function globally. If there are three real solutions then they will correspond to points that minimize the distance function locally on the surface. In this case the *min_dist* procedure needs to examine each point further. Table 7 lists several different values of x and the resulting value for y found via a numerical solution. Fig. 44 shows a plot of some of these solutions. Fig 44 also shows, for the point $x = (0,2,6)$, the local minima corresponding to the two additional roots of Eq. (17a).

The procedure for finding the closest point, y^* , on an edge to the point x is very similar to that of finding y^* for a surface. Once again Eq. (15) can be used, where in this case the constraining function $g(y)$ is the equation of the curve containing the edge, to find four nonlinear equations which can be solved simultaneously to yield the values of $y^* = y$ and λ . Alternatively two constraining functions could be used, these being the equations of the surfaces intersecting at this edge, to find five nonlinear equations which must be solved simultaneously. Of course, as in the case of the surface, there may be a set of points that locally minimize the Euclidean

$\mathbf{x}$	$\mathbf{y^*}$	$\mathbf{d(y^*)}$
(0.00,2.00,6.00)	(0.00,2.43,5.90)	0.44
"	(0.00,-0.18,0.03)	6.38
"	(0.00,-2.25,5.06)	4.35
(0.00,0.50,1.00)	(0.00,0.88,0.77)	0.44
(0.00,-0.50,1.00)	(0.00,-0.88,0.77)	0.94
(0.00,8.00,6.00)	(0.00,2.65,7.01)	5.45
"	(0.00,-0.83,0.69)	9.13
"	(0.00,-1.81,3.28)	10.18
(0.00,4.00,3.00)	(0.00,1.89,3.57)	2.19

Table 7: Results of solving Eq. (15) numerically for the surface equation $g(\mathbf{y}) = j^2 - k$.

Fig 44: Three $\mathbf{x}, \mathbf{y^*}$ pairs are shown for the surface $g(\mathbf{y}) = j^2 - k$.

distance to $\mathbf{x}$, and we would have to return the entire set of these to the *min_dist* procedure for further examination.

Appendix 2: AMP-CAD SOURCE CODE

The following is a listing of the AMP-CAD source code. This code is written in C and Common Lisp. It consists of five major components: The assembly representation which defines the classes of objects used in the representation and includes the various methods associated with these classes, the path planner which performs Phase 1 computations, the uncertainty analyzer which performs Phase 2- Stage 1 computations, the plan supervisor that invokes the path planner and the uncertainty analyzer and then performs Phase 2- Stage 2 operations to produce a motion plan, and finally the execution unit the executes the motions in the plan and performs error detection and recovery. The assembly representation is written in Common Lisp using CLOS, the plan supervisor and the execution unit are written in Common Lisp, and the path planner and uncertainty analyzer are written in C.

The assembly representation is listed first here, where the source code listing starts out with the definitions for the classes of objects used and then the source for the methods associated with these classes are listed. We have only included the source code for the methods associated with a few example feature subclasses. For a further description of the assembly representation see Section 3.

Next, the source code for the path planner is listed. The path planner takes as input an object file, an obstacles file, and a profile file. These files are all essentially created by the supervisor using the information and methods associated with the assembly representation. The object file describes the object, the obstacles file describes the obstacles, and the profile file indicates the starting point, the ending point, obj_size, and the maximal extent of the object. The path planner dumps its final results into a file called slp.out, although intermediate results are also dumped into files for analysis purposes. Details on the path planner were discussed in Sections 4 and 5.

The uncertainty analyzer, which shares some routines with the path planner, takes as input the object, obstacles, unobject, unobstacles files, as well as the slp.out file and an uncertainty file detailing the initial pose and control uncertainty associated with the assembly operation. Except for the slp.out these files are similarly generated by the plan supervisor using the assembly representation. The files unobject and unobstacles contain the mmc representations of the object and obstacles. Further details on the uncertainty analyzer were given in Section 6.

The plan supervisor, for which the source code is listed following the uncertainty analyzer source code, uses the results from the uncertainty analyzer to develop an assembly motion plan. The motion plan is represented as a list of move data structures, where each move data structure represents either a point-to-point motion or a force/guided motion. All point-to-point motions are specified using a force terminating condition along the axis of motion as an extra margin of safety. That is, a point-to-point motion will terminate prematurely if an unexpectedly large force is suddenly exerted on the grasped part, as might happen if some mistake has been made in the placement of the fixtured part. Thus the point-to-point motion will undergo the same type of error detection and recovery as a standard force/torque guided motion. This is possible because point-to-point motions are a subset of force/torque guided motions. Details on plan synthesis

were given in Section 6.

Finally, the source code for the execution unit is included. The execution unit takes as input a move data structure. It then issues this motion command to the robot, waits for the robot to complete the motion, and then examines the value of the force/torque vector and the position/orientation vector of the robot to determine if an error has occurred. If an error has occurred the appropriate recovery strategy is executed and the execution unit recursively calls itself to reattempt the given motion. The execution unit is explained further in Section 6.

REPRESENT

```

;; Assembly representation
;;
;; This file contains definitions for various
;; classes of objects used for assem
;; representation
;; Note: Attributes ending in "-h" are hidden
;; from
;; user -- user is not to specify a value for
;; them.
;; Instead they have procedural attachments
;; invoked
;; if needed.
(defclass feature ()
  ((sign :initarg :sign :accessor feature-sign)
   ; 1 if additive, -1 if subtractive
   ; 2 if additive but not an inserter
   (sym :initarg :sym :accessor feature-sym)
   (tree :initarg :tree :accessor feature-tree)
   (constraint-angle :initarg :constraint-angle
    :initform nil
    :accessor feature-constraint-angle)
   (constraint-numsyms :initarg
    :constraint-numsyms :initform nil
    :accessor feature-constraint-numsyms)
   (part :initarg :part :accessor feature-part)
   (csg :initarg :csg :accessor feature-csg)
   (nom :initarg :nom-h :initform nil :accessor
    feature-nom-h)
   (mmc :initarg :mmc-h :initform nil :accessor
    feature-mmc-h)))

(defclass gear (feature)
  ((radius :initarg :radius :accessor
    gear-radius)
   (num-teeth :initarg :num-teeth :accessor
    gear-num-teeth)))

(defclass spur-gear (gear)
  ((x1 :initarg :x1 :accessor spur-gear-x1) ;
   see notes
   (x2 :initarg :x2 :accessor spur-gear-x2) ;
   see notes
   (y :initarg :y :accessor spur-gear-y) ; see
   notes
   (h :initarg :h :accessor spur-gear-h) ;
   height
   (offset :initarg :offset :accessor
    spur-gear-offset)
   ; between teeth
   (facets :initarg :facets :initform 12
    :accessor spur-gear-facets)))

(defclass bolt (feature)
  ((radius :initarg :radius :accessor
    bolt-radius)
   (cheight :initarg :cheight :accessor
    bolt-cheight)
   (bradius :initarg :bradius :accessor
    bolt-bradius)
   (bheight :initarg :bheight :accessor
    bolt-bheight)
   (thread-space :initarg :thread-space
    :accessor bolt-thread-space)
   (thread-depth :initarg :thread-depth
    :accessor bolt-thread-depth)
   (thread-thick :initarg :thread-thick
    :accessor bolt-thread-thick)
   (num-thread :initarg :num-thread :accessor
    bolt-num-thread)))

(defclass hex-head (feature)
  ((height :initarg :height :accessor
    hex-head-height)
   (radius :initarg :radius :accessor
    hex-head-radius)))

(defclass square-head (feature)
  ((height :initarg :height :accessor
    square-head-height)
   (radius :initarg :radius :accessor
    square-head-radius)))

(defclass nut-hole (hole)
  ((radius :initarg :radius :accessor
    nut-hole-radius)
   (thread-space :initarg :thread-space
    :accessor nut-hole-thread-space)
   (thread-depth :initarg :thread-depth
    :accessor nut-hole-thread-depth)
   (thread-thick :initarg :thread-thick
    :accessor nut-hole-thread-thick)
   (oheight :initarg :oheight :accessor
    nut-hole-oheight)
   (cheight :initarg :cheight :accessor
    nut-hole-cheight)
   (cradius :initarg :cradius :accessor
    nut-hole-cradius)))

(defclass nut (feature)
  ((height :initarg :height :accessor
    nut-height)
   (radius :initarg :radius :accessor
    nut-radius)))

(defclass hex-nut (nut) ())

(defclass square-nut (nut) ())

(defclass peg (feature)
  ((oheight :initarg :oheight :accessor
    peg-oheight)))

;; Simple cylinder possibly with bevel
(defclass round-peg (peg)

```



```

((radius :initarg :radius :accessor
 round-peg-radius)
 (cheight :initarg :cheight :accessor
 round-peg-cheight)
 (bradius :initarg :bradius :accessor
 round-peg-bradius)
 (bheight :initarg :bheight :accessor
 round-peg-bheight)))

;; Peg composed of cylinders and cones
(defclass composite-round-peg (peg)
  ((parts :initarg :parts
    :accessor composite-round-peg-parts)))

;; Peg made up by n-sided regular polygon
(defclass nsided-peg (peg)
  ((radius :initarg :radius :accessor
    nsided-peg-radius)
   (height :initarg :height :accessor
    nsided-peg-height)
   (n :initarg :n :accessor nsided-peg-n)))

;; Arbitrary shaped peg
(defclass arb-peg (peg)
  ((parts :initarg :parts :accessor
    arb-peg-parts)))

(defclass threaded-peg (peg)
  ((radius :initarg :radius :accessor
    threaded-peg-radius)
   (thread-space :initarg :thread-space
    :accessor threaded-peg-thread-space)
   (thread-depth :initarg :thread-depth
    :accessor threaded-peg-thread-depth)
   (thread-thick :initarg :thread-thick
    :accessor threaded-peg-thread-thick)
   (bheight :initarg :bheight
    :accessor threaded-peg-bheight)
   (bradius :initarg :bradius
    :accessor threaded-peg-bradius)))

(defclass hole (feature)
  ((oheight :initarg :oheight :accessor
    hole-oheight)))

(defclass threaded-hole (hole)
  ((radius :initarg :radius :accessor
    threaded-hole-radius)
   (thread-space :initarg :thread-space
    :accessor threaded-hole-thread-space)
   (thread-depth :initarg :thread-depth
    :accessor threaded-hole-thread-depth)
   (thread-thick :initarg :thread-thick
    :accessor threaded-hole-thread-thick)
   (cheight :initarg :cheight
    :accessor threaded-hole-cheight)
   (height :initarg :height
    :accessor threaded-hole-height)
   (cradius :initarg :cradius
    :accessor threaded-hole-cradius)))

;; Simple cylinder possibly with bevel
(defclass round-hole (hole)
  ((radius :initarg :radius
    :accessor round-hole-radius)
   (cheight :initarg :cheight
    :accessor round-hole-cheight)
   (chradius :initarg :chradius
    :accessor round-hole-chradius)
   (chheight :initarg :chheight
    :accessor round-hole-chheight)))

;; Hole composed of cylinders and cones
(defclass composite-round-hole (hole)
  ((parts :initarg :parts
    :accessor composite-round-hole-parts)))

;; Hole made up by n-sided regular polygon
(defclass nsided-hole (hole)
  ((radius :initarg :radius :accessor
    nsided-hole-radius)
   (height :initarg :height :accessor
    nsided-hole-height)
   (n :initarg :n :accessor nsided-hole-n)))

;; Arbitrary shaped hole
(defclass arb-hole (hole)
  ((parts :initarg :parts :accessor
    arb-hole-parts)))

(defclass flexible (feature) ())

;; Simple spring
(defclass spring (flexible)
  ((compress :initarg :compress :accessor
    spring-compress)))

;; Simple helical spring
(defclass helical-spring (spring)
  ((radius :initarg :radius :accessor
    helical-spring-radius)
   (cheight :initarg :cheight :accessor
    helical-spring-cheight)))

(defclass part ()
  ((components :initarg :components :accessor
    part-components)
   (pose :initarg :pose :initform nil :accessor
    part-pose)
   (nom :initarg :nom-h :initform nil :accessor
    part-nom-h)
   (mmc :initarg :mmc-h :initform nil :accessor
    part-mmc-h)
   (csg :initarg :csg-h :initform nil :accessor
    part-csg-h)
   (center :initarg :center :initform nil
    :accessor part-center)
   (sym :initarg :sym-h :initform nil :accessor
    part-sym-h)
   (constraint-feature :initarg
    :constraint-feature-h
    :initform nil
    :accessor part-constraint-feature-h)
   (uncert :initarg :uncert :initform nil
    :accessor part-uncert)))

(defclass articulated-part (part)
  ((graph :initarg :graph :initform nil
    :accessor articulated-part-graph)
   (csg-graph :initarg :csg-graph
    :accessor articulated-part-csg-graph)
   (csg-joint-pose :initarg :csg-joint-pose
    :initform nil
    :accessor articulated-part-csg-joint-pose)
   (nom-graph :initarg :nom-graph-h :initform nil
    :accessor articulated-part-nom-graph-h)
   (mmc-graph :initarg :mmc-graph-h :initform nil
    :accessor articulated-part-mmc-graph-h)
   (brep-joint-pose :initarg :brep-joint-pose
    :initform nil
    :accessor
    articulated-part-brep-joint-pose)))

(defclass assembly ()
  ((grasped-part :initarg :grasped-part
    :accessor assembly-grasped-part))

```

```

(fixture-part :initarg :fixture-part
  :accessor assembly-fixture-part)
(description :initarg :description
  :accessor assembly-description)
(left-features :initarg :left-features
  :accessor assembly-left-features)
(right-features :initarg :right-features
  :accessor assembly-right-features)
(connection :initarg :connection
  :accessor assembly-connection)
(left-pose-start :initarg :left-pose-start
  :initform nil
  :accessor
    assembly-left-pose-start-h)
(left-pose-end :initarg :left-pose-end
  :initform nil
  :accessor assembly-left-pose-end-h)
(right-pose :initform nil :initarg :right-pose
  :accessor assembly-right-pose)
(numsyms :initarg :numsyms :initform nil
  :accessor assembly-numsyms)
(nom :initarg :nom-h :initform nil
  :accessor assembly-nom-h)
(mmc :initarg :mmc-h :initform nil
  :accessor assembly-mmc-h)
(csg :initarg :csg-h :initform nil
  :accessor assembly-csg-h)
(control-uncert :initarg :control-uncert
  :initform nil
  :accessor assembly-control-uncert)
(pose-uncert :initarg :pose-uncert :initform
  nil
  :accessor assembly-pose-uncert)
(done :initarg :done :initform nil
  :accessor assembly-done)
(center :initarg :center :initform nil
  :accessor assembly-center)
(grasp-plan :initarg :grasp-plan
  :accessor assembly-grasp-plan)
(fixture-plan :initarg :fixture-plan
  :initform nil
  :accessor assembly-fixture-plan)))

(defclass twinobj ()
  ((nom :initarg :nom :accessor twinobj-nom)
   (mmc :initarg :mmc :accessor twinobj-mmc)
   (csg :initarg :csg :accessor twinobj-csg)
   (movable-surfaces :initarg :movable-surfaces
     :initform nil :accessor
       twinobj-movable-surfaces)
   (aux :initarg :aux :initform nil :accessor
     twinobj-aux)
   (feature :initarg :feature :initform nil
     :accessor twinobj-feature)
   (center :initarg :center :initform nil
     :accessor twinobj-center)
   (operator :initarg :operator
     :accessor twinobj-operator)))

(defclass cone (twinobj)
  ((base :initarg :base :accessor cone-base)
   (axis :initarg :axis :accessor cone-axis)
   (bottomradius :initarg :bottomradius
     :accessor cone-bottomradius)
   (topradius :initarg :topradius :accessor
     cone-topradius)
   (facets :initarg :facets :accessor
     cone-facets)))

(defclass cylinder (twinobj)
  ((base :initarg :base :accessor cylinder-base)
   (axis :initarg :axis :accessor cylinder-axis)
   (radius :initarg :radius :accessor
     cylinder-radius)

```

```

   (facets :initarg :facets :accessor
     cylinder-facets)))

(defclass piped (twinobj)
  ((p :initarg :p :accessor piped-p)
   (e1 :initarg :e1 :accessor piped-e1)
   (e2 :initarg :e2 :accessor piped-e2)
   (e3 :initarg :e3 :accessor piped-e3)))

(defclass wedge (twinobj)
  ((p :initarg :p :accessor wedge-p)
   (e1 :initarg :e1 :accessor wedge-e1)
   (e2 :initarg :e2 :accessor wedge-e2)
   (e3 :initarg :e3 :accessor wedge-e3)))

(defclass composite (twinobj)
  ((components :initarg :components
    :accessor composite-components)))

(defclass attachment ()
  ((grasped :initarg :grasped
    :accessor attachment-grasped)
   (fixture :initarg :fixture
    :accessor attachment-fixture)
   (offset :initarg :offset :initform 0
    :accessor attachment-offset)
   (mate :initarg :mate :accessor
    attachment-mate)))

(defclass into (attachment) ())

(defclass thru (attachment) ())

(defclass onto (attachment) ())

(defclass threaded (attachment) ())

(defclass lineup (attachment) ())

(defclass mesh (attachment) ())

(defclass joint ()
  ((fixed :initarg :fixed :accessor joint-fixed)
   (mobile :initarg :mobile :accessor
    joint-mobile)
   (position :initarg :position :accessor
    joint-position)
   (offset :initarg :offset :accessor
    joint-offset)
   (initial-position :initarg :initial-position
    :accessor joint-initial-position)
   (initial-uncertainty :initarg
    :initial-uncertainty
    :accessor joint-initial-uncertainty)
   (final-position :initarg :final-position
    :accessor joint-final-position)
   (bounds :initarg :bounds :accessor
    joint-bounds)
   (fixed-feature :initarg :fixed-feature
    :accessor joint-fixed-feature)
   (mobile-feature :initarg :mobile-feature
    :accessor joint-mobile-feature)
   (transform :initarg :transform :accessor
    joint-transform)))

(defclass revolute (joint) ())

(defclass prismatic (joint) ())

(defvar *2PI* 6.28318530718)
(defvar *PI* 3.14159265359)
(defvar *facets* 24)
;; This file contains the various procedural
;; attachments

```

```
;; for attributes of the feature class. See Chap
;; 3.1 and
;; 9.2 of thesis.
```

```
;; feature- Initializes a generic feature.
(defmethod initialize-instance :after ((instance
  feature)
  &key tree sign)
  (build-parts instance tree)
  (if (null sign) (setf (feature-sign instance)
    2)))
```

```
;; feature-nom- If needed and not set build brep
;; from csg tree
;; and return nom realization of brep, if set
;; return.
```

```
(defun feature-nom (feature)
  (let ((nom (feature-nom-h feature)) (mmc))
    (cond ((null nom)
      (multiple-value-setq (nom mmc)
        (build-csg (feature-csg feature)))
      (setf (feature-nom-h feature) nom)
      (setf (feature-mmc-h feature) mmc))
      nom)))
```

```
;; feature-mmc- If needed and not set build brep
;; from csg tree
;; and return mmc realization of brep, if set
;; return.
```

```
(defun feature-mmc (feature)
  (let ((nom) (mmc))
    (cond ((null (feature-nom-h feature))
      (multiple-value-setq (nom mmc)
        (build-csg (feature-csg feature)))
      (setf (feature-nom-h feature) nom)
      (setf (feature-mmc-h feature) mmc) mmc)
      (t (feature-mmc-h feature)))))
```

```
;; This file contains the various procedural
;; attachments for
;; attributes of the part class. See Chap 3.2 and
;; 9.1 of thesis.
```

```
;; initialize-articulated-part- Instantiate a
;; joint data
;; structure for each member of the graph list.
;; The list of these joints is csg-graph.
```

```
(defmethod initialize-instance
  :after ((instance articulated-part) &key
    graph)
  (let ((bounds) (ffeat) (mfeat) (fpfixed)
    (fpmobile) (joint)
    (csg-graph) (final) (initial) (offset))
    (dolist (item graph)
      ;; Type= revolute, prismatic
      ;; ffeat= joining feature on fixed part
      ;; mfeat= joining feature on mobile part
      ;; initial= initial position of joint
      ;; final= final position of joint
      ;; offset= assume into, onto relation w
      ;; joining
      ;; features, offset same as in into,
      ;; onto
      ;; fixed= link assumed to be fixed
      ;; mobile= link assumed to move w r t fixed
      ;; link
      ;; bounds= bounds on position of joint
      (setf type (x item)) (setf ffeat (y item))
      (setf mfeat (z item))
      (setf offset (d item)) (setf bounds (e item))
      (setf initial (r item)) (setf final (seventh
        item))
      (setf fixed (feature-part ffeat))
```

```
(setf mobile (feature-part mfeat))
(setf fpfixed (get-position fixed ffeat))
(setf fpmobile (get-position mobile mfeat))
(setf joint
  (make-instance type :fixed fixed
    :mobile mobile
    :bounds bounds
    :fixed-feature ffeat
    :mobile-feature mfeat
    :position initial
    :offset offset
    :initial-position
      (get-nom initial)
    :initial-uncertainty
      (get-vunc initial)
    :final-position final
    :transform
      '(, (make-xform fpfixed)
        , (make-xform
          '(0 0 ,offset 0 0 0))
        , (invert-xform
          (make-xform fpmobile))))))
  (if (null csg-graph) (setf csg-graph '(
    joint))
    (setf csg-graph '(,@csg-graph ,joint))))
  (setf (articulated-part-csg-graph instance)
    csg-graph)))
```

```
;; initialize-part- Set feature-part attrib on
;; each feature to
;; point to part. Xform csg primitives on each
;; feature to
;; their position on the part.
```

```
(defmethod initialize-instance :after ((instance
  part)
  &key components pose uncert)
  (let ((obs) (item) (newcsg) (nom) (mmc))
    ;; For each component feature
    (dolist (item components)
      ;; Set up pointers on feature back to part
      (setf (feature-part (x item)) instance)
      ;; For each csg primitive on the feature
      (dolist (obs (feature-csg (x item)))
        ;; Set pointer on primitive back to feature
        (setf (twinobj-feature obs) (x item))
        ;; If feature has a transform specified,
        ;; transform primitive to this position.
        (cond ((y item)
          (setf (twinobj-nom obs)
            (twin-try
              '(xform ,@ (y item)
                , (twinobj-nom obs))))
          ;; If mmc in feature, xform it too.
          (if (not (null (twinobj-mmc obs)))
            (setf (twinobj-mmc obs)
              (twin-try
                '(xform ,@ (y item)
                  , (twinobj-mmc obs))))))))))
```

```
;; part-csg- If part-csg set return, else compute,
;; fill slot,
;; and return. This is an if-needed
;; function.
```

```
(defun part-csg (part)
  (let ((csg (part-csg-h part)))
    ;; If part is articulated, and position has
    ;; changed,
    ;; csg rep has changed, recalculate.
    (cond ((or (null csg) (changed-pose-csg
      part))
      (setf csg (build-part-csg part))
      (setf (part-csg-h part) csg)))
    csg))
```

```

;; part-nom- If part-nom set return, else compute,
;; fill slot,
;; and return. This is an if-needed
;; function.
(defun part-nom (part)
  (let ((nom (part-nom-h part)) (mmc))
    ;; If part is jointed and position has
    ;; changed since
    ;; last time nom was calculated
    ;; recalculate. Calculate
    ;; mmc while at it, doesn't really take
    ;; much extra
    ;; time, and if need nom, probably need mmc
    ;; next.
    (cond ((or (null nom) (changed-pose-brep
      part)))
      (multiple-value-setq (nom mmc)
        (build-part-brep part))
      (setf (part-nom-h part) nom)
      (setf (part-mmc-h part) mmc))) nom))

;; part-mmc- If part-mmc set return, else compute,
;; fill slot,
;; and return. This is an if-needed
;; function.
(defun part-mmc (part)
  (let ((nom) (mmc))
    ;; If mmc is null it may mean that no
    ;; tolerances specified,
    ;; so mmc not computed. By checking
    ;; part-nom, we'll see if
    ;; part-mmc has been considered yet. If not
    ;; calculated both
    ;; nom and mmc.
    (cond ((or (null (part-nom-h part))
      (changed-pose-brep part))
      (multiple-value-setq (nom mmc)
        (build-part-brep part))
      (setf (part-nom-h part) nom)
      (setf (part-mmc-h part) mmc) mmc)
      (t (part-mmc-h part))))))

;; articulated-part-nom-graph- If set return. Else
;; compute
;; compute this intermediate representation,
;; fill slot
;; and return. Nom-graph is a graph of the
;; brep with
;; joint position left unspecified. If all
;; joints
;; positions are specified, can build a brep
;; from this.
;; This is an if-needed function.
(defun articulated-part-nom-graph (part)
  (let ((nom (articulated-part-nom-graph-h part))
    (mmc)
    (link1-nom) (link2-nom) (link1-mmc) (link2-mmc)
    (flag nil))
    (cond ((null nom)
      (dolist (item (articulated-part-csg-graph
        part))
        ;; Get nom and mmc of fixed link
        (setf link1-nom (part-nom (joint-fixed
          item)))
        (setf link1-mmc (part-mmc (joint-fixed
          item)))
        ;; Get nom and mmc of mobile link
        (setf link2-nom (part-nom (joint-mobile
          item)))
        (setf link2-mmc (part-mmc (joint-mobile
          item)))
        ;; Keep track if mmc has actually been
        ;; specified for any link. If not,

```

```

        ;; articulated-part-mmc not needed.
        ;; Else, specify.
        (if link1-mmc (setq flag t)
          (setq link1-mmc link1-nom))
        (if link2-mmc (setq flag t)
          (setq link2-mmc link2-nom))
        ;; If not first node on nom-graph,
        ;; create nom-graph list.
        ;; Else append to end of list.
        (cond (nom
          (setf nom '(.@nom (link1-nom
            item link2-nom)))
          (setf mmc '(.@mmc (link1-mmc
            item link2-mmc))))
          (t
            (setf nom '(.link1-nom item
              link2-nom)))
            (setf mmc '(.link1-mmc
              item link2-mmc))))))
      (setf (articulated-part-nom-graph-h part)
        nom)
      ;; If mmc needed, specify. Else forget
      ;; about.
      (if flag (setf
        (articulated-part-mmc-graph-h part)
        mmc)
        (setf (articulated-part-mmc-graph-h part)
          nil))))
    nom))

;; articulated-part-mmc-graph- Since mmc-graph
;; will be nil
;; if no tolerances have been specified or if
;; mmc-graph has
;; not been set, check nom-graph to see if the
;; former is
;; case and not the latter. If necessary,
;; calculate mmc-graph
;; (and nom-graph).
(defun articulated-part-mmc-graph (part)
  (let ((nom) (mmc) (flag nil)
    (link1-nom) (link2-nom) (link1-mmc) (link2-mmc))
    (cond ((null (articulated-part-nom-graph-h
      part))
      (dolist (item (articulated-part-csg-graph
        part))
        (setf link1-nom
          (part-nom (joint-fixed item)))
        (setf link1-mmc
          (part-mmc (joint-fixed item)))
        (setf link2-nom
          (part-nom (joint-mobile item)))
        (setf link2-mmc
          (part-mmc (joint-mobile item)))
        (if link1-mmc
          (setf flag t)
          (setf link1-mmc link1-nom))
        (if link2-mmc
          (setf flag t)
          (setf link2-mmc link2-nom))
        (cond (nom
          (setf nom '(.@nom (link1-nom
            item link2-nom)))
          (setf mmc '(.@mmc (link1-mmc
            item link2-mmc))))
          (t
            (setf nom '(.link1-nom
              item link2-nom)))
            (setf mmc '(.link1-mmc
              item link2-mmc))))))
      (setf (articulated-part-nom-graph-h part)
        nom)
      (if flag (setf

```

```

(articulated-part-mmc-graph-h part) mmc)
(setf (articulated-part-mmc-graph-h part)
  nil)))
(t (articulated-part-mmc-graph-h part))))

;; Build nom brep for part, build mmc only if
;; necessary.
(defgeneric build-part-brep (instance part))

(defmethod build-part-brep ((part
  articulated-part))
  (let ((item) (transform) (nom) (mmc nil)
    (last-transform (part-pose part)) (poselist nil)
    (link) (nomg (articulated-part-nom-graph part))
    (mmcg (articulated-part-mmc-graph part)))
    (if (null last-transform)
      (setq last-transform '(0 0 0 0 0 0)))
      (setq last-transform (make-xform
        last-transform))
      ;; Set nom to nomrep for fixed link on
      ;; articulated part
      (setq nom (x (x nomg)))
      (setq nom (twin-try '(copy ,nom)))
      (cond (mmcg (setq mmc (x (x mmcg)))
        (setq mmc (twin-try '(copy ,mmc)))))
      ;; Calc xform and rep for rest of links in
      ;; graph.
      (dolist (item nomg)
        (setq link (z item))
        (setq transform (find-joint-transform (y
          item)))
        (setq transform
          (mmmult last-transform transform))
        (setq link (twin-try '(copy ,link)))
        (setq link (twin-try
          '(xform ,@ (lift-pose transform) ,link)))
        (if poselist
          (setq poselist
            (cons poselist (joint-position (y item))))
          (setq poselist '(, (joint-position (y
            item))))
          (setq nom (twin-try '(combine ,nom +
            ,link))))
          ;; If mmc rep needed, do the same for mmc.
          (if mmcg
            (dolist (item mmcg)
              (setq link (z item))
              (setq transform (find-joint-transform (y
                item)))
              (setq transform
                (mmmult last-transform transform))
              (setq link (twin-try '(copy ,link)))
              (setq link (twin-try
                '(xform
                  ,@ (lift-pose transform) ,link)))
              (setq mmc (twin-try '(combine ,mmc +
                ,link))))
              ;; Record position of joints when brep
              ;; calculated
              (setf (articulated-part-brep-joint-pose
                part) poselist)
              (values nom mmc)))

      (defmethod build-part-brep ((part part))
        (let ((item) (nom) (feat) (comps)
          (part-components part)
          (pose (part-pose part)) (mmc))
          (dolist (item comps)
            ;; Next line only necessary because twin not
            ;; robust
            (if (typep (x item) 'spur-gear)
              (build-teeth (x item))))

            ;; If any feature has mmc rep, must build part
            ;; mmc rep
            (cond ((apply 'myor (mapcar
              (lambda (x)
                (feature-mmc (x x))) comps))
              (setq mmc (feature-mmc (x (x comps))))
              (if (null mmc)
                (setq mmc (feature-nom (x (x comps))))
                (if (not (null pose))
                  (setq mmc (twin-try '(xform ,@pose ,mmc))))
              ;; For each feature on part
              (dolist (item (cdr comps))
                (setq feat (feature-mmc (x item)))
                (if (null feat)
                  (setq feat (feature-nom (x item)))
                  ;; Xform to parts current pose if not
                  ;; at 0
                  (if (not (null pose))
                    (setq feat (twin-try
                      '(xform ,@pose ,feat))))
                  ;; Add or subtract from current brep
                  (if (> (feature-sign (x item)) 0)
                    (setq mmc (twin-try
                      '(combine ,mmc + ,feat)))
                    (setq mmc (twin-try
                      '(combine ,mmc - ,feat))))))
                  (setq nom (feature-nom (x (x comps))))
                  (if (not (null pose)) (setq nom (twin-try
                    '(xform ,@pose ,nom))))
                  ;; For each feature
                  (dolist (item (cdr comps))
                    (setq feat (feature-nom (x item)))
                    ;; Transform if necessary
                    (if (not (null pose))
                      (setq feat (twin-try '(xform ,@pose
                        ,feat))))
                    ;; Add or subtract as needed.
                    (if (> (feature-sign (x item)) 0)
                      (setq nom (twin-try '(combine ,nom + ,feat)))
                      (setq nom (twin-try '(combine ,nom -
                        ,feat))))
                    (values nom mmc)))

                    build-teeth- Each tooth should be a composite
                    ;; primitive,
                    ;; which would be established by init-spur-gear
                    ;; routine.
                    ;; Unfortunately since twin cannot always combine
                    ;; two
                    ;; wedges making up tooth into a single convex
                    ;; object,
                    ;; we leave wedge primitives for csg rep which is
                    ;; needed if the gear is an obstacle. If the gear
                    ;; is an
                    ;; object, we combine wedges into teeth, they
                    ;; won't
                    ;; result in a convex tooth, but this is not
                    ;; necessary
                    ;; for the object. If we don't do this first,
                    ;; twin
                    ;; has trouble building up the brep of the gear.
                    (defun build-teeth (gear)
                      (let ((csg (feature-csg gear)) (tooth) (list)
                        (composite))
                        (setq list '(, (x csg)))
                        (setq csg (cdr csg))
                        (dotimes (i (gear-num-teeth gear))
                          (setq tooth (list (x csg) (y csg)))
                          (setq composite (build-composite tooth '+))
                          (setf (twinobj-feature composite) gear)
                          (setq list (append list '(, composite))))
                        (setq csg (cddr csg)))

                        build-teeth- Each tooth should be a composite
                        ;; primitive,
                        ;; which would be established by init-spur-gear
                        ;; routine.
                        ;; Unfortunately since twin cannot always combine
                        ;; two
                        ;; wedges making up tooth into a single convex
                        ;; object,
                        ;; we leave wedge primitives for csg rep which is
                        ;; needed if the gear is an obstacle. If the gear
                        ;; is an
                        ;; object, we combine wedges into teeth, they
                        ;; won't
                        ;; result in a convex tooth, but this is not
                        ;; necessary
                        ;; for the object. If we don't do this first,
                        ;; twin
                        ;; has trouble building up the brep of the gear.
                        (defun build-teeth (gear)
                          (let ((csg (feature-csg gear)) (tooth) (list)
                            (composite))
                            (setq list '(, (x csg)))
                            (setq csg (cdr csg))
                            (dotimes (i (gear-num-teeth gear))
                              (setq tooth (list (x csg) (y csg)))
                              (setq composite (build-composite tooth '+))
                              (setf (twinobj-feature composite) gear)
                              (setq list (append list '(, composite))))
                            (setq csg (cddr csg)))


```

```

(setf (feature-csg gear) list)))

(defgeneric build-part-csg (instance part))
;; build-articulated-part-csg- Create list of all
;; link csg
;; lists on part. Xform each prim. in all but
;; fixed link.
(defmethod build-part-csg ((part
  articulated-part))
  (let ((item) (link) (transform) (csg)
    (last-transform (part-pose part)) (poselist
      nil))
    (if (null last-transform)
      (setf last-transform '(0 0 0 0 0))
      (setf link (joint-fixed
        (x (articulated-part-csg-graph part))))
      (setf (part-pose link) last-transform)
      (setf last-transform (make-xform
        last-transform))
      (setf csg (build-part-csg link))
      (dolist (item (articulated-part-csg-graph
        part))
        (setf link (joint-mobile item))
        (setf transform (find-joint-transform
          item))
        (setf transform
          (mmult last-transform transform))
        (setf (part-pose link) (lift-pose
          transform))
        (setf link (build-part-csg link))
        (setf last-transform transform)
        (if poselist
          (setf poselist
            (cons poselist (joint-position item)))
          (setf poselist '(, (joint-position item))))
        (setf csg '(, @csg , @link)))
      ;; Record position of joints when csg tree
      ;; calculated.
      (setf (articulated-part-csg-joint-pose
        part) poselist)
      csg))

;; build-part-csg- Create list of each comp feat
;; csg list.
;; Xform primitives if part is not at origin
(defmethod build-part-csg ((part part))
  (let ((item) (feat) (csg) (list nil) (sitem)
    (obj) (ilist))
    (last) (comps (part-components part))
    (pose (part-pose part))
    (dolist (item comps)
      (setf feat (feature-csg (x item)))
      (setf ilist nil)
      (dolist (sitem feat)
        (setf obj (copy-twinobj sitem))
        (setf ilist (mappend ilist (list
          obj))))
      (cond ((> (feature-sign (x item)) 0)
        (setf last ilist)
        (setf list (mappend list ilist))
        (t (setf list (insert-into list last
          ilist))))
      (cond (pose
        (dolist (item list)
          (setf (twinobj-nom item)
            (twin-try
              '(xform ,@pose ,(twinobj-nom item))))
          (if (twinobj-mmcc item)
            (setf (twinobj-mmcc item)
              (twin-try '(xform ,@pose
                ,(twinobj-mmcc
                  item))))))))))

```

```

list))

(defun insert-into (master point list)
  (let ((newlist))
    (dolist (m master)
      (if newlist (setf newlist '(, @newlist
        ,m))
      (setf newlist '(,m)))
      (cond ((equal m (car point))
        (dolist (l list)
          (setf newlist '(, @newlist ,l))))))
    newlist))

;; changed-pose-csg- Check to see if the joint pos
;; of the part
;; have been changed since the last time a csg
;; rep was created.
(defun changed-pose-csg (part)
  (let ((list) (newlist))
    (cond ((typep part 'articulated-part)
      (setf list (articulated-part-csg-graph
        part))
      (setf newlist (list (joint-position (x
        list))))
      (dolist (item (cdr list))
        (setf newlist (cons newlist
          (joint-position item))))
      (not (equal newlist
        (articulated-part-csg-joint-pose part))))
    (t nil)))

;; changed-pose-brep- Check to see if joint pose
;; of the part have
;; been changed since the last time a brep rep
;; was created.
(defun changed-pose-brep (part)
  (let ((list) (newlist))
    (cond ((typep part 'articulated-part)
      (setf list (articulated-part-csg-graph
        part))
      (setf newlist (list (joint-position (x
        list))))
      (dolist (item (cdr list))
        (setf newlist
          (cons newlist (joint-position item))))
      (not (equal newlist
        (articulated-part-brep-joint-pose part))))
    (t nil)))

;; This file contains the various procedural
;; attachments for attributes of the assembly
;; class. See Chap 3.3 of thesis for details.
;;
;; Upon instantiation of an assembly, do certain
;; things. Build left-features, right-features,
;; and connection lists. Set up uncertainties.
(defmethod initialize-instance :after ((assem
  assembly)
  &key grasped-part
    fixtured-part description
    left-pose-start
    left-pose-end
    right-pose-nom mmcc
    control-uncert
    pose-uncert done
    center-grasp-plan
    fixtured-plan)
  (let ((app nil) (afp nil) (con nil) (item))
    (dolist (item description)
      (setf app (append (list (x item)) app))
      (setf afp (append (list (z item)) afp))
      (setf plist '(, (x (y item)) :grasped , (x item)

```

```

      :fixtured ,(z item)
      :mate ,asem ,@(cdr (y item))))
(setq con (append
  (list (apply 'make-instance plist)
    con)))

(setf (assembly-left-features asem) agp)
(setf (assembly-right-features asem) afp)
(setf (assembly-connection asem) con))
(if (null right-pose)
  (setf (assembly-right-pose asem) '(0 0 0 0 0 0)))
(if (null pose-uncert)
  (setf (assembly-pose-uncert asem) '(0 0 0 0 0 0)))
(if (null control-uncert)
  (setf (assembly-control-uncert asem)
    '(0 0 0 0 0 0 0)))

;; assembly-csg- If csg is set, return, else
;; compute
;; and return.
(defun assembly-csg (asem)
  (let ((csg (assembly-csg-h asem)))
    (cond ((null csg)
      (setq csg (build-asem-csg asem))
      (setf (assembly-csg-h asem) csg))
      csg))

;; assembly-nom- If nom brep set, return, else
;; compute
;; and return.
(defun assembly-nom (asem)
  (let ((nom (assembly-nom-h asem)) (mmc))
    (cond ((null nom)
      (multiple-value-setq (nom mmc)
        (build-asem-brep asem))
      (setf (assembly-nom-h asem) nom)
      (setf (assembly-mmc-h asem) mmc))
      nom))

;; assembly-mmc- If mmc brep is set, return. If
;; nom is set,
;; then since nom and mmc are built
;; simultaneously,
;; return nil because there are no tol uncerts or
;; mmc would be set already. Else build nom and
;; mmc
;; and return mmc.
(defun assembly-mmc (asem)
  (let ((nom) (mmc))
    (cond ((null (assembly-nom-h asem))
      (multiple-value-setq
        (nom mmc) (build-asem-brep asem))
      (setf (assembly-nom-h asem) nom)
      (setf (assembly-mmc-h asem) mmc) mmc)
      (t (assembly-mmc-h asem)))))

;; build-asem-brep- Develops brep of the
;; assembly. May
;; not be necessary for some applications. Mmc
;; rep
;; and nominal rep build simultaneously.
(defun build-asem-brep (mate)
  (let ((item nil) (list nil) (nom1 nil) (nom2
    nil) (c1 nil)
    (c2 nil) (mmc1 nil) (mmc2 nil)
    (gp (assembly-grasped-part mate))
    (fp (assembly-fixtured-part mate)) (nom) (mmc))
    (cond ((typep gp 'articulated-part)
      (dolist (item (articulated-part-csg-graph
        gp))
        (setf (joint-position item)
          (joint-final-position item))))
      ((typep fp 'articulated-part)
        (dolist (item (articulated-part-csg-graph
          fp))
          (setf (joint-position item)
            (joint-final-position item))))
        (setf csg1 (part-csg part)))
        ((typep part 'part) (setf csg1 (part-csg
          part)))
        (t (setf csg1 (assembly-csg part))))
      (setf pose (lift-pose (x (assembly-left-pose-end
        mate))))
      (dolist (item csg1)
        (setf obj (copy-twinobj item)))

```

```

      (joint-final-position item)))
      (setf nom1 (part-nom gp))
      (setf mmc1 (part-mmc gp))
      ((typep gp 'part)
        (setf nom1 (part-nom gp))
        (setf mmc1 (part-mmc gp)))
        (t (setf nom1 (assembly-nom gp))
          (setf mmc1 (assembly-mmc gp))))
          (cond ((typep fp 'articulated-part)
            (dolist (item (articulated-part-csg-graph
              fp))
              (setf (joint-position item)
                (joint-final-position item)))
              (setf nom2 (part-nom fp))
              (setf mmc2 (part-mmc fp)))
              ((typep fp 'part)
                (setf nom2 (part-nom fp))
                (setf mmc2 (part-mmc fp)))
                (t (setf nom2 (assembly-nom fp))
                  (setf mmc2 (assembly-mmc fp))))
                  (setf item (x (assembly-left-pose-end
                    mate)))
                    (setf c1 (twin-try '(copy ,nom1)))
                    (setf c2 (twin-try '(copy ,nom2)))
                    (if (not (null item))
                      (setf c1 (twin-try '(xform ,@(lift-pose item)
                        ,c1))))
                      (setf nom (twin-try '(combine ,c2 + ,c1)))
                      (setf (assembly-center mate) (twin-try '(c
                        ,nom)))
                      (cond ((or mmc1 mmc2)
                        (if (null mmc1) (setf mmc1 nom1))
                        (if (null mmc2) (setf mmc2 nom2))
                        (setf c1 (twin-try '(copy ,mmc1)))
                        (setf c2 (twin-try '(copy ,mmc2)))
                        (if (not (null item))
                          (setf c1 (twin-try
                            '(xform ,@(lift-pose item) ,c1))))
                          (setf mmc (twin-try '(combine ,c2 +
                            ,c1))))
                          (t (setf mmc nil))
                          (values nom mmc))))

;; build-asem-csg- Develops csg of assembly from
;; csg
;; reps of component parts.
(defun build-asem-csg (mate)
  (let ((item nil) (list nil) (obj) (part) (csg1)
    (csg2) (pose))
    (setf part (assembly-grasped-part mate))
    (cond ((typep part 'articulated-part)
      (dolist (item (articulated-part-csg-graph
        part))
        (setf (joint-position item)
          (joint-final-position item)))
      (setf csg1 (part-csg part)))
      ((typep part 'part) (setf csg1 (part-csg
        part)))
      (t (setf csg1 (assembly-csg part))))
      (setf part (assembly-fixtured-part mate))
      (cond ((typep part 'articulated-part)
        (dolist (item (articulated-part-csg-graph
          part))
          (setf (joint-position item)
            (joint-final-position item)))
          (setf csg2 (part-csg part)))
          ((typep part 'part) (setf csg2 (part-csg
            part)))
          (t (setf csg2 (assembly-csg part))))
          (setf pose (lift-pose (x (assembly-left-pose-end
            mate))))
          (dolist (item csg1)
            (setf obj (copy-twinobj item)))

```

```

(setf (twinobj-nom obj)
  (twin-try '(xform ,@pose ,(twinobj-nom
    obj))))
(if (twinobj-mmc obj)
  (setf (twinobj-mmc obj)
    (twin-try
      '(xform ,@pose ,(twinobj-mmc obj))))
  (setq list (mappend list (list obj))))
(setq pose (assembly-right-pose mate))
(dolist (item csg2)
  (setq obj (copy-twinobj item))
  (setf (twinobj-nom obj)
    (twin-try '(xform ,@pose ,(twinobj-nom
      obj))))
  (if (twinobj-mmc obj)
    (setf (twinobj-mmc obj)
      (twin-try
        '(xform ,@pose ,(twinobj-mmc obj))))
    (setq list (append list '(,obj))))
  list))

;; mappend- If list exist, append, else create.
(defun mappend (list item)
  (if (null list) (append list item)))

;; If set, return, else compute and return.
(defun assembly-left-pose-start (assem)
  (let ((pose (assembly-left-pose-start-h assem)))
    (cond ((null pose)
      (setq pose (hind-mating-start assem))
      (setf (assembly-left-pose-start-h assem)
        pose)))
    pose))

;; If set, return, else compute and return.
(defun assembly-left-pose-end (assem)
  (let ((pose (assembly-left-pose-end-h assem)))
    (cond ((null pose)
      (setq pose (endpt-loc assem))
      (setf (assembly-left-pose-end-h assem)
        pose)))
    pose))

;; find-mating-start- Find a starting position for
;; the
;; grasped part in an assembly.
(defun find-mating-start (assem)
  (let ((item) (height 0) (theight) (con nil)
    (tp1f1) (mate))
    (dolist (item (assembly-connection assem))
      (setq theight (- (+ (length-z-axis
        (attachment-featured item))
        *SAFE-HEIGHT*))
        (attachment-offset item)))
      (cond ((> theight height)
        (setq height theight)
        (setq con item))))
      (setq mate (attachment-mate con))
      (setq tp1f1 (make-xform (get-position
        (assembly-grasped-part mate)
        (attachment-grasped con))))
      (xmult (x (assembly-left-pose-end mate))
        tp1f1 (make-xform '(0 0 height 0 0 0))
        (invert-xform tp1f1))))

(defvar *SAFE-HEIGHT* 0.55)

;;
;; This file contains routines for creating a
;; toleranced csg rep.
;;

```

```

;; build-cylinder- Builds the brep of a cylinder
;; and sets
;; up a data struct to represent the
;; primitive.
(defun build-cylinder (start height radius oper)
  (let ((cyl) (nom) (mmc))
    ;; Take nominal values of height and radius
    (setq cyl (make-instance 'cylinder
      :base '(0 0 ,(get-nom start))
      :axis '(0 0 ,(get-nom height))
      :radius (get-nom radius)
      :facets *facets*))
    (setf (twinobj-operator cyl) oper)
    (setq nom (twin-try '(cylinder
      ,@(cylinder-base cyl)
      ,@(cylinder-axis cyl)
      ,@(cylinder-radius cyl)
      ,*facets*)))
    (setf (twinobj-nom cyl) nom)
    ;; Check to see if tolerances are specified
    (cond ((or (listp start) (listp height)
      (listp radius))
      ;; If so build mmc rep accordingly
      (if (equal oper '-')
        (setq mmc
          (twin-try
            '(cylinder 0 0 ,(get-max start)
              0 0 ,(get-min height)
              ,(get-min radius)
              ,*facets*)))
          (setq mmc
            (twin-try
              '(cylinder 0 0 ,(get-min start)
                0 0 ,(get-max height)
                ,(get-max radius)
                ,*facets*)))
            (setf (twinobj-mmc cyl) mmc))
          (t (setf (twinobj-mmc cyl) nil)))
        cyl))

    ;; build-flexible-cylinder- Builds a cylinder,
    ;; flags the top
    ;; surface of the cylinder as being movable.
    (defun build-flexible-cylinder (start ht compress
      rad oper)
      (let ((cyl (build-cylinder start ht rad oper)))
        (setf (twinobj-movable-surfaces cyl)
          '(,(*facets* ,compress))) cyl))

    ;; build-cone- Builds cone primitive, w/ ptr to
    ;; brep.
    (defun build-cone (start height bradius tradius
      oper)
      (let ((cone) (nom) (mmc))
        ;; Take nominal value of everything
        (setq cone (make-instance 'cone
          :base '(0 0 ,(get-nom start))
          :axis '(0 0 ,(get-nom height))
          :bottomradius (get-nom bradius)
          :topradius (get-nom tradius)
          :facets *facets*))
        (setf (twinobj-operator cone) oper)
        (setq nom (twin-try '(cone ,@(cone-base
          cone)
          ,@(cone-axis cone)
          ,@(cone-bottomradius cone)
          ,@(cone-topradius cone)
          ,*facets*)))
        (setf (twinobj-nom cone) nom)
        ;; Build mmc rep if any tolerances are
        ;; specified
        (cond ((or (listp start) (listp height)
          (listp bradius)

```



```

(listp radius))
(if (equal oper '-)
  (setq mmc (twin-try '(cone 0 0 ,(get-max
    start)
    0 0 ,(get-min height)
    ,(get-min bradius)
    ,(get-min radius)
    *facets*)))
  (setq mmc (twin-try '(cone 0 0 ,(get-min
    start)
    0 0 ,(get-max height)
    ,(get-max bradius)
    ,(get-max radius)
    *facets*)))
  (self (twinobj-mmc cone) mmc))
(t (self (twinobj-mmc cone) nil))
cone))

;; build-ppiped- Builds a parallelepiped (box),
;; with ptr to brep.
;; Note p could be of form ((xnom xunc) (ynom
;; yunc) z)
;; where xunc is an uncertain value of x. Ppiped
;; is
;; aligned with coordinate axis.
(defun build-ppiped (p x y z oper)
  (let ((box) (nom) (mmc))
    ;; Assume nominal values of everything
    (setq box (make-instance 'ppiped
      :p '(,(xif (x p)) ,(xif (y p))
      ,(xif (z p)))
      :e1 '(,(get-nom x) 0 0)
      :e2 '(0 ,(get-nom y) 0)
      :e3 '(0 0 ,(get-nom z)))
      (self (twinobj-operator box) oper)
      (setq nom (twin-try '(ppiped ,@p
        ,@(ppiped-e1 box)
        ,@(ppiped-e2 box)
        ,@(ppiped-e3 box))))
      (self (twinobj-nom box) nom)
      (cond ((or (listp x) (listp y) (listp z)
        (listp (x p))
        (listp (y p))
        (listp (z p)))
        (if (equal oper '-')
          (setq mmc (twin-try '(ppiped ,(yif (x p))
            ,(yif (y p))
            ,(yif (z p))
            ,(get-min x) 0 0
            0 ,(get-min y) 0
            0 0 ,(get-min z))))
          (setq mmc (twin-try '(ppiped ,(yif (x p))
            ,(yif (y p))
            ,(yif (z p)) ,(get-max x)
            0 0 0 ,(get-max y) 0
            0 0 ,(get-max z))))
          (self (twinobj-mmc box) mmc))
          (t (self (twinobj-mmc box) nil)))
        box))

;; build-arb-ppiped- Builds ppiped not nec.
;; aligned with
;; coord axis, e1,e2, and e3 are axes of ppiped.
;; Note
;; any element of p e1 e2 or e3 can be expressed
;; (el elunc) where el is nominal and elunc is
;; worst case value given uncertainty.
(defun build-arb-ppiped (p e1 e2 e3 oper)
  (let ((box) (nom) (mmc))
    (setq box (make-instance 'ppiped
      :p '(,(xif (x p)) ,(xif (y p))
      ,(xif (z p)))
      :e1 '(,(xif (x e1)) ,(xif (y e1))

```

```

      ,(xif (z e1)))
      :e2 '(,(xif (x e2)) ,(xif (y e2))
      ,(xif (z e2)))
      :e3 '(,(xif (x e3)) ,(xif (y e3))
      ,(xif (z e3))))
    (self (twinobj-operator box) oper)
    (setq nom (twin-try '(ppiped ,@(ppiped-p
      box)
      ,@(ppiped-e1 box)
      ,@(ppiped-e2 box)
      ,@(ppiped-e3 box))))
    (self (twinobj-nom box) nom)
    (cond ((or (listp (x p)) (listp (y p))
      (listp (z p))
      (listp (x e1)) (listp (y e1)) (listp (z e1))
      (listp (x e2)) (listp (y e2)) (listp (z e2))
      (listp (x e3)) (listp (y e3)) (listp (z e3)))
      (setq mmc (twin-try '(ppiped ,(yif (x
        p))
        ,(yif (y p))
        ,(yif (z p)) ,(yif (x e1))
        ,(yif (y e1)) ,(yif (z e1))
        ,(yif (x e2)) ,(yif (y e2))
        ,(yif (z e2)) ,(yif (x e3))
        ,(yif (y e3)) ,(yif (z e3))))
        (self (twinobj-mmc box) mmc))
        (t (self (twinobj-mmc box) nil)))
      box))

;; build-wedge- Builds wedge of arbitrary axes.
;; Parm
;; specified similar to build-arb-ppiped.
(defun build-wedge (p e1 e2 e3 oper)
  (let ((wedge) (nom) (mmc))
    (setq wedge (make-instance 'wedge
      :p '(,(xif (x p)) ,(xif (y p))
      ,(xif (z p)))
      :e1 '(,(xif (x e1)) ,(xif (y e1))
      ,(xif (z e1)))
      :e2 '(,(xif (x e2)) ,(xif (y e2))
      ,(xif (z e2)))
      :e3 '(,(xif (x e3)) ,(xif (y e3))
      ,(xif (z e3))))
    (self (twinobj-operator wedge) oper)
    (self (twinobj-aux wedge) nil)
    (setq nom (twin-try '(wedge ,@(wedge-p
      wedge)
      ,@(wedge-e1 wedge)
      ,@(wedge-e2 wedge)
      ,@(wedge-e3 wedge))))
    (self (twinobj-nom wedge) nom)
    ;; If mmc rep needed.
    (cond ((or (listp (x p)) (listp (y p))
      (listp (z p))
      (listp (x e1)) (listp (y e1)) (listp (z e1))
      (listp (x e2)) (listp (y e2)) (listp (z e2))
      (listp (x e3)) (listp (y e3)) (listp (z e3)))
      (setq mmc (twin-try '(wedge ,(yif (x
        p))
        ,(yif (y p)) ,(yif (z p))
        ,(yif (x e1))
        ,(yif (y e1)) ,(yif (z e1))
        ,(yif (x e2))
        ,(yif (y e2)) ,(yif (z e2))
        ,(yif (x e3)) ,(yif (y e3))
        ,(yif (z e3))))
        (self (twinobj-mmc wedge) mmc))
        (t (self (twinobj-mmc wedge) nil)))
      wedge))

;; build-faceted-cylinder- Same as build cylinder
;; only
;; lets us specify number of facets to be used,
;; in

```

```

;; so doing we can build an n-sided peg.
(defun build-faceted-cylinder (start ht rad facets
  oper)
  (let ((cyl) (nom) (mmc))
    (setq cyl (make-instance 'cylinder
      :base '(0 0 ,(get-nom start))
      :axis '(0 0 ,(get-nom ht))
      :radius (get-nom rad)
      :facets facets))
    (setf (twinobj-operator cyl) oper)
    (setq nom (twin-try '(cylinder
      .@(cylinder-base cyl)
      .@(cylinder-axis cyl)
      .(cylinder-radius cyl)
      .facets)))
    (setf (twinobj-nom cyl) nom)
    (cond ((or (listp start) (listp ht) (listp
      rad))
      (if (equal oper '-')
        (setq mmc
          (twin-try '(cylinder 0 0
            .(get-max start)
            0 0
            .(get-min ht)
            .(get-min rad)
            .facets)))
        (setq mmc
          (twin-try '(cylinder 0 0
            .(get-min start)
            0 0
            .(get-max ht)
            .(get-max rad)
            .facets)))
        (setf (twinobj-mmc cyl) mmc))
        (t (setf (twinobj-mmc cyl) nil)))
      cyl))

;; Destructively builds twinobj out of components
;; in list, assumes
;; union.
(defun build-composite (list oper)
  (let ((comp) (nom) (mmc) (item) (bb) (opera))
    (setq comp (make-instance 'composite
      :components list))
    (setf (twinobj-operator comp) oper)
    (cond ((apply 'myor (mapcar 'twinobj-mmc
      list))
      (if (null (twinobj-mmc (first list)))
        (setq mmc (twin-try
          'copy ,(twinobj-nom (first list))))
        (setq mmc (twinobj-mmc (first list)))
        (setf (twinobj-mmc (x list)) nil)
        (dolist (item (cdr list) mmc)
          (if (null (twinobj-mmc item))
            (setq bb (twin-try 'copy ,(twinobj-nom
              item))))
            (setq bb (twinobj-mmc item)))
            (setq opera (twinobj-operator item))
            (setq nom (twin-try
              'combine ,nom ,opera
              ,bb)))
            (cond ((twinobj-mmc item)
              (setf (twinobj-mmc item) nil))))
            (setf (twinobj-mmc comp) mmc))
            (t (setf (twinobj-mmc comp) nil)))
            (setq nom (twinobj-nom (x list)))
            (setf (twinobj-nom (x list)) nil)
            (dolist (item (cdr list) nom)
              (setq opera (twinobj-operator item))
              (setq nom (twin-try
                'combine ,nom ,opera
                ,(twinobj-nom item))))
              (setf (twinobj-nom item) nil)
              (setf (twinobj-nom comp) nom)

```

```

comp))

;; Constructively builds twinobj out of components
;;
;; in list checks operator
(defun build-csg (list)
  (let ((comp) (nom) (mmc nil) (item) (bb) (oper)
    (mainoper (twinobj-operator (x list))))
    (cond ((apply 'myor (mapcar 'twinobj-mmc
      list))
      (if (null (twinobj-mmc (first list)))
        (setq mmc (twin-try 'copy ,(twinobj-nom
          (first list))))
        (setq mmc (twin-try 'copy ,(twinobj-mmc
          (first list))))
        (dolist (item (cdr list) mmc)
          (if (null (twinobj-mmc item))
            (setq bb (twin-try
              'copy ,(twinobj-nom item))))
            (setq bb (twin-try
              'copy ,(twinobj-mmc item))))
            (if (not (equal mainoper
              (twinobj-operator item)))
              (setq oper '-') (setq oper '+))
            (setq mmc
              (twin-try 'combine ,mmc ,oper ,bb))))
            (setq nom (twin-try 'copy ,(twinobj-nom (x
              list))))
            (dolist (item (cdr list) nom)
              (setq bb (twin-try 'copy ,(twinobj-nom
                item))))
              (if (not (equal mainoper
                (twinobj-operator item)))
                (setq oper '-') (setq oper '+))
              (setq nom (twin-try
                'combine ,nom ,oper ,bb)))
              (values nom mmc)))

;; copy-twinobj- Makes a copy of a csg primitive.
(defgeneric copy-twinobj (inobj twinobj))

(defmethod copy-twinobj ((inobj wedge))
  (setq obj (make-instance 'wedge :p (wedge-p
    inobj)
    :e1 (wedge-e1 inobj)
    :e2 (wedge-e2 inobj)
    :e3 (wedge-e3 inobj)))
  (general-twinobj-copy inobj obj))

(defmethod copy-twinobj ((inobj ppiped))
  (setq obj (make-instance 'ppiped :p (ppiped-p
    inobj)
    :e1 (ppiped-e1 inobj)
    :e2 (ppiped-e2 inobj)
    :e3 (ppiped-e3 inobj)))
  (general-twinobj-copy inobj obj))

(defmethod copy-twinobj ((inobj cylinder))
  (setq obj
    (make-instance 'cylinder :base (cylinder-base
      inobj)
      :axis (cylinder-axis inobj)
      :radius (cylinder-radius inobj)
      :facets (cylinder-facets inobj)))
  (general-twinobj-copy inobj obj))

(defmethod copy-twinobj ((inobj cone))
  (setq obj
    (make-instance 'cone :base (cone-base inobj)
      :axis (cone-axis inobj)
      :bottomradius (cone-bottomradius inobj)
      :topradius (cone-topradius inobj)
      :facets (cone-facets inobj)))

```

```

(general-twinobj-copy inobj obj))

(defmethod copy-twinobj ((inobj composite))
  (setq obj (make-instance 'composite
    :components (composite-components
      inobj)))
  (general-twinobj-copy inobj obj))

;; general-twinobj-copy- Copies attributes common
;; to all
;; twinobjs
(defun general-twinobj-copy (inobj obj)
  (setf (twinobj-nom obj)
    (twin-try 'copy ,(twinobj-nom inobj))))
  (setf (twinobj-operator obj) (twinobj-operator
    inobj))
  (setf (twinobj-feature obj) (twinobj-feature
    inobj))
  (setf (twinobj-movable-surfaces obj)
    (twinobj-movable-surfaces inobj))
  (setf (twinobj-aux obj) (twinobj-aux inobj))
  (if (twinobj-mmcc inobj)
    (setf (twinobj-mmcc obj) (twin-try
      'copy ,(twinobj-mmcc inobj))))
    (setf (twinobj-mmcc obj) nil)) obj)

;; This file contains routines for finding final
;; mating pose
;; of an assembly. See Chap 3.5 of thesis.
;;
;; endpt-loc- Finds at least 1 poss pose for
;; grasped-part
;; to be mated w fixtured-part in assembly. Part
;; may
;; be part or subassembly (see assembly data
;; struct).
(defun endpt-loc (mate)
  (let ((alf (assembly-left-features mate))
    (arf (assembly-right-features mate))
    (ac (assembly-connection mate))
    (talf nil) (tarf nil) (tac nil))
    (cond ((typep mate 'assembly)
      (dolist (m ac)
        (cond ((not (typep m 'thru))
          (setq tac '(.@tac m))
          (setq talf '(.@talf (car alf)))
          (setq tarf '(.@tarf (car arf))))
          (setq alf (cdr alf)) (setq arf (cdr arf)))
        (setq alf talf) (setq arf tarf) (setq ac tac)
        (if (*constraints* (constraint-loc mate alf arf
          ac))
          (endpt-loc-r mate alf arf ac))
        (t '(0 0 0 0 0 0))))))

;; endpt-loc-r- recursive function. Endpt-loc
;; calls it
;; and combine calls it recursively. feat1
;; feat2 are
;; mating features of subassembly currently
;; under
;; consideration, mate is master assembly.
(defun endpt-loc-r (mate feat1 feat2 connection)
  (let ((f2p nil) (f1p nil) (connect-xform nil)
    (angle) (pose))
    (cond
      ; Mating single feature with single feature
      ((null (y feat1))
        (setq angle (feature-constraint-angle (x
          feat1)))
        (setq f2p (get-position
          (assembly-fixtured-part mate) (x
            feat2)))
        (setq f1p (get-position
          (assembly-grasped-part mate) (x
            feat1)))
        (setf (connect-xform
          (get-connection-xform (x connection)))
          (if (null (assembly-right-pose mate))
            (setq pose
              (xmult (make-xform f2p) connect-xform
                (invert-xform (make-xform f1p))))
            (setq pose (xmult (make-xform
              (assembly-right-pose mate)
              (make-xform f2p) connect-xform
              (invert-xform (make-xform f1p))))))
          (format t "Tf2-p2 = ~s ~% M = ~s ~% Tf1-p1 = ~s ~%"
            f2p (lift-pose connect-xform) f1p)
          (if angle (setq pose (new-part-pose
            mate (x feat1) (x feat2)
            '(0 0 0 0 0 ,(x angle)) pose)))
            '(pose))
          ; Mating several features with several features
          (t
            (combine mate (car feat1) (car feat2)
              (list (car connection))
              (cdr feat1) (cdr feat2) (cdr
                connection))))))

;; Finds the length of the mating feature along
;; mating axis.
(defgeneric length-z-axis (feat feature))

(defmethod length-z-axis ((feat spur-gear))
  (spur-gear-h feat))

(defmethod length-z-axis ((feat round-peg))
  (+ (round-peg-cheight feat)
    (round-peg-bheight feat)))

(defmethod length-z-axis ((feat
  composite-round-peg))
  (dolist (item
    (composite-round-peg-parts feat) height)
    (setq height
      (+ height (cadr item)))) height)

(defmethod length-z-axis ((feat nsided-peg))
  (nsided-peg-height feat))

(defmethod length-z-axis ((feat round-hole))
  (+ (round-hole-cheight feat)
    (round-hole-chheight feat)))

(defmethod length-z-axis ((feat
  composite-round-hole))
  (dolist (item
    (composite-round-hole-parts feat) height)
    (setq height (+ height (cadr item))))
    height)

(defmethod length-z-axis ((feat threaded-peg))
  (+ (threaded-peg-height feat)
    (threaded-peg-bheight feat)))

(defmethod length-z-axis ((feat nsided-hole))
  (nsided-hole-height feat))

(defmethod length-z-axis ((feat hex-nut))
  (+ (* 2 (hex-nut-cheight feat)) (hex-nut-oheight
    feat)))

(defmethod length-z-axis ((feat threaded-hole))
  (+ (threaded-hole-height feat)
    (threaded-hole-cheight feat)))

```

```
;; combine-Mate is defining assembly. Initially we
;; had (f11.f21)
```

```

;; mates with (f12.f22) via m1 and m2. The idea
;; is to
;; find a mating pose that satisfies f11 m1 f12,
;; and
;; f21 m2 f22. (Note f21, m2 and f22 are lists
;; of possible
;; multiple values. All constraints must be
;; satisfied.
(defun combine (mate f11 f21 m1 f12 f22 m2)
  (let ((ppart1 nil) (ppart2 nil) (sym1 nil) (sym2
      nil))
    (tf2-f2p) (tf2p-f2) (tf1-f1p) (tf1p-f1) (front)
    (fp (assembly-fixed-part mate))
    (gp (assembly-grasped-part mate))
    (k 0) (list nil) (newlist nil) (pospose)
    (possym) (angle))
    ;; If two sets of mating features
    (cond ((and (listp f12) (null (cadr f12)))
      (setq f12 (car f12)) (setq f22 (car f22)))
    ;; If a gear, handle specially
    (cond ((and (typep (x m1) 'mesh) (typep (x
      m2) 'onto))
      (list (shaft-mesh (x m1) f11 f21 f12 f22)))
    ((and (typep (x m2) 'mesh) (typep (x m1)
      'onto))
      (list (shaft-mesh (x m2) f12 f22 f11 f21)))
    ;; Else use standard algorithm
    (t
      (setq ppart1
        (car (endpt-loc-r mate '(f11) '(f21) m1)))
      (setq ppart2
        (car (endpt-loc-r mate '(f12) '(f22) m2)))
      (setq sym1 (feature-constraint-numsyms f11))
      (if (null sym1) (setq sym1 (y (feature-sym
        f11))))
      (setq sym2 (feature-constraint-numsyms f12))
      (if (null sym2) (setq sym2 (y (feature-sym
        f12))))
      ;; See notes or thesis to understand
      ;; equations,
      ;; the convention is
      ;; f2
      ;; tf2-f2p= T
      ;; f2'
      (setq tf2-f2p (mmmult (invert-xform
        (make-xform (get-position
          fp f21)))
        (make-xform (get-position
          fp f22))))
      (setq tf1-f1p (mmmult (invert-xform
        (make-xform (get-position
          gp f11)))
        (make-xform (get-position
          gp f12))))
      ;; Iterate on feature pr w/ fewest number of
      ;; symmetries, sym=0 means continuous sym.
      (cond ((or (and (> sym1 0) (< sym1 sym2))
        (equal sym2 0))
        (setq tf2p-f2 (invert-xform tf2-f2p))
        (setq front (xmult
          (invert-xform
            (get-connection-xform (x m2)))
            tf2p-f2 (get-connection-xform
              (x m1))))
        (setq angle (feature-constraint-angle f11))
        (if (null angle) (setq angle '()0))
        (do ((k (x angle) (getnextp k sym1)
          (cdr angle))))
          ((almost *2PI* k) list)
          (setq angle (cdr angle))
          (setq possym
            (lift-pose
              (xmult front (rotz k) tf1-f1p)))

```

```

(cond ((valid-sym possym f12)
(format t "possym= ~s ~%" possym)
(setq pospose (new-part-pose
mate f11 f21
'(0 0 0 0 0 .k) ppart1))
(setq list
(append '(,pospose) list))))))
(t
(setq angle (feature-constraint-angle f12))
(if (null angle) (setq angle '(0)))
(setq tf1p-f1 (invert-xform tf1-f1p))
(setq front
(xmult (invert-xform
(get-connection-xform (x m1)))
tf2-f2p (get-connection-xform
(x m2))))
(do ((k (x angle) (getnextp k sym2
(cdr angle))))
((almost *2PI* k) list)
(setq angle (cdr angle))
(setq possym
(lift-pose
(xmult front (rotz k) tf1p-f1)))
(cond ((valid-sym possym f11)
(format t "possym= ~s ~%" possym)
(setq pospose (new-part-pose
mate f12 f22 '(0 0 0 0 0 .k) ppart2))
(setq list (append '(,pospose) list))))))
;; If more than 2 pr of mating features,
;; consider 2 pr at
;; a time and recursively combine with other
;; features.
(t (setq list (combine mate (car f12) (car
f22)
(list (car m2))
(cdr f12) (cdr f22) (cdr m2)))
(setq ppart1 (car (endpt-loc-r mate
'(,f11) '(,f21) m1)))
(setq sym1 (y (feature-sym f11)))
(do ((k 0 (getnext k sym1))) ((< *2PI* k)
newlist)
(setq pospose (new-part-pose mate f11
f21 '(0 0 0 0 0 .k) ppart1))
(if (almost-member pospose list)
(setq newlist (append
'(',pospose) newlist))))))
;; new-part-pose- Find the new pose of feat when
;; transform by sym
(defun new-part-pose (mate feat f2 sym pp)
(let ((fp nil) (fp1 nil) (xform nil) (list nil)
(theta nil)
(lever nil) (newpose nil) (theta1 nil))
(cond ((not (typep f2 'spur-gear))
(setq fp (mmult pp
(make-xform (get-position
(assembly-grasped-part mate)
feat)))
(setq fp1 (invert-xform fp))
(xmult fp (make-xform sym) fp1 pp))
;; Handle spur-gears specially.
(t (setq lever (+ (gear-radius feat)
(gear-radius f2)
(/ (+ (spur-gear-y feat) (spur-gear-y f2))
2.0)))
(setq newpose (lift-pose pp))
(setq theta1 (r newpose))
(setq theta (- theta1 (/ *PI* (gear-num-teeth
feat))))
(setq newpose '(',(+ (- (x newpose) (* lever
(cos theta)))
(* lever (cos (+ theta1 (r sym))))
.+, (- (y newpose) (* lever (sin theta)))
(* lever (sin (+ theta1 (r sym))))
.+, (z newpose) .+, (d newpose) .+, (e newpose)
.+, theta1 (r sym))))
(make-xform newpose))))))
;; same-pose- Returns t if m1 and m2 are almost
;; same
;; homogeneous transform pose matrices.
(defun same-pose (m1 m2)
(setq m1 (lift-pose m1))
(setq m2 (lift-pose m2))
(if (and (almost (x m1) (x m2))
(almost (y m1) (y m2))
(almost (z m1) (z m2))
(almost (d m1) (d m2))
(almost (e m1) (e m2))
(almost (r m1) (r m2))) t nil))
;; same-vector- Returns t if m1 & m2 are almost
;; same vector.
(defun same-vector (m1 m2)
(cond ((null m1) t)
(t (and (almost (car m1) (car m2))
(same-vector (cdr m1) (cdr m2))))))
;; almost- Returns t if m1,m2 are almost equal
;; allows user to
;; make allowances for floating point
;; error.
(defun almost (x y)
(if (< (abs (- x y)) 0.05) t nil))
(defun almost-angle (x y)
(if (or (almost x y) (< (- (cos x) (cos y))
0.001)) t nil))
;; getnext- Find next value of theta where sym
;; maintained.
(defun getnext (k sym)
(cond ((equal sym 1) *2PI*)
((equal sym 0) (+ k .261799387799))
(t (+ k (/ *2PI* sym))))))
;; getnextp- Find next value of theta where sym
;; maintained
;; when there is a constraint angle to be
;; considered.
(defun getnextp (k sym angle)
(cond (angle (x angle))
((equal sym 1) *2PI*)
((equal sym 0) (+ k .261799387799))
(t (+ k (/ *2PI* sym))))))
;; getlast- Find last value of theta where sym
;; maintained.
(defun getlast (k sym)
(cond ((equal sym 1) *2PI*)
((equal sym 0) (- k .261799387799))
(t (- k (/ *2PI* sym))))))
;; almost-member- Member function using almost
;; function (defined earlier) as a test.
(defun almost-member (el list)
(cond ((null list) nil)
(t (or (same-pose el (car list))
(almost-member el (cdr list))))))
;; Produces mating xform M that relates pose of
;; grasped
;; feature to position of fixtured feature when
;; mated as
;; specified by the connection argument.
(defgeneric get-connection-xform

```

```

(connection attachment))

(defmethod get-connection-xform ((connection
into))
(make-xform
'(0 0 ,(value (attachment-offset connection)) 0
0 0)))

(defmethod get-connection-xform ((connection
onto))
(make-xform
'(0 0 ,(value (attachment-offset connection)) 0
0 0)))

(defmethod get-connection-xform
((connection threaded))
(make-xform
'(0 0 ,(value (attachment-offset connection)) 0
0 0)))

(defmethod get-connection-xform ((connection
mesh))
(let ((theta) (r) (dtheta))
(f1 (attachment-grasped connection))
(f2 (attachment-featured connection)) (p1) (p2)
(conns)
(mate (attachment-mate connection)) (onto nil))
(setq conns (assembly-connection mate))
(dolist (item conns)
(if (typep item 'onto) (setq onto item)))
(cond (onto (shaft-mesh connection f1 f2
(attachment-grasped onto)
(attachment-featured onto)))
(t (setq p2 (get-position
(assembly-featured-part mate) f2))
(setq p1 (get-position
(assembly-grasped-part mate) f1))
(setq dtheta (* (r p2) (/ (gear-num-teeth f2)
(gear-num-teeth f1))))
(setq theta (/ (- (spur-gear-x2 f2)
(spur-gear-x1 f2)) 2.5))
(setq theta (- theta dtheta))
(setq r (+ (gear-radius f1) (gear-radius f2)
(/ (+ (spur-gear-y f1)
(spur-gear-y f2)) -2.5)))
(mmmult (rotz (- 0 (r p2)))
(make-xform '(r 0
(attachment-offset mesh)
0 0 ,theta))))))

(defmethod get-connection-xform ((connection
lineup))
(let ((t1 nil) (t2 nil) (h1 nil) (h2 nil))
(feat1 (attachment-grasped connection))
(feat2 (attachment-featured connection)))

(setq t1 (class-name (class-of feat1)))
(setq t2 (class-name (class-of feat2)))
(cond ((and (member t1 '(hex-nut
threaded-hole))
(member t2 '(hex-nut threaded-hole)))
(setq h1 (length-z-axis feat1)))
((and (not (member t1 '(hex-nut
threaded-hole)))
(not (member t2 '(hex-nut threaded-hole))))
(setq h1 (length-z-axis feat2)))
((and (member t1 '(hex-nut threaded-hole))
(not (member t2 '(hex-nut threaded-hole))))
(setq h1 (+ (length-z-axis feat1)
(length-z-axis feat2))))
(t (setq h1 0)))
(make-xform
'(0 0 ,(+ h1 (attachment-offset
connection)) 0 0 0)))

```

```

(defun shaft-mesh (con f11 f21 f12 f22)
(let ((alpha1) (alpha) (dtheta) (theta) (m 0)
(x2) (alpha2)
(p2 0) (mplace 0) (toff) (p3) (connect-xform)
(f1p))
(setq p2 (get-position (feature-part f21)
f21))
(setq p3 (get-position (feature-part f22)
f22))
(setq dtheta (* (r p2) (/ (gear-num-teeth
f21)
(gear-num-teeth f11))))
(setq mplace (atan (- (y p3) (y p2)) (- (x
p3) (x p2))))
(setq toff (/ *PI* (gear-num-teeth f21)))
(setq theta (- toff dtheta mplace))
(setq theta (resolve-theta theta
(gear-num-teeth f11)))
(setq connect-xform
(mmmult (rotz (- 0 (r p2)))
(make-xform '(0 0 ,(attachment-offset
con)
0 0 ,theta))))
(setq f1p (get-position (feature-part f11) f11))
(xmult (make-xform p3) connect-xform
(invert-xform (make-xform f1p))))))

(defun resolve-theta (theta num)
(let ((inc (/ *2PI* num)) (times))
(setq times (truncate (/ theta inc)))
(- theta (* times inc))))

;; valid-sym- Find out if pose is a valid sym of
;; the feature feat.
;; Assume all syms are of the form rotz(x).
;; This routine
;; could easily be expanded to take into
;; account
;; different types of symmetries.
(defun valid-sym (pose feat)
(cond ((and (almost 0 (x pose))
(almost 0 (y pose))
(almost 0 (z pose))
(almost 0 (d pose))
(almost 0 (e pose))
(valid-rotz-sym (r pose) feat)) t)
(t nil)))

;; valid-rotz-sym- For syms about z-axis, find out
;; if angle is a
;; valid rotz symmetry of feature.
(defun valid-rotz-sym (angle feat)
(let ((offangle (feature-constraint-angle feat))
(fca (feature-constraint-angle feat))
(sym (feature-constraint-numsyms feat))
(spacing))
(cond (fca (member angle fca :test
'almost))
(t
(if (null sym) (setq sym (y (feature-sym
feat))))
(if (null offangle) (setq offangle 0))
(cond ((equal sym 0) t)
(t (setq angle (- angle offangle))
(setq spacing (/ *2PI* sym))
(if (almost (rem angle spacing) 0)
t nil))))))

;; This file contains code for resolving the
;; constraints on
;; assembly operations. See Chap 3.6 of thesis.
;;

```

```

;; constraint-loc- Finds global constraints on
;; assem with
;; left-features alf and right-features arf and
;; connections
;; connection.
(defun constraint-loc (assem alf arf connection)
  (let ((agp (assembly-grasped-part assem))
        (afp (assembly-fixtured-part assem))
        (f11) (f21) (m1) (f12) (f22) (m2) (feat1)
        (feat2)
        (sym1) (sym2) (sym3 1) (sym4 1) (k 0) (l 0) (i)
        (j)
        (ang1) (ang2) (ang3) (ang4)
        (tf2?-f1?) (tf1?-f11) (tf11-f12) (tf12-f2?)
        (part1) (part2) (feat1?) (feat2?) (mate1?)
        (mate2?))
    (list) (list2) (assem?) (possym))
    (setq f11 (x alf)) (setq f21 (x arf))
    (setq m1 '(x connection))
    (setq f12 (cdr alf)) (setq f22 (cdr arf))
    (setq m2 (cdr connection))
    (cond ((null f12) t)
          ((and (listp f12) (null (cadr f12))))
    (setq f12 (car f12)) (setq f22 (car f22))
    (cond ((and (typep (x m1) 'mesh)
                 (typep (x m2) 'onto)) t)
          ((and (typep (x m2) 'mesh)
                 (typep (x m1) 'onto)) t)
    (t (setq part1 (feature-part f21))
        (setq part2 (feature-part f22))
        (setq m1 (x m1)) (setq m2 (x m2))
        ;; feat1 is an earlier mated feature on
        ;; part1
        ;; mate? is the way it mates to feat1?
        (multiple-value-setq
         (feat1 feat1? mate1? assem?)
         (find-mate-desc part1 afp))
        (multiple-value-setq
         (feat2 feat2? mate2? assem?)
         (find-mate-desc (feature-part f22)
                          afp))
        (cond ((and feat1? feat2?)
              (setq tf2?-f1?
                    (mmmult
                     (invert-xform
                      (make-xform (get-position
                                   afp feat2?)))
                     (make-xform (get-position
                                   afp feat1?)))
                    (and feat1? (null feat2?))
                    (setq tf2?-f1?
                          (make-xform (get-position
                                       afp feat1?)))
                    (and (null feat1?) feat2?)
                    (setq tf2?-f1?
                          (invert-xform
                           (make-xform (get-position
                                         afp feat2?)))
                          (t (setq tf2?-f1? (make-xform
                                             '(0 0 0 0 0 0))))
                    (cond (mate1?
                          (setq tf1?-f11
                                (xmult
                                 (get-connection-xform mate1?)
                                 (invert-xform
                                  (make-xform (get-position
                                                part1 feat1)))
                                  (make-xform (get-position
                                                part1 f21))
                                  (get-connection-xform m1))))
                          (t
                           (setq tf1?-f11
                                 (xmult
                                  (invert-xform
                                   (make-xform (get-position
                                                  part1 feat1)))
                                   (make-xform (get-position
                                                  part1 f21))
                                   (get-connection-xform m1))))
                    (t
                     (setq tf1?-f11
                           (xmult
                            (invert-xform
                             (make-xform (get-position
                                             part1 feat1)))
                             (make-xform (get-position
                                             part1 f21))
                             (get-connection-xform m1))))
                    (invert-xform
                     (make-xform (get-position
                                   part1 feat1)))
                    (make-xform (get-position
                                   part1 f21))
                    (get-connection-xform m1))))))
    (t
     (setq tf1?-f11
           (xmult
            (invert-xform
             (make-xform (get-position
                           part1 feat1)))
            (make-xform (get-position
                           part1 f21))
            (get-connection-xform m1))))))

```

```

(invert-xform
 (make-xform (get-position
               part1 feat1)))
(make-xform (get-position
              part1 f21))
(get-connection-xform m1))))
(setq tf11-f12
  (xmult
   (invert-xform (make-xform (get-position
                               agp f11)))
   (make-xform (get-position agp f12))))
(cond (mate2?
      (setq tf12-f2?
            (invert-xform
             (xmult (get-connection-xform
                    mate2?)
                    (invert-xform
                     (make-xform
                      (get-position
                       part2 feat2)))
                     (make-xform
                      (get-position
                       part2 f22))
                     (get-connection-xform
                      m2))))))
      (t
       (setq tf12-f2?
             (invert-xform
              (xmult (invert-xform
                      (make-xform
                       (get-position
                        part2 feat2)))
                      (make-xform
                       (get-position
                        part2 f22))
                      (get-connection-xform
                       m2))))))
      (cond (feat1?
            (setq sym1 (y (feature-sym feat1?)))
            (setq ang1 (feature-constraint-angle
                        feat1?))
            (if (null ang1) (setq ang1 '(0))))
            (t
             (setq sym1 0)
             (setq ang1 '(0))))
      (cond (feat2?
            (setq sym2 (y (feature-sym feat2?)))
            (setq ang2 (feature-constraint-angle
                        feat2?))
            (if (null ang2) (setq ang2 nil)))
            (t
             (setq sym2 0)
             (setq ang2 nil)))
      (setq sym3 (y (feature-sym f11)))
      (setq ang3 (feature-constraint-angle f11))
      (if (null ang3) (setq ang3 '(0)))
      (setq sym4 (y (feature-sym f12)))
      (setq ang4 (feature-constraint-angle f12))
      (if (null ang4) (setq ang4 '(0)))
      ;; Essentially we want to find symmetries
      ;; where
      ;; T(afp,feat1?) sym(feat1?) T(feat1?,f11)
      ;; sym(f11) T(f11,agp) = T(afp,feat2?)
      ;; sym(feat2?)
      ;; T(feat2?,f12) sym(f12) T(f12,agp) so we
      ;; solve
      ;; equation for sym(feat2?), plug in numbers,
      ;; &
      ;; whenever possym is a poss symmetry of
      ;; feat2?, we
      ;; know equation holds, and record the symms.
      (do ((i (x ang3)
              (getnextp i sym3 (cdr ang3))))

```



```

(multiple-value-setq
 (feat feat? mate massem)
 (find-mate-desc part
 (assembly-featured-part assem)))
(values feat feat? mate massem)))
(t (values nil nil nil nil))))

;; aligned- are feat1's and feat2's mating axes
;; coincident?
(defun aligned (mate feat1 feat2)
  (let ((pose1) (pose2))
    (setq pose1 (get-position
      (assembly-grasped-part mate) feat1))
    (setq pose2 (get-position
      (assembly-grasped-part mate) feat2))
    (if (and (almost (x pose1) (x pose2))
      (almost (y pose1) (y pose2))
      (almost (d pose1) (d pose2))
      (almost (e pose1) (e pose2)))
      t nil)))

;; valid-part-sym- is pose a valid sym? if so
;; return the sym
;; assume all symmetries are of form rotz(),
;; and
;; angle can be found knowing # of symmetries
;; of the feature.
(defun valid-part-sym (pose feat number angle)
  (let ((vector (lift-pose pose)))
    (cond ((null feat) nil)
      ((and (almost (x vector) 0)
        (almost (y vector) 0)
        (almost (z vector) 0)
        (almost (d vector) 0)
        (almost (e vector) 0))
        (cond (angle (member (r vector) angle) (r
          vector))
          ((equal number 0) (r vector))
          (t (setq spacing (/ *2PI* number))
            (if (almost (rem (r vector) spacing) 0)
              (r vector) nil))))
      (t nil))))

(defvar *constraints* nil)

;;
;; This file contains methods associated with
;; classes and
;; subclasses of peg and hole.
;;
;; round-peg- Build rep for possibly bevelled,
;; cylindrical peg.
(defmethod initialize-instance :after ((peg
  round-peg)
  &key radius cheight
  bradius bheight)
  (let ((csg))
    (setq csg (list (build-cylinder
      bheight cheight radius '+)))
    (if (> bheight 0)
      (setq csg (append
        csg (list (build-cone
          0 bheight bradius radius '+))))
      (setf (feature-sym peg) '(k 0))
      (setf (feature-sign peg) 1)
      (setf (feature-csg peg) csg)))

;; composite-round-peg- Build rep for peg that can
;; be
;; described using cylinders and cones.
(defmethod initialize-instance
  :after ((peg composite-round-peg) &key parts)
  (let ((csg) (height 0) (part))
    (setq csg (list (buildgccone (x parts) 0 '
      +)))
    (setq height (y (x parts)))
    (dolist (part (cdr parts) csg)
      (setq csg (append csg (list (buildgccone
        part height '-))))
      (setq height (ssum height (y part))))
    (setf (feature-sym peg) '(k 0))
    (setf (feature-sign peg) 1)
    (setf (feature-csg peg) csg)))

;; nsided-hole- Build up an n-sided polygonal hole
(defmethod initialize-instance :after ((hole
  nsided-hole)
  &key radius height n)
  (let ((csg))
    (setq csg (list (build-faceted-cylinder
      0 height radius n '-)))
    (setf (feature-sym hole) '(k ,n))
    (setf (feature-sign hole) -1)
    (setf (feature-csg hole) csg)))

;; arb-peg- Build up an arbitrarily shaped peg.
(defmethod initialize-instance :after ((instance
  arb-peg)
  &key parts)
  (build-parts instance parts)
  (setf (feature-sign instance) 1))

;; round-hole- Build up a possibly chamfered,
;; cylindric hole.
(defmethod initialize-instance :after ((hole
  round-hole)
  &key radius
  cheight chradius chheight)
  (let ((csg))
    (setq csg (list (build-cylinder
      chheight cheight radius '-)))
    (if (> chheight 0)
      (setq csg
        (append csg '(, (build-cone
          0 chheight chradius
          radius '-))))
      (setf (feature-sym hole) '(k 0))
      (setf (feature-sign hole) -1)
      (setf (feature-csg hole) csg)))

;; composite-round-hole- Build hole that described
;; with cylinders and cones.
(defmethod initialize-instance :after
  ((hole composite-round-hole) &key parts)
  (let ((csg) (height 0) (part))
    (setq csg (list (buildgccone (x parts) 0 '
      -)))
    (setq height (y (x parts)))
    (dolist (part (cdr parts) csg)
      (setq csg (append csg (list (buildgccone
        part height '-))))
      (setq height (ssum height (y part))))
    (setf (feature-sym hole) '(k 0))
    (setf (feature-sign hole) -1)
    (setf (feature-csg hole) csg)))

;; nsided-hole- Build up an n-sided polygonal hole
(defmethod initialize-instance :after ((hole
  nsided-hole)
  &key radius height n)
  (let ((csg))
    (setq csg (list (build-faceted-cylinder
      0 height radius n '-)))
    (setf (feature-sym hole) '(k ,n))
    (setf (feature-sign hole) -1)
    (setf (feature-csg hole) csg)))

```

```
(setf (feature-sign hole) -1)
(setf (feature-csg hole) csg)))

;; arb-hole- Build up an arbitrarily shaped hole.
(defmethod initialize-instance :after ((instance
arb-hole)
&key parts)
(build-parts instance parts)
(setf (feature-sign instance) -1))
```

```
; This is default routine, used for feature,
;; arb-peg, arb-hole.
```

```
(defun build-parts (instance parts)
(let ((csg) (part))
(setq csg (list (buildrep (first parts))))
(dolist (part (cdr parts) csg)
(setq csg (append csg (list (buildrep
part))))))
(setf (feature-csg instance) csg)))
```

```
(defun buildgccone (part height oper)
(cond ((equal (x part) 'cylinder)
(build-cylinder height (sdif (y part) 0.01)
(ssum (z part) 0.01) oper))
(t (build-cone height (sdif (y part) 0.01)
(ssum (z part) 0.01) (d part) oper))))
```

```
(defun buildrep (list)
(let ((op (car list)) (parms (cdr list))
(transform) (rep))
(multiple-value-setq (parms transform)
(split-list parms 'at))
(setq rep (cond ((equal op 'ppiped) (apply
'build-ppiped
parms))
((equal op 'cylinder) (apply
'build-cylinder
parms))
((equal op 'cone) (apply 'build-cone
parms))
((equal op 'composite) (apply
'build-composite
parms))
((equal op 'wedge) (apply 'build-wedge
parms))))
(cond (transform
(setq transform (car transform))
(if (< (length transform) 6)
(setq transform (append transform
'(0 0 0))))
(setq rep (twin-try '(xform ,@transform
.rep))))
(t rep))))
```

```
(defun split-list (list key)
(let ((first list) (last))
(setq last (member key list))
(cond (last (setq first (ldiff list last))
(setq last (cdr last))))
(values first last)))
```

```
;;
;; This file contains methods associated with
;; class and
;; subclasses of gear.
```

```
;; spur-gear- Build csg rep for spur gear, sets up
;;
;; symmetry and sign.
(defmethod initialize-instance :after ((gear
spur-gear)
&key radius x1 x2 y h
num-teeth offset facets)
```

```
(let ((fac *facets*))
(setq *facets* facets)
(setf (feature-sign gear) 1)
(setf (feature-csg gear)
(build-gear-r radius x1 x2 y h num-teeth
offset))
(setq *facets* fac)
(setf (feature-sym gear) '(k ,num-teeth))))
```

```
;; build-gear-r- Actual routine that builds a spur
;; gear csg rep.
```

```
(defun build-gear-r (crad x1 x2 y h num offset)
(let ((p nil) (p2) (e1 nil) (e2 nil) (e3 nil)
(csg) (irad)
(tooth) (tooth1) (tooth2))
; Find inner radius of gear which is outer
; radius-depth, if crad or y is toleranced,
; irad will be as well
(setq irad (sdif crad y))
; Start with disk for gear
(setq csg '(, (build-cylinder 0 h irad '+)))
; Do loop to build teeth representation
; See notes for significance of x1,x2,y,e1,e2
;; e3,p
(do ((*tnum* 0 (1+ *tnum*))
((equal *tnum* num))
```

```
; Build up wedge1 for a tooth
; place-tooth xforms wedge to the num-th
; tooth pose around the disk.
```

```
(setq p (place-tooth num *tnum*
',(list-max irad)
,(list-min
(sdiv (ssum x2 x1) 2.0)) 0 1)
nil))
(setq e1 (place-tooth num *tnum*
'(0 ,(list-min (sdif 0 x2)) 0 1)
p))
(setq e2 (place-tooth num *tnum*
',(list-min y)
,(list-min
(sdiv (ssdif x2 x1) -2)) 0 1)
p))
(setq e3 (place-tooth num *tnum*
'(0 0 ,(list-max h) 1) p))
(setq tooth1 (build-wedge p e1 e2 e3 '+))
```

```
; Build up wedge2 for a tooth
(setq p2 (place-tooth num *tnum*
',(list-max crad) 0 0 1) p))
(setq e1 (place-tooth num *tnum*
'(0 ,(list-min x1) 0 1) p))
(setq e2 (place-tooth num *tnum*
',(list-min (sdif 0 y))
,(list-min
(sdiv (ssdif x2 x1) -2)) 0 1)
p))
(setq e3 (place-tooth num *tnum*
'(0 0 ,(list-max h) 1) p))
(setq tooth2 (build-wedge p2 e1 e2 e3 '+))
(setf (twinobj-aux tooth1) tooth1)
(setf (twinobj-aux tooth2) tooth1)
(setq tooth '(,tooth1 ,tooth2))
; Append to csg tree
(setq csg (append csg tooth)))
csg))
```

```
;; place-tooth- Finds homogen xform describing
;; where
;; num-th tooth should be placed around the
;; gear
;; given total # of teeth.
(defun place-tooth (totnum num point xform)
```

```

(let ((theta (/ (* num *2PI*) totnum))
      (a1 nil) (a2 nil) (a3 nil) (a4 nil) (a) (b)
      (result))
  (setq theta (* -1 theta))
  (setq a1 '(, (cos theta) , (* -1 (sin theta)) 0
             0))
  (setq a2 '(, (sin theta) , (cos theta) 0 0))
  (setq a3 '(0 0 1 0))
  (setq a4 '(0 0 0 1))
  (setq result
    (cond ((or (listp (x point)) (listp (y point))
              (listp (z point))))
          (setq a (mvmult
                    '(,a1 ,a2 ,a3 ,a4) '(, (xif (x point))
                                           , (xif (y point))
                                           , (xif (z point))
                                           1)))
            (setq b (mvmult
                      '(,a1 ,a2 ,a3 ,a4) '(, (yif (x point))
                                              , (yif (y point))
                                              , (yif (z point))
                                              1)))
            '(, (if (equal (x a) (x b)) (x a)
                    (list (x a) (x b)))
              , (if (equal (y a) (y b)) (y a)
                    (list (y a) (y b)))
              , (if (equal (z a) (z b)) (z a)
                    (list (z a) (z b))))))
    (t (mvmult '(,a1 ,a2 ,a3 ,a4) point))))
  (cond (xform
        (mvmult
          (mvmult '(((1 0 0 , (x xform)) (0 1 0 , (y
xform)))
                    (0 0 1 , (z xform)) ,a4)
          (mvmult '(((1 0 0 0) (0 1 0 0) ,a3 ,a4)
                    '(((1 0 0 , (- (x xform)))
                      (0 1 0 , (- (y xform)))
                      (0 0 1 , (- (z xform))) ,a4)))
          (append result '(1))))
        (t result))))

;; This file contains methods associated with
;; class and
;; subclasses of joint.
;;
;; find-joint-transform- fixed-link to joining
;; feat, joining
;; feat to joining feat (offset), joint position,
;; joining feat
;; to mobil link. multiplication of these xform
;; relates position
;; of the mobile link to the position of the
;; fixed link.
;;
(defgeneric find-joint-transform (instance joint))

(defmethod find-joint-transform ((instance
revolute))
  (let ((ltof) (ftol) (ftof) (offset)
        (ttrans (joint-transform instance)))
    (setq ltof (x trans))
    (setq offset (y trans))
    (setq ftol (z trans))
    ;; Joint position here refers to
    ;; orientation about Z axis.
    (setq ftof (make-xform
                  '(0 0 0 0 , (joint-position instance))))
    (xmult ltof offset ftof ftol)))

(defmethod find-joint-transform ((instance
prismatic))
  (let ((ltof) (ftol) (ftof) (offset)
        (ttrans (joint-transform instance)))

```

```

      (setq ltof (x trans))
      (setq offset (y trans))
      (setq ftol (z trans))
      ;; Joint position here refers to
      ;; translation along Z axis.
      (setq ftof (make-xform
                    '(0 0 , (joint-position instance) 0 0 0)))
      (xmult ltof offset ftof ftol)))

;; This file contains methods associated with
;; flexible class.
;;
;; helical-spring- Builds csg rep and sets
;; symmetry and
;; sign for helical spring feature.
(defmethod initialize-instance
  :after ((spring helical-spring)
          &key radius compress cheight)
  (setf (feature-csg spring)
        '(, (build-flexible-cylinder
              0 cheight compress radius '+)))
  (setf (feature-sym hole) '(k 0))
  (setf (feature-sign spring) 1))

```

PATHPLAN

```

/* Source code for Phase 1 Path Planning */
/* Program PATHPLAN */
/* Calling format- pathplan profile */
/* Input:
profile- file containing dimension of problem,
goal point, start point, obj_size,
maxmo
object- File describing the object, created
by the program twincon in this instance
which takes twin boundary reps and
converts them into a convenient
representation
for our purposes. The idea is that if a
different modeler were used a different
conversion program would be written to
convert
the breps into my standard format
obstacles- File describing the obstacles, also
created by twincon. The function build-input
in the assembly supervisor takes care of
of invoking twincon */
/* Output:
mpv.out- minimum potential valley used
poa.out- initial path found
soa.out- collision free path
slp.out- collision free path with extra
points removed */

/* Description: This program finds a
collision-free path
for an assembly operation using the algorithm
discussed
in Chapter 4 and 5 of my thesis */

#include <stdio.h>
#include <math.h>
#include "common.h"
/* Header file, common w uncertanal */

obstacle *obst; /* List of obstacles*/
side *bound; /* List of bounding surfaces*/

```

```

line *edge; /* List of edges*/
vertex *vertice; /* List of vertices*/

/* # of obstacles, bounding surfaces, edges,
   vertices,
   points in path, degrees of freedom, sides on
   object */
int nob, nbound, nedge, nvert, npoints, ndim,
    objside;
int collisionflag=0;
int inobstflag;
extern int initobject();
extern struct node *openl;
extern int soa();
extern double fsqr(), distnom(), md();
extern double pointpot();
extern oside *object;
double minmo, maxmo;
double **path, **vel;
struct node *openl=NULL;
struct node *closed=NULL;
FILE *fp;

/* TOO BE REMOVED */
extern double objectpot(), clearance();
/* main- This is the main routine for pathplan. */
main(argc, argv)
int argc;
char **argv;
{
    double x1,y1,z1,x2,y2,z2,rotx1,rotx2;
    double roty1,roty2,rotz1,rotz2,x,y,z;
    double *t1, *t2, mag;
    double answer, goaldist;
    struct node *mpv, *mpvn, *goal, *temp, *n;
    int i, done;
    float d1,d2,d3,d4,d5,d6;

    if (argc < 2) {
        printf("Calling Format: progr
        param_file_name0);
        exit();
    }
    if ((fp=fopen(argv[1],"r")) == NULL) {
        perror(argv[1]);
        exit();
    }
    /* Read from parameter file- dimension, start
       point,
       end point, min and max extent of object */
    fscanf(fp,"%d",&ndim);
    fscanf(fp,"%f %f %f %f %f %f",&d1,&d2,&d3,
        &d4,&d5,&d6);
    x1=d1; y1=d2; z1=d3; rotx1=d4; roty1=d5; rotz1
        =d6;
    if (ndim > 6) {
        t1=(double *)calloc(ndim-6,sizeof(double));
        t2=(double *)calloc(ndim-6,sizeof(double));
    }
    /* Read in starting and ending joint positions
       if any */
    for (i=0; i<ndim-6; i++) {
        fscanf(fp,"%f",&d1);
        t1[i]=d1;
    }
    fscanf(fp,"%f %f %f %f %f %f",&d1,&d2,
        &d3,&d4,&d5,&d6);
    x2=d1; y2=d2; z2=d3; rotx2=d4; roty2=d5; rotz2
        =d6;
    for (i=0; i<ndim-6; i++) {
        fscanf(fp,"%f",&d1);
        t2[i]=d1;
    }
}

```

```

fscanf(fp,"%f %f",&d1,&d2);
minmo=d1; maxmo=d2;
fclose(fp);

/* Read in obstacles from file "obstacles" */
/
initobstacles();
/* Read in object from file "object" */
/* Note: object and obstacles created by
   twincon.c */
initobject();
/* Find out if goal point within an obstacle */
/
inobst(x1,y1,z1);
/* Initialize sphere tessellation */
read_icos("/home/rvl/q/research/pathplan
/icos.coord");

/* For now, ignore flexibilities by
   compressing all
   flexible obstacles */
compress();
/* Find minimum potential valley graph */
n=(struct node *)calloc(1,sizeof(struct
    node));
n->next[0]=NULL; n->father=NULL;
goal=(struct node *)calloc(1,sizeof(struct
    node));
n->x=x1; n->y=y1; n->z=z1;
goal->x=x2; goal->y=y2; goal->z=z2;
n->pot=pointpot(x1,y1,z1);
n->md=md(x1,y1,z1); goal->md=md(x2,y2,z2);
put_on_open(n); done=0;
/* An appropriate minimum potential valley is
   found
   within this while loop */
while ((openl != NULL) && (done == 0)) {
    n=openl;
    if (!on_closed(n))
        put_on_closed(n);
    openl=openl->open;
    goaldist=sqrt(fsqr(n->x-goal->x)
        +fsqr(n->y-goal->y)+
        fsqr(n->z-goal->z));
    if ((goaldist < n->md) || (goaldist < goal->md))
    {
        mpv=(struct node *)
        calloc(1,sizeof(struct node));
        mpv->x=x2; mpv->y=y2; mpv->z=z2;
        mpv->md=goal->md; mpv->rotx=rotx2;
        mpv->roty=roty2; mpv->rotz=rotz2;
        mpv->next[0]=NULL; temp=n; mpvn=mpv;
        for (i=0; i<ndim-6; i++)
            mpv->t[i]=t1[i];
        while (temp != NULL) {
            mpv=(struct node *)calloc(1,sizeof(struct
                node));
            mpv->next[0]=mpvn;
            mpv->x=temp->x; mpv->y=temp->y;
            mpv->z=temp->z; mpv->md=temp->md;
            mpvn=mpv; temp=temp->father;
        }
        mpv->rotx=rotx1; mpv->roty=roty1;
        mpv->rotz=rotz1;
        for (i=0; i<ndim-6; i++)
            mpv->t[i]=t1[i];
        for (i=0; i<ndim-6; i++)
            mpv->t[i]=t1[i];
        if (expand_path(mpv,rotx1,roty1,rotz1,t1
            ,rotx2,
            roty2,rotz2,t2) > 0)
            done=1;
    }
}

```

```
/* Remove n from open if exhausted all successors
*/
```

```
find_all_minimums(n);
for (i=0; i<n->sons; i++) {
    n->next[i]->g=fsqr(x1-n->next[i]->x)+
        fsqr(y1-n->next[i]->y)+
        fsqr(z1-n->next[i]->z);
    n->next[i]->h=fsqr(goal->x-n->next[i]->x)+
        fsqr(goal->y-n->next[i]->y)+
        fsqr(goal->z-n->next[i]->z);
    n->next[i]->father=n;
    if (!on_closed(n->next[i]))
        put_on_open(n->next[i]);
}
```

```
    }
    if (done == 0)
        printf("FAILED, NO VALID PATH0);
}
```

```
/* expand-path- Once a min potential valley
from start to end is found, expand into
initial path to complete stage 1 */
expand_path(mpv,rotx1,roty1,rotz1,t1,rotx2,roty2,
    rotz2,t2)
```

```
struct node *mpv;
double rotx1,roty1,rotz1,*t1,rotx2,roty2,rotz2,
    *t2;
```

```
{
    int i,npoints;
    struct colnode *cols;

    dumpmpv(mpv);
    /* Dump mpv to file, used for analysis */
    /* Find an initial path through min potential
    valley */
    uncompress();
    npoints=initpath(mpv,&path,&cols,1);
    /* Dump initial path into poa.out,
    used only for analysis */
    dumpresults("poa.out",npoints,path);
    /* If collisions found while creating initial
    path,
    locally optimize path */
    if (collisionflag > 0) {
        npoints=soa(path,npoints,cols);
        /* Round off rotx, roty, rotz if possible */
        cleanup(path,npoints);
    }
    /* Get rid of duplicate points and dump
    results */
    npoints=shrink(path,npoints);
    dumpresults("soa.out",npoints,path);
    /* Find straight line segments thru path and
    only use
    endpoints of line segments */
    npoints=slp(path,npoints);
    dumpresults("slp.out",npoints,path);
    if (npoints == 0) {
        compress();
        collisionflag=0;
    }
    return(npoints);
}
```

```
/* cleanup- Round rotx,roty,rotz to closest
multiple of PI/2
if close */
```

```
cleanup(path,npoints)
double **path;
int npoints;
{
    int i,j;
```

```
    for (i=0; i<npoints; i++)
        for (j=3; j<6; j++) {
            if (fabs(path[j][i]) < .05)
                path[j][i]=0.0;
            if (fabs(path[j][i]-PI/2.0) < .05)
                path[j][i]=PI/2.0;
            if (fabs(path[j][i]-PI) < .05)
                path[j][i]=PI;
            if (fabs(path[j][i]-(3.0*PI)/2.0) < .05)
                path[j][i]=(3.0*PI)/2.0;
        }
}
```

```
/* shrink- Occasionally, the soa algorithm puts
the same
point into the path two or more times, get
rid
of duplicate points */
```

```
shrink(path,npoints)
double **path;
int npoints;
{
    int i,j,k,tpoints,flag;

    tpoints=npoints;

    for (i=0; i<tpoints-1; i++) {
        flag=0;
        for (j=0; j<ndim; j++)
            if (path[j][i] != path[j][i+1])
                flag=1;
        /* If path[j][k]=path[j][k+1], get rid of k+1th
        point */
        if (flag == 0) {
            for (j=i; j<tpoints-1; j++) {
                for (k=0; k<ndim; k++)
                    path[k][j]=path[k][j+1];
            }
            tpoints--;
            i--;
        }
    }
    return(tpoints);
}
```

```
inobst(x,y,z)
double x,y,z;
```

```
{
    int i,j,num,hole,chole,sptr,flag;
    double minsum;

    inobstflag=0;
    for (i=0; i<nobs; i++)
        obst[i].consider=1;
    for (i=0; i<nobs; i++) {
        if (obst[i].shape >= 0) {
            if (distnom(x,y,z,i) < 0.001) {
                flag=1;
                for (hole=0; hole < obst[i].nholes; hole++) {
                    choke=obst[i].holes[hole];
                    minsum=0;
                    for (j=0; j < obst[chole].nsides; j++) {
                        sptr=obst[chole].side[j];
                        minsum+=fabs(bound[sptr].a*x+
                            bound[sptr].b*y+
                            bound[sptr].c*z +bound[sptr].d);
                        minsum+=bound[sptr].a*x+bound[sptr].b*y+
                            bound[sptr].c*z +bound[sptr].d;
                    }
                    if (minsum < 0.001) {
```

```

    flag=0;
    hole=obst[i].nholes;
  }
}
if (flag==1) {
  obst[i].consider=0;
  inobstflag=1;
  for (j=0; j<nobs; j++)
    if (obst[i].id == obst[j].id)
      obst[j].consider=0;
}
}
}

/* Put node on n in ascending heuristic order */
put_on_open(n)
struct node *n;
{
  struct node *ptr1;
  struct node *ptr2;
  if ((on_open(n)) || (on_closed(n)))
    return(0);
  if ((openl == NULL) || (n->h < openl->h)) {
    n->open=openl;
    openl=n;
  }
  else {
    ptr1=openl;
    ptr2=openl->open;
    while ((ptr2 != NULL) &&
      (n->g+n->h > ptr2->g+ptr2->h)) {
      ptr1=ptr2;
      ptr2=ptr2->open;
    }
    n->open=ptr2;
    ptr1->open=n;
  }
}

put_on_closed(n)
struct node *n;
{
  if (closed == NULL) {
    closed=n;
    n->closed=NULL;
  }
  else {
    n->closed=closed;
    closed=n;
  }
}

on_open(n)
struct node *n;
{
  struct node *temp;
  temp=openl;
  while (temp != NULL) {
    if (fabs(n->x-temp->x)+fabs(n->y-temp->y)+
      fabs(n->z-temp->z) < 0.05)
      return(1);
    temp=temp->open;
  }
  return(0);
}

on_closed(n)
struct node *n;
{
  struct node *temp;
  temp=closed;

```

```

    while (temp != NULL) {
      if (fabs(n->x-temp->x)+fabs(n->y-temp->y)+
        fabs(n->z-temp->z) < 0.05)
        return(1);
      temp=temp->closed;
    }
    return(0);
  }
}

print_open()
{
  struct node *temp;
  temp=openl;
  while (temp != NULL) {
    printf("%f %f %f %f\n",temp->x,temp->y,
      temp->z,temp->md);
    temp=temp->open;
  }
}

print_closed()
{
  struct node *temp;
  temp=closed;
  while (temp != NULL) {
    printf("%f %f %f %f\n",temp->x,temp->y,
      temp->z,temp->md);
    temp=temp->closed;
  }
}

/* This file contains code for calculating
min_dist(x) by the
first method presented in thesis. Routine is
md_internal.
Two methods are used interchangeably throughout
code. */
#include <stdio.h>
#include <math.h>
#include "common.h"

extern obstacle *obst;
extern side *bound;
extern vertex *vertice;
extern line *edge;
extern double fsqr();

extern int nobs, nbounds, nedge, nvert, npoints,
ndim;
double md();
double md_internal();
extern double distnom();

double md(x,y,z)
double x,y,z;
{
  return(fabs(md_internal(x,y,z)));
}

/* Find minimum distance from point to nearest
obstacle */
/* Return -1 * mindist if point inside of fixtured
part */
/* Return mindist otherwise */
double md_internal(x,y,z)
double x,y,z;
{
  double i1, i2, j1, j2, k1, k2, mindist, dist,
det, mag;
double v1x,v1y,v1z,v2x,v2y,v2z,v3x,v3y,v3z,num
,den;
double lambda1,lambda2;
double xstar,ystar,zstar,b[3],a[3][3],a1[3][3]
,n1[3],n2[3],n3[3];

```

```

double point[3],mid[3],normal[3],minodist;
int i, minvert, edgei, edgej, j, k, l, m, ptr,
    p1, p2, p3;
int outside, face;
int flag,minobst,h,inflag;

mindist=20000;
for (i=0; i<nobs; i++) {
    if (obst[i].consider == 1) {
        /* Dont consider obstacles too far away */
        minodist=20000; inflag=0;
        for (j=0; j<nvert; j++) {
            if (onobst(j) == 1) {
                dist=sqrt(fsqr(x-vertex[j].x)
                    +fsqr(y-vertex[j].y)+
                    fsqr(z-vertex[j].z));
                if (dist < minodist) {
                    minodist=dist;
                    minvert=j;
                }
            }
        }
    }
    for (j=0; j<vertex[minvert].nedges; j++) {
        v1x=x-vertex[minvert].x; v1y
            =y-vertex[minvert].y;
        v1z=z-vertex[minvert].z;
        den=sqrt(fsqr(v1x)+fsqr(v1y)+fsqr(v1z));
        v1x/=den; v1y/=den; v1z/=den;
        edgei=vertex[minvert].edge[j];
        if (edge[edgei].vert1 == minvert) {
            v2x=edge[edgei].i;
            v2y=edge[edgei].j;
            v2z=edge[edgei].k;
        }
        else {
            v2x=-1*edge[edgei].i;
            v2y=-1*edge[edgei].j;
            v2z=-1*edge[edgei].k;
        }
        den=sqrt(fsqr(v2x)+fsqr(v2y)+fsqr(v2z));
        v2x/=den; v2y/=den; v2z/=den;
        if (v1x*v2x+v1y*v2y+v1z*v2z > 0) {
            if (obst[i].shape < 0) {
                if (inflag == 0)
                    h=0;
                else
                    h=obst[i].nsides;
                if (distanom(x,y,z,i) < 0.000001)
                    j=vertex[minvert].nedges;
                inflag=1;
                for (h; h<obst[i].nsides; h++) {
                    face=obst[i].side[h];
                    if (bound[face].real == 0) {
                        edgej=bound[face].edge[0];
                        if (edgej < 0)
                            edgej*= -1;
                        edgej--;
                        xstar=(vertex[edge[edgej].vert1].x+
                            vertex[edge[edgej].vert2].x)
                            /2.0;
                        ystar=(vertex[edge[edgej].vert1].y+
                            vertex[edge[edgej].vert2].y)
                            /2.0;
                        zstar=(vertex[edge[edgej].vert1].z+
                            vertex[edge[edgej].vert2].z)
                            /2.0;
                        v1x=x-xstar; v1y=y-ystar; v1z
                            =z-zstar;
                        den=sqrt(fsqr(v1x)+fsqr(v1y)
                            +fsqr(v1z));
                        v1x/=den; v1y/=den; v1z/=den;
                        v2x=v2y=v2z=0.0;
                        for (k=0; k<bound[face].nedge[0]; k+
                            +) {

```

```

edgej=bound[face].edge[k];
if (edgej > 0) {
    edgej--;
    v2x+=
        =vertice[edge[edgej].vert1].x;
    v2y+=
        =vertice[edge[edgej].vert1].y;
    v2z+=
        =vertice[edge[edgej].vert1].z;
}
else {
    edgej=(-1*edgej)-1;
    v2x+=
        =vertice[edge[edgej].vert2].x;
    v2y+=
        =vertice[edge[edgej].vert2].y;
    v2z+=
        =vertice[edge[edgej].vert2].z;
}
}
v2x/=bound[face].nedge[0];
v2y/=bound[face].nedge[0];
v2z/=bound[face].nedge[0];
v2x-=xstar; v2y-=ystar; v2z-=zstar;
den=sqrt(fsqr(v2x)+fsqr(v2y)
    +fsqr(v2z));
v2x/=den; v2y/=den; v2z/=den;
if (v2x*v1x+v2y*v1y+v2z*v1z > 0) {
    n[0]=bound[face].a;
    n[1]=bound[face].b;
    n[2]=bound[face].c;
    dist=n[0]*x+n[1]*y+n[2]*z
        +bound[face].d;
    point[0]=x-dist*n[0];
    point[1]=y-dist*n[1];
    point[2]=z-dist*n[2];
    if (valid_point(point[0],
point[1],point[2],i) ) {
        dist=fabs(dist);
        outside=0; ptr=0;
        for (m=0;
            m<bound[face].numloops;
m++) {
            for (k
                =ptr;k<bound[face].nedge[m]+ptr;
                k++) {
                    if (bound[face].edge[k] < 0)
                    {
                        edgej=(bound[face].edge[k]*
                            -1)-1;
                        p1=edge[edgej].vert2;
                        v2x=-n[1]*edge[edgej].k+
                            n[2]*edge[edgej].j;
                        v2y=-n[2]*edge[edgej].i+
                            n[0]*edge[edgej].k;
                        v2z=-n[0]*edge[edgej].j+
                            n[1]*edge[edgej].i;
                        mid[0]=vertice[p1].x-0.5*
                            edge[edgej].i;
                        mid[1]=vertice[p1].y-0.5*
                            edge[edgej].j;
                        mid[2]=vertice[p1].z-0.5*
                            edge[edgej].k;
                    }
                    else {
                        edgej
                            =bound[face].edge[k]-1;
                        p1=edge[edgej].vert1;
                        v2x=-n[1]*edge[edgej].k-
                            n[2]*edge[edgej].j;
                        v2y=-n[2]*edge[edgej].i-
                            n[0]*edge[edgej].k;
                        v2z=-n[0]*edge[edgej].j-
                            n[1]*edge[edgej].i;
                    }
                }
            }
        }
    }
}

```

```

    n[1]*edge[edgej].i;
    mid[0]=vertice[p1].x+0.5*
    edge[edgej].i;
    mid[1]=vertice[p1].y+0.5*
    edge[edgej].j;
    mid[2]=vertice[p1].z+0.5*
    edge[edgej].k;
  }
  v1x=point[0]-mid[0];
  v1y=point[1]-mid[1];
  v1z=point[2]-mid[2];
  if (v1x*v2x+v1y*v2y+v1z*v2z
    < 0){
    outside=1;
    break;
  }
  ptr=k;
}
if ((dist < minodist) &&
    (outside == 0))
  minodist=dist;
}
}
}
if (j != vertice[minvert].nedges) {
  v1x=vertice[edge[edgei].vert1].x;
  v1y=vertice[edge[edgei].vert1].y;
  v1z=vertice[edge[edgei].vert1].z;
  v2x=vertice[edge[edgei].vert2].x;
  v2y=vertice[edge[edgei].vert2].y;
  v2z=vertice[edge[edgei].vert2].z;
  b[0]=x*v1x+y*v1y+z*v1z;
  b[1]=x*v2x+y*v2y+z*v2z; b[2]=1;
  a[0][0]=fsqr(v1x)+fsqr(v1y)+fsqr(v1z);
  a[0][1]=v1x*v2x+v1y*v2y+v1z*v2z; a[0][2]
  =1;
  a[1][0]=v1x*v2x+v1y*v2y+v1z*v2z;
  a[1][1]=fsqr(v2x)+fsqr(v2y)+fsqr(v2z);
  a[1][2]=1; a[2][0]=a[2][1]=1; a[2][2]=0;
  det=a[0][0]*(a[1][1]*a[2][2]-a[2][1]
    *a[1][2])
    -a[1][0]*(a[0][1]*a[2][2]-a[2][1]
    *a[0][2])
    +a[2][0]*(a[0][1]*a[1][2]-a[0][2]
    *a[1][1]);
  a1[0][0]=(a[1][1]*a[2][2]-a[2][1]
    *a[1][2])/det;
  a1[1][0]=(a[2][0]*a[1][2]-a[1][0]
    *a[2][2])/det;
  a1[2][0]=(a[1][0]*a[2][1]-a[2][0]
    *a[1][1])/det;
  a1[0][1]=(a[2][1]*a[0][2]-a[0][1]
    *a[2][2])/det;
  a1[1][1]=(a[0][0]*a[2][2]-a[2][0]
    *a[0][2])/det;
  a1[2][1]=(a[2][0]*a[0][1]-a[2][1]
    *a[0][0])/det;
  a1[0][2]=(a[0][1]*a[1][2]-a[1][1]
    *a[0][2])/det;
  a1[1][2]=(a[1][0]*a[0][2]-a[0][0]
    *a[1][2])/det;
  a1[2][2]=(a[0][0]*a[1][1]-a[0][1]
    *a[1][0])/det;
  lambda1=a1[0][0]*b[0]+a1[0][1]*b[1]
    +a1[0][2]*b[2];
  lambda2=a1[1][0]*b[0]+a1[1][1]*b[1]
    +a1[1][2]*b[2];
  if ((0 <= lambda1) && (0 <= lambda2)) {
    xstar=lambda1*v1x+(1-lambda1)*v2x;
    ystar=lambda1*v1y+(1-lambda1)*v2y;

```

```

    zstar=lambda1*v1z+(1-lambda1)*v2z;
    dist=sqrt(fsqr(x-xstar)+
      fsqr(y-ystar)
      +fsqr(z-zstar));
    if (dist < minodist)
      minodist=dist;
    /* Already considered every face */
    if (obst[i].shape < 0)
      l=edge[edgei].nfaces;
    else
      l=0;
    for (l<edge[edgei].nfaces; l++) {
      v1x=x-xstar; v1y=y-ystar; v1z
      =z-zstar;
      den=sqrt(fsqr(v1x)+fsqr(v1y)
        +fsqr(v1z));
      v1x/=den; v1y/=den; v1z/=den;
      v2x=v2y=v2z=0.0;
      face=edge[edgei].face[l];
      for (k=0; k<bound[face].nedge[0];
        k++) {
        edgej=bound[face].edge[k];
        if (edgej > 0) {
          edgej--;
          v2x+
            =vertice[edge[edgej].vert1].x;
          v2y+
            =vertice[edge[edgej].vert1].y;
          v2z+
            =vertice[edge[edgej].vert1].z;
        }
        else {
          edgej=(-1*edgej)-1;
          v2x+
            =vertice[edge[edgej].vert2].x;
          v2y+
            =vertice[edge[edgej].vert2].y;
          v2z+
            =vertice[edge[edgej].vert2].z;
        }
      }
      v2x/=bound[face].nedge[0];
      v2y/=bound[face].nedge[0];
      v2z/=bound[face].nedge[0];
      v2x=-xstar; v2y=-ystar; v2z-
      =zstar;
      den=sqrt(fsqr(v2x)+fsqr(v2y)
        +fsqr(v2z));
      v2x/=den; v2y/=den; v2z/=den;
      if (v2x*v1x+v2y*v1y+v2z*v1z > 0) {
        n[0]=bound[face].a; n[1]
        =bound[face].b;
        n[2]=bound[face].c;
        dist=n[0]*x+n[1]*y+n[2]*z
          +bound[face].d;
        point[0]=x-dist*n[0];
        point[1]=y-dist*n[1];
        point[2]=z-dist*n[2];
        if (valid_point(point[0],
          point[1],point[2],i)) {
          dist=fabs(dist); outside=0;
          ptr=0;
          for (m=0;
            m<bound[face].numloops;
            m++) {
            for (k
              =ptr;k<bound[face].nedge[m]+ptr;
              k++) {
              if (bound[face].edge[k] < 0) {
                edgej=(bound[face].edge[k]*-1)-1;
                p1=edge[edgej].vert2;
                v2x=-n[1]*edge[edgej].k+
                  n[2]*edge[edgej].j;

```



```

v2y=-n[2]*edge[edgej].i+
    n[0]*edge[edgej].k;
v2z=-n[0]*edge[edgej].j+
    n[1]*edge[edgej].i;
mid[0]=vertice[p1].x-0.5*
    edge[edgej].i;
mid[1]=vertice[p1].y-0.5*
    edge[edgej].j;
mid[2]=vertice[p1].z-0.5*
    edge[edgej].k;
}
else {
    edgej=bound[face].edge[k]-1;
    p1=edge[edgej].vert1;
    v2x=n[1]*edge[edgej].k-
        n[2]*edge[edgej].j;
    v2y=n[2]*edge[edgej].i-
        n[0]*edge[edgej].k;
    v2z=n[0]*edge[edgej].j-
        n[1]*edge[edgej].i;
    mid[0]=vertice[p1].x+0.5*
        edge[edgej].i;
    mid[1]=vertice[p1].y+0.5*
        edge[edgej].j;
    mid[2]=vertice[p1].z+0.5*
        edge[edgej].k;
}
v1x=point[0]-mid[0];
v1y=point[1]-mid[1];
v1z=point[2]-mid[2];
if (v1x*v2x+v1y*v2y+v1z*v2z
    < 0) {
    outside=1;
    break;
}
}
ptr=k;
}
if ((dist < minodist) &&
    (outside == 0))
    minodist=dist;
}
}
}
}
}
}
}
}
if (minodist < mindist) {
    mindist=minodist;
    minobst=i;
}
}
}
if (obst[minobst].shape > 0) {
    if (distanom(x,y,z,minobst) < 0.00001)
        mindist*=-1;
}
else {
    if (distanom(x,y,z,minobst) > 0.00001)
        mindist*=-1;
}
return(mindist);
}
/* bordered_by- Return t if edge i borders face
   face */
bordered_by(face,i)
int face,i;
{
    int j;

    for (j=0; j<edge[i].nfaces; j++)

```

```

    if (edge[i].face[j] == face)
        return(1);
    return(0);
}

/* damn_close- i is close enough to j to assume
they are the same */
damn_close(i,j)
int i,j;
{
    if (fabs(vertex[i].x-vertex[j].x)+
        fabs(vertex[i].y-vertex[j].y)+
        fabs(vertex[i].z-vertex[j].z) < 0.01)
        return(1);
    return(0);
}

/* Return the obst containing the vertex ver if
this
obst is not an exception */
onobst(ver)
int ver;
{
    int sptr, eptr, i, j, k, l, ptr, exc, ourobst;

    ourobst=bound[edge[vertex[ver].edge[0]]
        .face[0].obstacle;
    return(ourobst);
}

/* valid_point- Return t if (x,y,z) is valid point
on obst i */
valid_point(x,y,z,i)
double x,y,z;
int i;
{
    int m,flag;

    flag=1;
    if (obst[i].shape < 0) {
        for (m=0; m<nobs; m++) {
            if ((m != i) && (obst[m].shape < 0))
                if (dismom(x,y,z,m) <= 0.00001)
                    return(0);
        }
    }
    else {
        for (m=0; m<obst[i].nholes; m++)
            if (dismom(x,y,z,obst[i].holes[m]) <=
                0.00001)
                return(0);
    }
    return(1);
}

/* This file contains code for calculating
min_dist
by the method outlined in thesis. This routine
is
called min_dist(x,y,z). This file also contains
the
point potential function, object potential
function,
and code to find initial path from the mpv
found */
#include <stdio.h>
#include <math.h>
#include "common.h"
#define cube(x) ((x) * (x) * (x))
#define du 0.00001
#define dv 0.00001
#define step 0.00001

```

```

typedef struct minclust {
    int j,k;
    struct minclust *next;
} minclust;

extern double objectpot(), md_internal();
extern int collisionflag,nbounds;
extern double maxmo;
extern int ndim;
extern double fsqr();
extern obstacle *obst;
extern int nob;
extern side *bound;
extern line *edge;
extern vertex *vertex;
extern double distnom(), fastmin_dist();
extern double distnom_over();
extern int tpoint(), movable(), tform();
extern int objside;
extern double objbound[6];
double dist_to_hole(), grad_search();
double Rx(), Ry(), Rz();
double dist_to_any_hole();
double find_closest_vertex();
double dist_to_intersect();
double distnom(), distnom_over();
double pointpot(), min_dist();
oside *object;

/* Calculate the inverse of the potential of
a single point */
double min_dist(x,y,z)
double x,y,z;
{
    int i,j,k,cobst,jp,kp;
    double jsum,fsum,dti,mdti;

    fsum=200000.0;
    for (i=0; i<nob; i++) {
/* First calc potential w r t ith obstacle for
all
obsts that are not holes and are to be
considered */
        if ((obst[i].shape >= 0) &&
            (obst[i].consider == 1)) {
            jsum=distnom(x,y,z,i);
            /* Check hole situation */
            if (jsum < 0.000001) {
                /* Calc distance to void if greater than 0 */
                jsum=dist_to_any_hole(x,y,z,i,0.0);
                /* Not in any hole, collision, quit */
                if (jsum < 0)
                    return(0);
            }
            else {
                /* Get exact distance to obstacle */
                mdti=dist_to_any_hole(x,y,z,i,jsum);
                if (mdti < 0) {
                    mdti=2000;
                    for (j=0; j<obst[i].nsides; j++) {
                        jp=obst[i].side[j];
                        if (bound[jp].dist > 0.00001) {
                            for (k=j+1; k<obst[i].nsides; k++) {
                                kp=obst[i].side[k];
                                if (bound[kp].dist > 0.00001) {
                                    dti=dist_to_intersect(jp,kp,
                                        x,y,z);
                                    if ((dti > 0) && (dti < mdti))
                                        mdti=dti;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    if (mdti < 2000)
        jsum=mdti;
    }
    if (jsum < fsum) {
        fsum=jsum;
    }
}
/* Now take care of case when point over hole
*/
return(fsum);
}

double dist_to_any_hole(x,y,z,cobst,dist)
double x,y,z,dist;
int cobst;
{
    int i,j,flag;
    double *X,t;

    /* Find closest hole if one is as close or
    closer than cobst */
    flag= -1;
    X=(double *)calloc(obst[cobst].nholes
        ,sizeof(double));
    for (i=0; i<obst[cobst].nholes; i++) {
        X[i]=distnom_over(x,y,z,obst[cobst].holes[i]);
        if (dist-X[i] > -0.0000001) {
            dist=X[i]; flag=obst[cobst].holes[i];
            i=obst[cobst].nholes;
        }
    }
}

/* If point in obstacle, only consider holes in
obstacle.
If nearest hole is farther than cobst, holes
don't
effect potential */
if (flag == -1) {
    free(X);
    return(-1);
}

/* Now find distance to surface of that hole *
/
t=dist_to_hole(x,y,z,&flag,X,1,cobst);
free(X);
return(t);
}

double dist_to_hole(x,y,z,ch,X,nh,cobst)
double x,y,z,*X;
int *ch,nh,cobst;
{
    int i,j,k,sptr,optr,chole,*tchole,inside,ch2;
    double dist,t,tsum,xstar,ystar,zstar,tt,gy;

    chole=ch[0];
    dist=X[chole]; tsum=20000;
    for (i=0; i<obst[chole].nsides; i++) {
        sptr=obst[chole].side[i];
        /* t < 0 if on nonmaterial side of face */
        t=bound[sptr].a*x+bound[sptr].b*y+
            bound[sptr].c*z+bound[sptr].d;
        /* If t > 0, over face, can't be valid point */
        if (t <= 0) {
            /* Find closest point */
            xstar=x-bound[sptr].a*t;
            ystar=y-bound[sptr].b*t;
            zstar=z-bound[sptr].c*t;
            inside=1; t=1000;

```

```

/* See if point inside of face */
for (j=0; j<obst[chole].nsides; j++) {
optr=obst[chole].side[j];
/* Point not inside face */
if ((optr != sprtr) && ((gy=
bound[optr].a*xstar+bound[optr].b*ystar+
bound[optr].c*zstar+bound[optr].d) >
0.0000001)) {
tt=dist_to_intersect(sptr,optr,x,y,z);
if (tt < t) t=tt; inside=0;
}
if ((optr != sprtr) && (fabs(gy) < 0.0000001)) {
inside=0;
tt=sqrt(fsqr(x-xstar)+fsqr(y-ystar)
+fsqr(z-zstar));
if (tt < t) t=tt;
}
}
/* If point within face */
if (inside == 1) {
inside=0;
/* See if point is valid */
/* Check to see if on obst surface */
for (j=0; j<obst[cobst].nsides; j++) {
optr=obst[cobst].side[j];
if (fabs(bound[optr].a*xstar+
bound[optr].b*ystar+
bound[optr].c*zstar+
bound[optr].d) < 0.000001) {
inside=1; j=obst[cobst].nsides; t=1000;
}
}
/* If not on obst, check to see if in any obst
in obst */
if (inside == 1)
j=obst[cobst].nholes;
else
j=0;
for (j; j<obst[cobst].nholes; j++) {
ch2=obst[cobst].holes[j];
/* Point is not valid, also in ch2 */
if ((ch2 != chole) &&
(distnom(xstar,ystar,zstar,ch2)
< 0.000001)) {
inside=1;
/* Check for endless recursion */
if (!one_of(ch2,ch,nh)) {
tchole=(int *)calloc(nh+1,sizeof(int));
tchole[0]=ch2;
for (k=0; k<nh; k++)
tchole[k+1]=ch[k];
/* Recursively calculate dist to ch2 */
t=dist_to_hole(x,y,z,tchole,X,nh+1,cobst);
free(tchole);
j=obst[cobst].nholes;
}
}
}
/* If not inside any other hole, valid point */
if (inside == 0)
t=sqrt(fsqr(x-xstar)+fsqr(y-ystar)
+fsqr(z-zstar));
if (fabs(t) < tsum)
tsum=fabs(t);
}
return(tsum);
}
one_of(i,list,num)
int i,*list,num;
{

```

```

int j;
for (j=0; j<num; j++)
if (i == list[j])
return(1);
return(0);
}
double dist_to_intersect(i,j,x,y,z)
int i,j;
double x,y,z;
{
double dx,dy,dz,v1x,v2x,v1y,v2y,v1z,v2z,b[3]
,a[3][3];
double a1[3][3],det,lambd1,lambd2,xstar,ystar
,zstar;
int k,l,tk,tl,edgek,edgel;

tk=tl=0;
for (k=0; k<bound[i].numloops; k++)
tk=bound[i].nedge[k];
for (l=0; l<bound[j].numloops; l++)
tl=bound[j].nedge[l];
for (k=0; k<tk; k++) {
edgek=abs(bound[i].edge[k])-1;
for (l=0; l<tl; l++) {
edgel=abs(bound[j].edge[l])-1;
if (edgek == edgel) {
k=tk; l=tl;
}
}
}
if (edgek != edgel)
return(-1);
v1x=vertice[edge[edgek].vert1].x;
v1y=vertice[edge[edgek].vert1].y;
v1z=vertice[edge[edgek].vert1].z;
v2x=vertice[edge[edgek].vert2].x;
v2y=vertice[edge[edgek].vert2].y;
v2z=vertice[edge[edgek].vert2].z;

b[0]=x*v1x+y*v1y+z*v1z; b[1]=x*v2x+y*v2y+z*v2z;
b[2]=1; a[0][0]=fsqr(v1x)+fsqr(v1y)+fsqr(v1z);
a[0][1]=v1x*v2x+v1y*v2y+v1z*v2z;
a[0][2]=1;
a[1][0]=v1x*v2x+v1y*v2y+v1z*v2z;
a[1][1]=fsqr(v2x)+fsqr(v2y)+fsqr(v2z);
a[1][2]=1;
a[2][0]=a[2][1]=1; a[2][2]=0;
det=a[0][0]*(a[1][1]*a[2][2]-a[2][1]*a[1][2])
-a[1][0]*(a[0][1]*a[2][2]-a[2][1]*a[0][2])
+a[2][0]*(a[0][1]*a[1][2]-a[0][2]*a[1][1]);
a1[0][0]=(a[1][1]*a[2][2]-a[2][1]*a[1][2])/det;
a1[1][0]=(a[2][0]*a[1][2]-a[1][0]*a[2][2])/det;
a1[2][0]=(a[1][0]*a[2][1]-a[2][0]*a[1][1])/det;

a1[0][1]=(a[2][1]*a[0][2]-a[0][1]*a[2][2])/det;
a1[1][1]=(a[0][0]*a[2][2]-a[2][0]*a[0][2])/det;
a1[2][1]=(a[2][0]*a[0][1]-a[2][1]*a[0][0])/det;

a1[0][2]=(a[0][1]*a[1][2]-a[1][1]*a[0][2])/det;
a1[1][2]=(a[1][0]*a[0][2]-a[0][0]*a[1][2])/det;
a1[2][2]=(a[0][0]*a[1][1]-a[0][1]*a[1][0])/det;

lambd1=a1[0][0]*b[0]+a1[0][1]*b[1]+a1[0][2]*
*b[2];
lambd2=a1[1][0]*b[0]+a1[1][1]*b[1]+a1[1][2]*
*b[2];
if ((lambd1 > 0) && (lambd2 > 0)) {
xstar=lambd1*v1x+(1-lambd1)*v2x;
ystar=lambd1*v1y+(1-lambd1)*v2y;
zstar=lambd1*v1z+(1-lambd1)*v2z;
}
}

```

```

else if (lambda2 <= 0) {
    xstar=v1x; ystar=v1y; zstar=v1z;
}
else {
    xstar=v2x; ystar=v2y; zstar=v2z;
}
return(sqrt(fsq(x-xstar)+fsq(y-ystar)
+fsq(z-zstar)));
}

/* distnom- This func is used to calculate the
distance
between the point (x,y,z) and the nominal
obstacle,
the distance func defined in any of my reports
being
most closely related to a city block distance */
double distnom(x,y,z,i)
double x,y,z;
int i; /* Consider the ith obstacle */
{
    double sum;
    int j,sptr;

    sum=0;
    for (j=0; j < obst[i].nsides; j++) {
        sptr=obst[i].side[j];
        bound[sptr].dist=bound[sptr].a*x+bound[sptr].b
*y
        +bound[sptr].c*z+bound[sptr].d;
        if (bound[sptr].dist > sum)
            sum=bound[sptr].dist;
    }
    return(sum);
}

double distnom_over(x,y,z,i)
double x,y,z;
int i; /* Consider the ith obstacle */
{
    double sum,t;
    int j,sptr;

    sum=0;
    for (j=0; j < obst[i].nsides; j++) {
        sptr=obst[i].side[j];
        sum+=bound[sptr].a*x+bound[sptr].b*y+
        bound[sptr].c*z+bound[sptr].d;
        sum+=fabs(bound[sptr].a*x+bound[sptr].b*y+
        bound[sptr].c*z+bound[sptr].d);
    }
    return(sum/2.0);
}

/* Returns 1 if nearest point to x,y,z on obstside
happens
at a point where the surface exists. Some faces
of a
hole may be partially real, (denoting
transition from
free space to obstacle) and partially false
(denoting,
say the face between this hole and another,
i.e.
entrance of a hole) */
real_surf_solid(obstside,x,y,z)
int obstside;
double x,y,z;
{
    int i,j,sptr,theobst,hole;

    theobst=bound[obstside].obstacle;

```

```

for (i=0; i<obst[theobst].nholes; i++) {
    hole=obst[theobst].holes[i];
    if (distnom(x,y,z,hole) < 0.00001)
        return(0);
    }
    return(1);
}

double pointpot(x,y,z)
double x,y,z;
{
    double pot;
    pot=min_dist(x,y,z);
    return(1/(DELTA+pot));
}

/* Get the clearance of the part by searching for
the
minimum distance between a point on object and
a point on the obstacles */
double clearance(x,y,z,rx,ry,rz,th,lx,ly,lz,fast)
double x,y,z,rx,ry,rz,th,lx,ly,lz;
int fast;
{
    double ox,oy,oz,**samp,dist;
    double nx,ny,nz,smallest,su,sv;
    double lx1,ly1,lz1,lx2,ly2,lz2,lx3,ly3,lz3;
    int nearmin[1000][2],j,k,l,dof,sptr,sj,numclust;
    struct minclust *cluster[200],*thead;

    samp=(double **)calloc(objside,sizeof(double
    *));
    /* Special consideration of flexible obstacles */
    /* This is explained in Chapter 9 of thesis */
    for (sptr=0; sptr<nbounds; sptr++) {
        if (bound[sptr].compflag == 1) {
            for (j=0; j<objside; j++) {
                bound[sptr].compressed=0;
                dof=movable(j);
                if (dof != -1)
                    tform(object[j].samples[0][0],
                    object[j].samples[0][1],
                    object[j].samples[0][2],
                    &ox,&oy,&oz,dof,th);
                else {
                    ox=object[j].samples[0][0];
                    oy=object[j].samples[0][1];
                    oz=object[j].samples[0][2];
                }
                tpoint(lx,ly,lz,rx,ry,rz,ox,oy,oz,
                    &lx1,&ly1,&lz1);
                tpoint(x,y,z,rx,ry,rz,ox,oy,oz,
                    &lx2,&ly2,&lz2);
                lx3=lx2-lx1; ly3=ly2-ly1; lz3=lz2-lz1;
                for (k=0; k<object[j].numsamp; k++) {
                    if (dof != -1)
                        tform(object[j].samples[k][0],
                        object[j].samples[k][1],
                        object[j].samples[k][2],
                        &ox,&oy,&oz,dof,th);
                    else {
                        ox=object[j].samples[k][0];
                        oy=object[j].samples[k][1];
                        oz=object[j].samples[k][2];
                    }
                    tpoint(x,y,z,rx,ry,rz,
                        ox,oy,oz,&nx,&ny,&nz);
                    if (moved(sptr,nx,ny,nz,lx3,ly3,lz3)
                        == 1) {
                        bound[sptr].compressed=1;
                        bound[sptr].d+=bound[sptr].compress;
                    }
                }
            }
        }
    }
}

```

```

    }
}
/* Start of actual calculation */
smallest=1000;
for (j=0; j<objside; j++) {
    samp[j]=(double *)
        calloc(object[j].numsamp,sizeof(double));
    dof=movable(j);
    for (k=0; k<object[j].numsamp; k++) {
        /* Transform sampling point to object pose */
        if (dof != -1)
            tform(object[j].samples[k][0],
                object[j].samples[k][1],
                object[j].samples[k][2],
                &ox,&oy,&oz,dof,th);
        else {
            ox=object[j].samples[k][0];
            oy=object[j].samples[k][1];
            oz=object[j].samples[k][2];
        }
        tpoint(x,y,z,rx,ry,rz,
            ox,oy,oz,&nx,&ny,&nz);

        /* Calculate minimum distance */
        if (fast == 0)
            samp[j][k]=min_dist(nx,ny,nz);
        /* If we are just trying to decide if a
        collision has occurred or not, not
        worried about exact distance */
        else
            smallest=fastmin_dist(nx,ny,nz);
        if (samp[j][k]<smallest) {
            smallest=samp[j][k];
        }
        /* Return immediately if collision */
        if (smallest < 0.00000001) {
            for (sptr=0; sptr < nbounds; sptr++)
                if (bound[sptr].compressed == 1) {
                    bound[sptr].compressed=0;
                    bound[sptr].d-=bound[sptr].compress;
                }
            for (j=0; j<objside; j++)
                free(samp[j]);
            free(samp);
            return(smallest);
        }
    }
}
/* Form near minimum set */
l=0;
for (j=0; j<objside; j++)
    for (k=0; k<object[j].numsamp; k++)
        if (samp[j][k] < smallest+eta) {
            nearmin[l][0]=j;
            nearmin[l][1]=k;
            l++;
        }
/* Perform gradient search at each cluster */
for (j=0; j<l; j++) {
    su=object[nearmin[j][0]].samples[nearmin[j][1]][3];
    sv=object[nearmin[j][0]].samples[nearmin[j][1]][4];

    dist=grad_search(su,sv,nearmin[j][0],x,y,z,rx,
        ry,rz,th);
    if (dist < smallest) {
        smallest=dist;
        /* Return immediately if collision */
        if (smallest < 0.00000001) {
            for (sptr=0; sptr < nbounds; sptr++)
                if (bound[sptr].compressed == 1) {

```

```

                    bound[sptr].compressed=0;
                    bound[sptr].d-=bound[sptr].compress;
                }
            for (j=0; j<objside; j++)
                free(samp[j]);
            free(samp);
            return(smallest);
        }
    }
}
for (sptr=0; sptr < nbounds; sptr++)
    if (bound[sptr].compressed == 1) {
        bound[sptr].compressed=0;
        bound[sptr].d-=bound[sptr].compress;
    }
for (j=0; j<objside; j++)
    free(samp[j]);
free(samp);
return(smallest);
}

double grad_search(u,v,f,x,y,z,rx,ry,rz,th)
double u,v,x,y,z,rx,ry,rz,*th;
int f;
{
    int dof;
    double dist1,dist2,ox,oy,oz;
    double dx,dy,dz,mag;
    double dddu,dddv,nxp,nyp,nzp,nx,ny,nz;

    dof=movable(f);
    if (dof != -1)
        tform(Rx(u,v,f),Ry(u,v,f),Rz(u,v,f),
            &ox,&oy,&oz,dof,th);
    else {
        ox=Rx(u,v,f); oy=Ry(u,v,f); oz=Rz(u,v,f);
    }
    tpoint(x,y,z,rx,ry,rz,
        ox,oy,oz,&nxp,&nyp,&nzp);
    dist2=min_dist(nxp,nyp,nzp);
    do {
        dist1=dist2; nx=nxp; ny=nyp; nz=nzp;
        if (dof != -1)
            tform(Rx(u+du,v,f)-nx,Ry(u+du,v,f)-ny,
                Rz(u+du,v,f)-nz,&ox,&oy,&oz,dof,th);
        else {
            ox=Rx(u+du,v,f)-nx; oy=Ry(u+du,v,f)-ny;
            oz=Rz(u+du,v,f)-nz;
        }
        tpoint(x,y,z,rx,ry,rz,ox,oy,oz,&dx,&dy,&dz);
        dddu=min_dist(nx+dx,ny+dy,nz+dz)-
            min_dist(nx-dx,ny-dy,nz-dz);
        if (dof != -1)
            tform(Rx(u+dv,v,f)-nx,Ry(u+dv,v,f)-ny,
                Rz(u+dv,v,f)-nz,&ox,&oy,&oz,dof,th);
        else {
            ox=Rx(u+dv,v,f)-nx; oy=Ry(u+dv,v,f)-ny;
            oz=Rz(u+dv,v,f)-nz;
        }
        tpoint(x,y,z,rx,ry,rz,ox,oy,oz,&dx,&dy,&dz);
        dddv=min_dist(nx+dx,ny+dy,nz+dz)-
            min_dist(nx-dx,ny-dy,nz-dz);
        mag=sqrt(ddduddu+dddv*dddv);
        dddu/=mag; dddv/=mag;
        u-=step*dddv; v-=step*dddu;
        if (dof != -1)
            tform(Rx(u,v,f),Ry(u,v,f),Rz(u,v,f),
                &ox,&oy,&oz,dof,th);
        else {
            ox=Rx(u,v,f); oy=Ry(u,v,f); oz=Rz(u,v,f);
        }
        tpoint(x,y,z,rx,ry,rz,
            ox,oy,oz,&nxp,&nyp,&nzp);
    }
}

```

```

    if (in_face(nxp,nyp,nzp,f) == 0)
        return(dist1);
    dist2=min_dist(nxp,nyp,nzp);
    } while (dist2 < dist1);
    return(dist1);
}

```

/* in_face- Test to see if point x,y,z is inside face

j by applying the sum of angles test */

in_face(x,y,z,j)

double x,y,z;

int j;

```

{
    int l,next,k,ptr;
    double si,co,sum,angle;
    double v1x,v1y,v1z,v2x,v2y,v2z;

```

```

    ptr=0;
    sum=0;
    for (l=0; l<object[j].numloops; l++) {
        for (k=ptr; k<object[j].nvert[l]+ptr; k++) {
            next=((k-ptr)+1) % object[j].nvert[l]+ptr;
            v1x=object[j].x[next]-x; v1y
            =object[j].y[next]-y;
            v1z=object[j].z[next]-z; v2x
            =object[j].x[k]-x;
            v2y=object[j].y[k]-y; v2z=object[j].z[k]-z;
            if ((fabs(fsqr(v1x)+fsqr(v1y)+fsqr(v1z)) <
                0.000000001) ||
                (fabs(fsqr(v2x)+fsqr(v2y)+fsqr(v2z)) <
                0.000000001))
                return(1);
            si=(v1y*v2z-v1z*v2y)*object[j].nx+
            (v1z*v2x-v1x*v2z)*object[j].ny+
            (v1x*v2y-v1y*v2x)*object[j].nz;
            co=v1x*v2x+v1y*v2y+v1z*v2z;
            /* If si=0, co=0 point is on edge vertex,
            force to be inside */
            angle=atan2(si,co);
            if (fabs(angle- M_PI) < 0.000000001)
                return(1);
            sum+=angle;
        }
        ptr+=k;
    }
    /* If inside should be multiple of 2pi, else
    zero */
    if (fabs(sum) > 1)
        return(1);
    return(0);
}

```

double Rx(u,v,f)

double u,v;

int f;

```

{
    return(object[f].r[0][0]*u+
        object[f].r[0][1]*v+
        object[f].r[0][2]);
}

```

double Ry(u,v,f)

double u,v;

int f;

```

{
    return(object[f].r[1][0]*u+
        object[f].r[1][1]*v+
        object[f].r[1][2]);
}

```

double Rz(u,v,f)

double u,v;

int f;

```

{
    return(object[f].r[2][0]*u+
        object[f].r[2][1]*v+
        object[f].r[2][2]);
}

```

/* cross_product- Returns m cross n */

cross_product(mx,my,mz,nx,ny,nz,cx,cy,cz)

double mx,my,mz,nx,ny,nz,*cx,*cy,*cz;

```

{
    *cx=my*nz-ny*mz;
    *cy=mz*nx-nz*mx;
    *cz=mx*ny-nx*my;
}

```

/* Find initial path between nodes generated by

mpv */

initpath(st,path,cols,initial)

struct node *st;

double ***path;

struct colnode **cols;

int initial;

```

{
    struct node *ptr, *next, *last, *new;
    double **vel;
    int i,j,k,n,npoints,dim,num;
    double x1,y1,x2,y2,z1,z2,distance,mag,mindist
    ,alpha;
    double rotx1,roty1,rotz1,rotx2,roty2,rotz2,*t1,
    *t2;
    double tempvrx,tempvry,tempvrz,trx1,try1,trz1
    ,tr1[10];
    double totdist, tempvr[10],*tv;
    struct colnode *current, *temp;

```

ptr=st; totdist=0.0;

npoints=0; (*cols)=NULL;

/* Assume no obstacles will be considered */

for (i=0; i<nobs; i++)

obst[i].consider=0;

if (initial)

rotx1=ptr->rotx; roty1=ptr->roty; rotz1

=ptr->rotz;

t1=ptr->t;

while (ptr->next[0] != NULL) {

/* Next node is last node, ptr->md not necessarily

distance */

if (ptr->next[0]->next[0] == NULL) {

last=ptr->next[0];

ptr->md=sqrt(fsqr(ptr->x-last->x)+

fsqr(ptr->y-last->y)+fsqr(ptr->z-last->z));

else

ptr->md=sqrt(fsqr(ptr->x-ptr->next[0]->x)+

fsqr(ptr->y-ptr->next[0]->y)+

fsqr(ptr->z-ptr->next[0]->z));

npoints++;

totdist+=ptr->md;

ptr=ptr->next[0];

/* Now, find obstacles close enough to

consider */

mindist=1000;

for (i=0; i<nobs; i++) {

obst[i].dist=distnom_over(ptr->x,ptr->y

,ptr->z,i);

if (obst[i].dist < mindist)

mindist=obst[i].dist;

}

if (mindist < maxmo)

```

mindist=2*maxmo;
for (i=0; i<nobs; i++)
    if (obst[i].dist < mindist*1.5)
        obst[i].consider=1;
}

npoints++;
num=npoints*5;
vel=(double **)calloc(ndim,sizeof(double *));
(*path)=(double **)calloc(ndim,sizeof(double *));
for (j=0; j<ndim; j++)
    vel[j]=(double *)calloc(num,sizeof(double));

if (initial) {
    rotx2=last->rotx; roty2=last->roty; rotz2=last->rotz;
    t2=last->t;
    for (j=0; j<ndim-6; j++)
        tempvrx[j]=(t2[j]-t1[j])/totdist;
    tempvrx=(rotx2-rotx1)/totdist;
    tempvry=(roty2-roty1)/totdist;
    tempvrz=(rotz2-rotz1)/totdist;
    trx1=rotx1;
    try1=roty1;
    trz1=rotz1;
    ptr=st;
    while (ptr->next[0]->next[0] != NULL) {
        ptr->next[0]->rotx=ptr->rotx+tempvrx*ptr->md;
        ptr->next[0]->roty=ptr->roty+tempvry*ptr->md;
        ptr->next[0]->rotz=ptr->rotz+tempvrz*ptr->md;
        for (j=0; j<ndim-6; j++)
            ptr->next[0]->t[j]=ptr->t[j]+tempvrx[j]*ptr->md;
        ptr=ptr->next[0];
    }
    i=0; ptr=st;
    while (ptr->next[0] != NULL) {
        next=ptr->next[0];
        /* Use vel as temporary storage of path */
        vel[0][i]=ptr->x; vel[1][i]=ptr->y; vel[2][i]=ptr->z;
        vel[3][i]=ptr->rotx; vel[4][i]=ptr->roty;
        vel[5][i]=ptr->rotz;
        for (dim=6; dim<ndim; dim++)
            vel[dim][i]=ptr->t[dim-6];
        if (i == 0)
            distance=clearance(vel[0][i],vel[1][i],vel[2][i],
                                vel[3][i],vel[4][i],vel[5][i],
                                vel[0][i],vel[1][i],vel[2][i],ptr->t,0);
        else
            distance=clearance(vel[0][i],vel[1][i],vel[2][i],
                                vel[3][i],vel[4][i],vel[5][i],
                                vel[0][i-1],vel[1][i-1],vel[2][i-1],ptr->t,0);
        if ((*cols) == NULL) {
            current=(struct colnode *)
                calloc(1,sizeof(struct colnode));
            current->collision=distance;
            (*cols)=current;
        }
        else {
            temp=(struct colnode *)
                calloc(1,sizeof(struct colnode));
            temp->collision=distance;
            current->next=temp;
        }
        current=temp;
    }
    current=temp;
}
if ((distance < 0.000001) && (i != 0)) {
    collisionflag++;
    printf("distance= %f, z= %f0,distance\n",vel[2][i]);
}
alpha=atan2(distance,maxmo);
rotx1=ptr->next[0]->rotx-ptr->rotx;
roty1=ptr->next[0]->roty-ptr->roty;
rotz1=ptr->next[0]->rotz-ptr->rotz;
mag=fabs(rotx1)+fabs(roty1)+fabs(rotz1);
if ((mag > alpha) && (distance > 0.00001))
    distance*=(alpha/mag);
if ((distance < 0.1) && (distance > 0.00000001))
    distance=0.1;
if ((distance < ptr->md) && (distance > 0.000001)) {
    npoints++;
    if (npoints == num) {
        num *= 2;
        tv=(double *)calloc(num,sizeof(double));
        for (dim=0; dim<ndim; dim++) {
            for (k=0; k<num; k++)
                tv[k]=vel[dim][k];
            free(vel[dim]);
            vel[dim]=(double *)calloc(num,sizeof(double));
            for (k=0; k<num; k++)
                vel[dim][k]=tv[k];
        }
        free(tv);
    }
    new=(struct node *)calloc(1,sizeof(struct node));
    new->md=ptr->md-distance;
    mag=sqrt(fsqr(next->x-ptr->x)+fsqr(next->y-ptr->y)+fsqr(next->z-ptr->z));
    new->x=ptr->x+(distance/mag)*(next->x-ptr->x);
    new->y=ptr->y+(distance/mag)*(next->y-ptr->y);
    new->z=ptr->z+(distance/mag)*(next->z-ptr->z);
    new->rotx=ptr->rotx+(distance/mag)*(next->rotx-ptr->rotx);
    new->roty=ptr->roty+(distance/mag)*(next->roty-ptr->roty);
    new->rotz=ptr->rotz+(distance/mag)*(next->rotz-ptr->rotz);
    for (dim=0; dim<ndim-6; dim++)
        new->t[dim]=ptr->t[dim]+distance*(ptr->next[0]->t[dim]-ptr->t[dim]);
    ptr->next[0]=new;
    next->next[0]=next;
    new=next;
    i++;
    ptr=next;
}
vel[0][i]=ptr->x; vel[1][i]=ptr->y; vel[2][i]=ptr->z;
vel[3][i]=ptr->rotx; vel[4][i]=ptr->roty;
vel[5][i]=ptr->rotz;
for (dim=6; dim<ndim; dim++)
    vel[dim][i]=ptr->t[dim-6];
}
/* Now smooth path generated and place in array
path */
for (dim=0; dim<ndim; dim++)
    (*path)[dim]=(double *)
        calloc(npoints+2,sizeof(double));
for (i=0; i<npoints; i++) {
    if ((i == 0) || (i == npoints-1)) {

```

```

    for (dim=0; dim<ndim; dim++)
        (*path)[dim][i]=vel[dim][i];
    }
    else {
        for (dim=0; dim<ndim; dim++)
            (*path)[dim][i]=vel[dim][i];
    }
}

free(vel);
return(npoints);
}

/* This file contains for stage 2 local (serial)
optimization */
#include <stdio.h>
#include <math.h>
#include "common.h"

int SCALE=0.1;
extern double objectpot(),pointpot();
extern double fsqr(),maxmo;
extern int ndim,nbounds;
extern int objside;
extern oside *object;
extern transform *joint;
extern int numjoints, rigid;
extern int collisionflag;
extern side *bound;
extern int nob;
extern obstacle *obst;
extern double distnom();
extern double clearance();
extern int moved();
struct hlist *fhr();
double fastmin_dist();
double getpot1();
struct bumblen *get_successors();

/* soa- Serially optimize collision points on path
*/
int soa(path,npoints,cols)
double **path;
int npoints;
struct colnode *cols;
{
    struct hlist *soareg, *ptr;
    struct node *st, *current;
    int i,j,number, place, allocflag, nextone,
        pathplace;
    double **temppath, **newpath;

    /* Find hard regions in path */
    soareg=fhr(path,npoints,cols);

    /* If no hard regions return, else need to do
some work */
    if (soareg == NULL) {
        return(npoints);
    }

    newpath=(double **)calloc(ndim,sizeof(double
*));
    if (soareg->start > 0) {
        for (i=0; i<ndim; i++)
            newpath[i]=(double *)
                calloc(soareg->start,sizeof(double));
        appendpath(newpath,path,0,0,soareg->start);
        place=soareg->start;
    }
    else
        place=0;

    ptr=soareg; allocflag=0;
    while (ptr != NULL) {
        if (allocflag != 0) {
            free(temppath);
            allocflag=1;
            compress();
            nextone=fbfc(ptr,path,npoints);
            uncompress();
            if (nextone == 0) {
                number=bumblepath(ptr,path,&temppath);
            }
            else {
                number=initorpath(ptr,path,&temppath,&cols);
                number=soa(temppath,number,cols);
                free(cols);
            }
            if (number == 0) {
                printf("Invalid assembly0);
                return(0);
            }
            if (newpath[0] == NULL)
                for (i=0; i<ndim; i++)
                    newpath[i]=(double *)
                        calloc(number,sizeof(double));
            else
                for (i=0; i<ndim; i++)
                    newpath[i]=(double *)
                        realloc(newpath[i],
                            (place+number)*sizeof(double));
            appendpath(newpath,temppath,place,
                0,number);
            place+=number;
            if (ptr->next == NULL) {
                pathplace=ptr->end+1;
                number=npoints-pathplace;
            }
            else {
                pathplace=ptr->end+1;
                number=ptr->next->start-pathplace;
            }
            if (number > 0) {
                for (i=0; i<ndim; i++)
                    newpath[i]=(double *)
                        realloc(newpath[i],
                            (place+number)*sizeof(double));
                appendpath(newpath,path,place,pathplace,
                    number);
                place+=number;
            }
            ptr=ptr->next;
        }
        for (i=0; i<ndim; i++)
            path[i]=newpath[i];
        free(temppath);
        return(place);
    }

    /* Append # pts from path2 to path1 starting w/
place
2th pt on path2 and putting it into place1th
point
in path1 */
    appendpath(path1,path2,place1,place2,number)
    double **path1, **path2;
    int place1,place2,number;
    {
        int i,j;

        for (i=0; i<number; i++) {
            for (j=0; j<ndim; j++)
                path1[j][i+place1]=path2[j][i+place2];
        }
    }
}

```



```

/* fhr- Find hard regions, or collis. regions in
   path
   to be fixed */
struct hlist *fhr(path,npoints,cols)
double **path;
int npoints;
struct colnode *cols;
{
    int i,dim,pot,startregion,lengthregion,
    badregion,lastr;
    double col;
    struct hlist *tempnode, *nextreg, *soareg;
    struct colnode *temp;

    soareg=NULL; lengthregion=0;
    temp=cols->next; nextreg=NULL;
    for (i=1; i < npoints; i++) {
        if (i < npoints-1) {
            col=temp->collision;
            temp=temp->next;
        }
        else
            col=1;

        if (col < 0.000001) {
            if (lengthregion == 0)
                startregion=i;
            lengthregion++;
        }
        else {
            if (lengthregion != 0) {
                badregion=startregion+lengthregion/2;
                tempnode=(struct hlist *)
                calloc(1,sizeof(struct hlist));
            }
            if ((temp != NULL) &&
                (temp->collision > 0.00001))
                tempnode->end=i+1;
            else
                tempnode->end=i;
            if (nextreg == NULL) {
                if (startregion > 1)
                    tempnode->start=startregion-2;
                else
                    tempnode->start=0;
            }
            else {
                if (nextreg->end < startregion-2)
                    tempnode->start=startregion-2;
                else
                    tempnode->start=nextreg->end;
            }
            tempnode->place=badregion;
            tempnode->x=path[0][badregion];
            tempnode->y=path[1][badregion];
            tempnode->z=path[2][badregion];
            tempnode->rotx=path[3][badregion];
            tempnode->roty=path[4][badregion];
            tempnode->rotz=path[5][badregion];
            for (dim=6; dim<ndim; dim++)
                tempnode->t[dim-6]=path[dim][badregion];
            tempnode->next=NULL;
            lastr=lengthregion;
            if (soareg == NULL) {
                soareg=nextreg=tempnode;
            }
            else {
                nextreg->next=tempnode;
                nextreg=tempnode;
            }
            lengthregion=0;
        }
    }
}

```

```

    }
    }
    return(soareg);
}

/* fbfc- Find a more optimal point for the hard
   point
   thru the use of the Cauchy gradient algorithm */
fbfc(hard,path,npoints)
struct hlist *hard;
double **path;
int npoints;
{
    double alpha1, alpha2, alpha3;
    double point[10];
    double pot,pot1,pot2,x,y,z,phi,theta,psi,dtran;
    double *dd,*tempt,lower,upper,dmag;
    int i,j,dim,dim2;

    dd=(double *)calloc(ndim,sizeof(double));
    tempt=(double *)calloc(ndim-6,sizeof(double));
    dtran=0.00001;
    pot2=objectpot(hard->x,hard->y,hard->z
    ,hard->rotx,
    hard->roty,hard->rotz,hard->t,path,npoints);
    while (clearance(hard->x,hard->y,hard->z
    ,hard->rotx,
    hard->roty,hard->rotz,hard->t,hard->x,hard->y,
    hard->z,1) < 0.000001) {
        pot=pot2;
        point[0]=hard->x; point[1]=hard->y;
        point[2]=hard->z; point[3]=hard->rotx;
        point[4]=hard->roty; point[5]=hard->rotz;
        for (dim=6; dim<ndim; dim++)
            point[dim]=hard->t[dim-6];
        for (dim=0; dim<ndim; dim++) {
            point[dim]+=dtran;
            dd[dim]=objectpot(point[0],point[1],point[2]
            ,
            point[3],point[4],point[5],
            &point[6],path,npoints);
            point[dim]-=2*dtran;
            dd[dim]=objectpot(point[0],point[1]
            ,
            point[2],
            point[3],point[4],point[5],
            &point[6],path,npoints);
            point[dim]+=dtran;
        }
        lower=0.0; upper=0.01;
        dmag=sqrt(fsqr(dd[0])+fsqr(dd[1])+fsqr(dd[2])+
            fsqr(dd[3])+fsqr(dd[4])+fsqr(dd[5]));
        alpha1=upper*0.25; alpha2=upper*0.75;
        while ((alpha2-alpha1) > 0.0001) {
            alpha1/=dmag; alpha2/=dmag;
            path[0][npoints]=hard->x-alpha1*dd[0];
            path[1][npoints]=hard->y-alpha1*dd[1];
            path[2][npoints]=hard->z-alpha1*dd[2];
            path[3][npoints]=hard->rotx-alpha1*dd[3];
            path[4][npoints]=hard->roty-alpha1*dd[4];
            path[5][npoints]=hard->rotz-alpha1*dd[5];
            for (dim=6; dim<ndim; dim++)
                path[dim][npoints]=hard->t[dim-6]-
                alpha1*dd[dim];
            pot1=getpot1(npoints,path);
            path[0][npoints]=hard->x-alpha2*dd[0];
            path[1][npoints]=hard->y-alpha2*dd[1];
            path[2][npoints]=hard->z-alpha2*dd[2];
            path[3][npoints]=hard->rotx-alpha2*dd[3];
            path[4][npoints]=hard->roty-alpha2*dd[4];
            path[5][npoints]=hard->rotz-alpha2*dd[5];
            for (dim=6; dim<ndim; dim++)
                path[dim][npoints]=hard->t[dim-6]-

```

```

    alpha2*dd[dim];
    pot2=getpot1(npoints,path);
    if (pot1 < pot2) {
        upper=((upper-lower)/2.0)+lower;
        alpha3=alpha1;
    }
    else
        lower=((upper-lower)/2.0)+lower;
    alpha1=((upper-lower)*0.25)+lower;
    alpha2=((upper-lower)*0.75)+lower;
}

if (pot1 < pot2) {
    path[0][npoints]=hard->x-alpha3*dd[0];
    path[1][npoints]=hard->y-alpha3*dd[1];
    path[2][npoints]=hard->z-alpha3*dd[2];
    path[3][npoints]=hard->rotx-alpha3*dd[3];
    path[4][npoints]=hard->roty-alpha3*dd[4];
    path[5][npoints]=hard->rotz-alpha3*dd[5];
    for (dim=6; dim<ndim; dim++)
        path[dim][npoints]=hard->t[dim-6]-
            alpha3*dd[dim];
    pot2=pot1;
}
if (pot <= pot2) {
    return(0);
}
else {
    for (dim=6; dim<ndim; dim++)
        tempt[dim-6]=path[dim][npoints];
    roundoff(path[0][npoints],path[1][npoints],
        path[2][npoints],&path[3][npoints],
        &path[4][npoints],&path[5][npoints]
        ,&tempt);
    hard->x=path[0][npoints];
    hard->y=path[1][npoints];
    hard->z=path[2][npoints];
    hard->rotx=path[3][npoints];
    hard->roty=path[4][npoints];
    hard->rotz=path[5][npoints];
    for (dim=0; dim<ndim-6; dim++)
        hard->t[dim]=tempt[dim];
}
}
free(dd);
if (ndim > 6)
    free(tempt);
i=hard->start;
if (fabs(hard->x-path[0][i])
    +fabs(hard->y-path[1][i])+
    fabs(hard->z-path[2][i])
    +fabs(hard->rotx-path[3][i])+
    fabs(hard->roty-path[4][i])+
    fabs(hard->rotz-path[5][i]) < 0.01) {
    return(0);
}
i=hard->end;
if (fabs(hard->x-path[0][i])
    +fabs(hard->y-path[1][i])+
    fabs(hard->z-path[2][i])
    +fabs(hard->rotx-path[3][i])+
    fabs(hard->roty-path[4][i])+
    fabs(hard->rotz-path[5][i]) < 0.01) {
    return(0);
}
return(1);
}

roundoff(x,y,z,protx,proty,protz,prott)
double x,y,z,*protx,*proty,*protz,*prott;
{
    double temprx,tempry,temprz;

```

```

    temprx= *protx; tempry= *proty; temprz=
        *protz;
    if (fabs(*protx) < 0.05)
        temprx=0.0;
    if (fabs(*protx-(PI/2.0)) < 0.05)
        temprx= PI/2.0;
    if (fabs(*protx-PI) < 0.05)
        temprx= PI;
    if (fabs(*protx-(3.0*PI/2.0)) < 0.05)
        temprx= (3.0*PI)/2.0;
    if (fabs(*proty) < 0.05)
        tempry=0.0;
    if (fabs(*proty-(PI/2.0)) < 0.05)
        tempry= PI/2.0;
    if (fabs(*proty-PI) < 0.05)
        tempry= PI;
    if (fabs(*proty-(3.0*PI/2.0)) < 0.05)
        tempry= (3.0*PI)/2.0;
    if (fabs(*protz) < 0.05)
        temprz=0.0;
    if (fabs(*protz-(PI/2.0)) < 0.05)
        temprz= PI/2.0;
    if (fabs(*protz-PI) < 0.05)
        temprz= PI;
    if (fabs(*protz-(3.0*PI/2.0)) < 0.05)
        temprz= (3.0*PI)/2.0;
    if (clearance(x,y,z,temprx,tempry,temprz,prott
        x,y,z,1) > 0.000001) {
        *protx=temprx; *proty=tempry; *protz=temprz;
    }
    else if (clearance(x,y,z,temprx,tempry,
        *protz,prott,x,y,z,1) > 0.000001) {
        *protx=temprx; *proty=tempry;
    }
    else if (clearance(x,y,z,temprx,*proty,temprz,
        prott,x,y,z,1) > 0.000001) {
        *protx=temprx; *protz=temprz;
    }
    else if (clearance(x,y,z,*protx,tempry,temprz,
        prott,x,y,z,1) > 0.000001) {
        *protz=temprz; *proty=tempry;
    }
}

/* initorpath- Create init subpath using
replacement
point found by fbfc, similar to method used for
originally creating phase 1 initial path */
initorpath(ptr,path,tempmpath,ncols)
struct hlist *ptr;
double **path,***tempmpath;
struct colnode **ncols;
{
    struct node *st,*current;
    int i,number;

    current=(struct node *)
        calloc(1,sizeof(struct node));
    current->x=path[0][ptr->start];
    current->y=path[1][ptr->start];
    current->z=path[2][ptr->start];
    current->rotx=path[3][ptr->start];
    current->roty=path[4][ptr->start];
    current->rotz=path[5][ptr->start];
    current->md=sqrt(fsqr(current->x-ptr->x)+
        fsqr(current->y-ptr->y)+
        fsqr(current->z-ptr->z));
    for (i=6; i<ndim; i++)
        current->t[i-6]=path[i][ptr->start];
    st=current;
    current=(struct node *)

```

```

    calloc(1, sizeof(struct node));
    current->x=ptr->x; current->y=ptr->y;
    current->z=ptr->z; current->rotx=ptr->rotx;
    current->roty=ptr->roty; current->rotz
    =ptr->rotz;
    current->md
    =sqrt(fsqr(current->x-path[0][ptr->end])+
    fsqr(current->y-path[1][ptr->end])+
    fsqr(current->z-path[2][ptr->end]));
    for (i=6; i<ndim; i++)
        current->t[i-6]=ptr->t[i-6];
    st->next[0]=current;
    current=(struct node *)
    calloc(1, sizeof(struct node));
    current->x=path[0][ptr->end];
    current->y=path[1][ptr->end];
    current->z=path[2][ptr->end];
    current->rotx=path[3][ptr->end];
    current->roty=path[4][ptr->end];
    current->rotz=path[5][ptr->end];
    for (i=6; i<ndim; i++)
        current->t[i-6]=path[i][ptr->end];
    st->next[0]->next[0]=current;
    number=initpath(st, temppath, ncols, 0);
    return(number);
}

/* objectpot- Calculates value of the object
potential. See Chapter 4 of thesis for
definition of object potential */
double objectpot(x,y,z,rotx,roty,rotz,th)
double x,y,z,rotx,roty,rotz,*th;
{
    double nx,ny,nz,value,ox,oy,oz;
    int j,k,l,dof;

    value=0;
    for (j=0; j<objside; j++) {
        dof=movable(j);
        for (k=0; k<object[j].numfacets; k++) {
            if (dof != -1)
                tform(object[j].centers[k][0],
                object[j].centers[k][1],
                object[j].centers[k][2],
                &ox,&oy,&oz,dof,th);
            else {
                ox=object[j].centers[k][0];
                oy=object[j].centers[k][1];
                oz=object[j].centers[k][2];
            }
            tpoint(x,y,z,rotx,roty,rotz,
            ox,oy,oz,&nx,&ny,&nz);
            value+=pointpot(nx,ny,nz)*object[j].area[k];
        }
    }
    return(value);
}

/* getpot1- Calculated potential for point given *
/
double getpot1(npoints,path)
int npoints;
double **path;
{
    double x,y,z,rx,ry,rz,th[4];
    int dim;

    path[0][npoints]=x; y=path[1][npoints];
    z=path[2][npoints]; rx=path[3][npoints];
    ry=path[4][npoints];
    rz=path[5][npoints];
    for (dim=6; dim<ndim; dim++)
        th[dim-6]=path[dim][npoints+1];

    return(objectpot(x,y,z,rx,ry,rz,th));
}

/* Returns approximate min_dist for a point,
fast calculation */
double fastmin_dist(x,y,z)
double x,y,z;
{
    int i,j,k,l,spr,chole,hole;
    double jsum;

    for (i=0; i<nobs; i++) {
        /* calculate potential w.r.t. ith obstacle for
        all
        obsts that aren't holes and are to be
        considered */
        if ((obst[i].shape >= 0) && (obst[i].consider
        == 1)) {
            jsum=distnom(x,y,z,i);
            /* If in obstacle i, consider holes in
            obstacle i */
            if (jsum < 0.000001) {
                /* Is it in hole? */
                for (hole=0; hole < obst[i].nholes; hole++) {
                    choke=obst[i].holes[hole];
                    /* If in hole find dist. to nearest
                    surface */
                    if (distnom(x,y,z,chole) < 0.000001) {
                        return(1.0);
                    }
                }
                return(0.0);
            }
        }
    }
    return(1.0);
}

/* bumblepath- Brute force search invoked if all
else fails */
bumblepath(hard,path,tpath)
struct hlist *hard;
double **path;
double ***tpath;
{
    double mindist,ex,ey,ez,erx,ery,erz,*et;
    int i,j,k,done,npoints;
    struct bumblen *n,*g,*open,*closed,*temp,
    *m,*s;

    et=(double *)calloc((ndim-6),sizeof(double));
    g=(struct bumblen *)calloc(1,sizeof(struct
    bumblen));
    g->x=path[0][hard->start]; g->y
    =path[1][hard->start];
    g->z=path[2][hard->start]; g->rotx
    =path[3][hard->start];
    g->roty=path[4][hard->start]; g->rotz
    =path[5][hard->start];
    g->t=(double *)calloc((ndim-6)
    ,sizeof(double));
    g->father=NULL; g->next=NULL;
    s=g; closed=NULL; open=s;
    for (i=6; i<ndim; i++)
        g->t[i-6]=path[i][hard->start];
    ex=path[0][hard->end]; ey=path[1][hard->end];
    ez=path[2][hard->end]; erx=path[3][hard->end];
    ery=path[4][hard->end]; erz
    =path[5][hard->end];
    for (i=6; i<ndim; i++)

```

```

et[i-6]=path[i][hard->end];
npoints=done=0;
while ((done == 0) && (open != NULL)) {
    n=open; open=open->next; n->next=closed;
    closed=n;
    mindist=clearance(n->x,n->y,n->z,n->rotx,
    ,n->roty,
    n->rotz,n->t,n->x,n->y,n->z,0);
    if (fsqr(n->x-ex)+fsqr(n->y-ey)+fsqr(n->z-ez)
    < fsqr(mindist)) {
        temp=(struct bumblen *)
        calloc(1,sizeof(struct bumblen));
        temp->x=ex; temp->y=ey; temp->z=ez;
        temp->rotx=erx; temp->roty=ery;
        temp->rotz=erz;
        for (i=6; i<ndim; i++)
            temp->t[i]=et[i];
        done=1;
        n->child[0]=temp;
        temp->father=n;
        n=temp;
    }
    else {
        if (mindist > 0.00001) {
            if (mindist < 0.01)
                mindist=0.01;
            m=get_successors(n,mindist);
            for (i=0; i<12; i++) {
                temp=m->next; m->father=n;
                n->child[i]=m;
                m->h=fsqr(ex-m->x)+fsqr(ey-m->y)+
                fsqr(ez-m->z)+ fsqr(erx-m->rotx)+
                fsqr(ery-m->roty)+fsqr(erz-m->rotz);
                add_to_bumble_open(&open,m);
                m=temp;
            }
        }
    }
    if (done == 1) {
        while (temp != NULL) {
            npoints++;
            temp=temp->father;
        }
        (*tpath)=(double **)calloc(ndim,sizeof(double
        *));
        for (j=0; j<ndim; j++)
            (*tpath)[j]=(double *)
            calloc(npoints,sizeof(double));
        for (j=npoints-1; j >= 0; j--) {
            (*tpath)[0][j]=n->x; (*tpath)[1][j]=n->y;
            (*tpath)[2][j]=n->z; (*tpath)[3][j]=n->rotx;
            (*tpath)[4][j]=n->roty; (*tpath)[5][j]
            =n->rotz;
            for (k=0; k<ndim-6; k++)
                (*tpath)[k+6][j]=n->t[k];
            n=n->father;
        }
        while (open != NULL) {
            n=open;
            open=open->next;
            cfree(n);
        }
        while (closed != NULL) {
            n=closed;
            closed=closed->next;
            cfree(n);
        }
        return(npoints);
    }
}
add_to_bumble_open(open,m)

```

```

struct bumblen **open,*m;
{
    struct bumblen *temp;

    if (((*open) == NULL) || (m->h < (*open)->h)){
        temp=(*open);
        (*open)=m;
        (*open)->next=temp;
        return(0);
    }

    temp=(*open);
    while ((temp->next != NULL) &&
    (temp->next->h < m->h))
        temp=temp->next;
    m->next=temp->next; temp->next=m;
}

struct bumblen *get_successors(n,distance)
struct bumblen *n;
double distance;
{
    struct bumblen *list,*temp;
    int i;
    double t;

    t=atan2(distance,maxmo);

    list=temp=(struct bumblen *)
    calloc(1,sizeof(struct bumblen));
    list->x=n->x+distance; list->y=n->y;
    list->z=n->z; list->rotx=n->rotx;
    list->roty=n->roty; list->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        list->t[i]=n->t[i];

    temp->next=(struct bumblen *)
    calloc(1,sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x-distance; temp->y=n->y;
    temp->z=n->z; temp->rotx=n->rotx;
    temp->roty=n->roty; temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
    calloc(1,sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y+distance;
    temp->z=n->z; temp->rotx=n->rotx;
    temp->roty=n->roty; temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
    calloc(1,sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y-distance;
    temp->z=n->z; temp->rotx=n->rotx;
    temp->roty=n->roty; temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
    calloc(1,sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y;
    temp->z=n->z+distance; temp->rotx=n->rotx;
    temp->roty=n->roty; temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)

```

```

    calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y;
    temp->z=n->z-distance; temp->rotx=n->rotx;
    temp->roty=n->roty; temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx+t; temp->roty=n->roty;
    temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx-t; temp->roty=n->roty;
    temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx; temp->roty=n->roty+t;
    temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx; temp->roty=n->roty-t;
    temp->rotz=n->rotz;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx; temp->roty=n->roty;
    temp->rotz=n->rotz+t;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    temp->next=(struct bumblen *)
        calloc(1, sizeof(struct bumblen));
    temp=temp->next;
    temp->x=n->x; temp->y=n->y; temp->z=n->z;
    temp->rotx=n->rotx; temp->roty=n->roty;
    temp->rotz=n->rotz-t;
    for (i=0; i<ndim-6; i++)
        temp->t[i]=n->t[i];

    return(list);
}

/* This file connects pts on straight line path
   segments together, effectively reducing number
   of path segments in path */
#include <stdio.h>
#include <math.h>
#include "common.h"

```

```
extern double fsqr();
```

```

extern int ndim;

#define MAXLINE 1.5
/* Max length of joined segment */

/* slp- Generate collision-free straight line
   path using computed path as a guide. */
int slp(path, npoints)
double **path;
int npoints;
{
    int i, j, k;
    double distance, temp;

    for (i=0; i< npoints-2; i++) {
        distance=sqrt(
            fsqr(path[0][i+2]-path[0][i])+
            fsqr(path[1][i+2]-path[1][i])+
            fsqr(path[2][i+2]-path[2][i]));
        if (distance < MAXLINE) {
            if (online(path, i)) {
                for (j=i+1; j<npoints-1; j++)
                    for (k=0; k<ndim; k++)
                        path[k][j]=path[k][j+1];
                npoints--;
                i--;
            }
        }
    }
    return(npoints);
}

online(path, i)
double **path;
int i;
{
    double *flag, bottom, mean;
    int dim, active;

    flag=(double *)calloc(ndim, sizeof(double));
    active=0;
    for (dim=0; dim<ndim; dim++) {
        bottom=path[dim][i+2]-path[dim][i];
        if (bottom != 0) {
            flag[active]=(path[dim][i+1]-path[dim][i])
                /bottom;
            active++;
        }
    }
    mean=0;
    for (dim=0; dim<active; dim++)
        mean+=flag[dim];
    mean/=active;
    for (dim=0; dim<active; dim++)
        if ((flag[dim]-mean > 0.05) ||
            (mean-flag[dim] > 0.05)) {
            free(flag);
            return(0);
        }
    free(flag);
    return(1);
}

/* This file contains routines common to Phase 1
   and Phase 2
   Stage 1 */
#include <stdio.h>
#include <math.h>
#include "common.h"

#define OBJECTINC 0.05

extern struct obstacle *obst; /* number of
   obstacles */

```

```

extern struct side *bound; /* all sides of all
obstacles */
extern struct vertex *vertice; /* vertices of
obstacles */
extern struct line *edge; /* edges of obstacles */
extern struct oside *object;
struct transform *joint;
int numjoints, rigid;
extern int nob, nbounds, nedge, nvert, npoints,
obside;
double fsqr();

```

```

double fsqr(x)
double x;
{
    return(x*x);
}

```

```

/* Reads obst desc in from a file called
obstacles. */
/* Twincon converts twin file into this format */
initobstacles()
{

```

```

    int i,j,k,ptr,total,lastsolid,holes;
    FILE *fp;
    float tx,ty,tz,td;

    if ((fp=fopen("obstacles","r")) == NULL)
        perror("obstacles");
    fscanf(fp,"%d",&nobs);
    obst=(struct obstacle *)
        calloc(nobs,sizeof(struct obstacle));
    for (i=0; i<nobs; i++) {
        fscanf(fp,"%d",&obst[i].nsides);
        obst[i].side=(int *)calloc(obst[i].nsides
            ,sizeof(int));
        for (j=0; j<obst[i].nsides; j++)
            fscanf(fp,"%d",&obst[i].side[j]);
        fscanf(fp,"%d",&obst[i].shape);
        if (obst[i].shape >= 0) {
            lastsolid=i;
            holes=0;
        }
        else {
            if (holes == 0)
                obst[lastsolid].holes=(int *)calloc(1
                    ,sizeof(int));
            else
                obst[lastsolid].holes=(int *)
                    realloc(obst[lastsolid].holes,
                        (holes+1)*sizeof(int));
            obst[lastsolid].holes[holes]=i; holes++;
            obst[lastsolid].nholes=holes;
        }
        fscanf(fp,"%d",&obst[i].id);
    }
}

```

```

fscanf(fp,"144d0",&nvert);
vertice=(struct vertex *)
    calloc(nvert,sizeof(struct vertex));
for (i=0; i<nvert; i++) {
    fscanf(fp,"%f %f %f %f",&tx,&ty,&tz);
    vertice[i].x=tx; vertice[i].y=ty; vertice[i].z
        =tz;
    fscanf(fp,"%d",&vertice[i].nedges);
    vertice[i].edge=(int *)
        calloc(vertice[i].nedges,sizeof(int));
    for (j=0; j<vertice[i].nedges; j++)
        fscanf(fp,"%d",&vertice[i].edge[j]);
}

```

```

fscanf(fp,"144d0",&nedge);
edge=(struct line *)

```

```

    calloc(nedge,sizeof(struct line));
for (i=0; i<nedge; i++) {
    fscanf(fp,"%f %f %f %d %d0,
        &tx,&ty,&tz,&edge[i].vert1,&edge[i].vert2);
    edge[i].i=tx; edge[i].j=ty; edge[i].k=tz;
    fscanf(fp,"%d",&edge[i].nfacs);
    edge[i].face=(int *)
        calloc(edge[i].nfacs,sizeof(int));
    for (j=0; j<edge[i].nfacs; j++)
        fscanf(fp,"%d",&edge[i].face[j]);
}

```

```

fscanf(fp,"%d",&nbounds);
bound=(struct side *)
    calloc(nbounds,sizeof(struct side));
for (i=0; i<nbounds; i++) {
    fscanf(fp,"%f %f %f %f0,&tx,&ty,&tz,&td);
    bound[i].a=tx; bound[i].b=ty; bound[i].c=tz;
    bound[i].d=td;
    fscanf(fp,"%d %d %d %d",&bound[i].obstacle,
        &bound[i].numloops,&(bound[i].type),
        &(bound[i].real));
    ptr=0;
    bound[i].nedge=(int *)
        calloc(bound[i].numloops,sizeof(int));
    total=0;
    for (j=0; j<bound[i].numloops; j++) {
        fscanf(fp,"%d",&bound[i].nedge[j]);
        total+=bound[i].nedge[j];
    }
    bound[i].edge=(int *)calloc(total,sizeof(int));
    for (j=0; j<bound[i].numloops; j++) {
        for (k=ptr; k<bound[i].nedge[j]+ptr; k++)
            fscanf(fp,"%d",&bound[i].edge[k]);
        ptr=k;
    }
    fscanf(fp,"%d %f0,&bound[i].compflag,&tx);
    bound[i].compress=tx;
}
fclose(fp);

```

```

/* Assume obj is centered at (x,y,z,rotx,roty,
rotz),
compute pose of the obj point ox,oy,oz in the
world xnew,ynew,znew */
tpoint(x,y,z,rotx,roty,rotz,ox,oy,oz,xnew,ynew,
znew)
double x, y,z,rotx,roty,rotz,ox,oy,oz,*xnew,*ynew,
*znew;
{
    double cx,sx,cy,sy,cz,sz;
    cx=cos(rotx); sx=sin(rotx); cy=cos(roty);
    sy=sin(roty); cz=cos(rotz); sz=sin(rotz);
    *xnew=(cz*cy)*ox+(cz*sy*sx-sz*cx)*oy
        +(cz*sy*cx+sz*sx)*oz+x;
    *ynew=(sz*cy)*ox+(sz*sy*sx+cz*cx)*oy
        +(sz*sy*cx-cz*sx)*oz+y;
    *znew=(-1*sy)*ox+(cy*sx)*oy+(cy*cx)*oz+z;
}

```

```

movable(j)
int j;
{
    int i;

    if ((rigid == 0) || (j < joint[0].start))
        return(-1);
    for (i=0; i<numjoints; i++)
        if ((j >= joint[i].start) && (j <
            joint[i].end))
            return(i);
}

```

```
/* Read in vertices of faces of object */
initobject()
```

```
{
    int i,j,k,ptr,total;
    double cx,cy,cz,sx,sy,sz;
    FILE *fp;
    float tx,ty,tz,tu,tv;

    if ((fp=fopen("object","r")) == NULL)
        perror("object");
    fscanf(fp,"%d",&rigid);
    fscanf(fp,"%d",&objside);
    object=(struct oside *)
        calloc(objside,sizeof(struct oside));
    for (i=0; i<objside; i++) {
        fscanf(fp,"%d",&(object[i].numsamp));
        object[i].samples=(double **)
            calloc(object[i].numsamp,sizeof(double *));
        for (k=0; k<object[i].numsamp; k++) {
            object[i].samples[k]=(double *)
                calloc(5,sizeof(double));
            fscanf(fp,"%f %f %f %f %f",&tx,&ty,&tz,&tu
                ,&tv);
            object[i].samples[k][0]=tx;
            object[i].samples[k][1]=ty;
            object[i].samples[k][2]=tz;
            object[i].samples[k][3]=tu;
            object[i].samples[k][4]=tv;
        }
        fscanf(fp,"%d",&(object[i].numfacets));
        object[i].area=(double *)
            calloc(object[i].numfacets,sizeof(double));
        object[i].centers=(double **)
            calloc(object[i].numfacets,sizeof(double *));
        for (k=0; k<object[i].numfacets; k++) {
            object[i].centers[k]=(double *)
                calloc(3,sizeof(double));
            fscanf(fp,"%f %f %f",&tx,&ty,&tz,&tu);
            object[i].centers[k][0]=tx;
            object[i].centers[k][1]=ty;
            object[i].centers[k][2]=tz;
            object[i].area[k]=tu;
        }
        for (k=0; k<3; k++) {
            fscanf(fp,"%f %f %f",&tx,&ty,&tz);
            object[i].r[k][0]=tx;
            object[i].r[k][1]=ty;
            object[i].r[k][2]=tz;
        }

        ptr=0;
        fscanf(fp,"%d",&(object[i].numloops));
        object[i].nvert=(int *)
            calloc(object[i].numloops
                ,sizeof(int));
        total=0;
        for (k=0; k<object[i].numloops; k++) {
            fscanf(fp,"%d",&(object[i].nvert[k]));
            total+=object[i].nvert[k];
        }
        object[i].x=(double *)calloc(total
            ,sizeof(double));
        object[i].y=(double *)calloc(total
            ,sizeof(double));
        object[i].z=(double *)calloc(total
            ,sizeof(double));
        for (k=0; k<object[i].numloops; k++) {
            for (j=ptr; j<ptr+object[i].nvert[k]; j++) {
                fscanf(fp,"%f %f %f %f %f",&tx,&ty,&tz);
                object[i].x[j]=tx;
                object[i].y[j]=ty;
                object[i].z[j]=tz;
            }
        }
    }
}
```

```

    }
    ptr=j;
}
cx=object[i].x[1]-object[i].x[0];
cy=object[i].y[1]-object[i].y[0];
cz=object[i].z[1]-object[i].z[0];
sx=object[i].x[2]-object[i].x[1];
sy=object[i].y[2]-object[i].y[1];
sz=object[i].z[2]-object[i].z[1];
object[i].nx=cy*sz-sy*cz;
object[i].ny=cx*sz-sz*cx;
object[i].nz=cx*sy-sx*cy;
}
if (rigid == 1) {
    fscanf(fp,"%d",&numjoints);
    joint=(struct transform *)
        calloc(numjoints,sizeof(struct
            transform));
    for (i=0; i<numjoints; i++) {
        fscanf(fp,"%f %f %f %f %f %f"
            ,&joint[i].first[0],
            &joint[i].first[1],
            &joint[i].first[2],
            &joint[i].first[3],
            &joint[i].first[4],
            &joint[i].first[5]);
        fscanf(fp,"%f %f %f %f %f %f"
            ,&joint[i].third[0],
            &joint[i].third[1],
            &joint[i].third[2],
            &joint[i].third[3],
            &joint[i].third[4],
            &joint[i].third[5]);
        fscanf(fp,"%d %d",&joint[i].start
            ,&joint[i].end);
        fscanf(fp,"%1s",&joint[i].type[0]);
        cx=cos(joint[i].first[3]); sx
            =sin(joint[i].first[3]);
        cy=cos(joint[i].first[4]); sy
            =sin(joint[i].first[4]);
        cz=cos(joint[i].first[5]); sz
            =sin(joint[i].first[5]);
        joint[i].firstm[0][0]=cz*cy;
        joint[i].firstm[0][1]=sz*cy;
        joint[i].firstm[0][2]=-sy;
        joint[i].firstm[1][0]=cz*sy*sx-sz*cx;
        joint[i].firstm[1][1]=sz*sy*sx+cz*cx;
        joint[i].firstm[1][2]=cy*sx;
        joint[i].firstm[2][0]=cz*sy*cx+sz*sx;
        joint[i].firstm[2][1]=sz*sy*cx-cz*sx;
        joint[i].firstm[2][2]=cy*cx;
        joint[i].firstm[3][0]=joint[i].first[0];
        joint[i].firstm[3][1]=joint[i].first[1];
        joint[i].firstm[3][2]=joint[i].first[2];
        cx=cos(joint[i].third[3]); sx
            =sin(joint[i].third[3]);
        cy=cos(joint[i].third[4]); sy
            =sin(joint[i].third[4]);
        cz=cos(joint[i].third[5]); sz
            =sin(joint[i].third[5]);
        joint[i].thirdm[0][0]=cz*cy;
        joint[i].thirdm[0][1]=sz*cy;
        joint[i].thirdm[0][2]=-sy;
        joint[i].thirdm[1][0]=cz*sy*sx-sz*cx;
        joint[i].thirdm[1][1]=sz*sy*sx+cz*cx;
        joint[i].thirdm[1][2]=cy*sx;
        joint[i].thirdm[2][0]=cz*sy*cx+sz*sx;
        joint[i].thirdm[2][1]=sz*sy*cx-cz*sx;
        joint[i].thirdm[2][2]=cy*cx;
        joint[i].thirdm[3][0]=joint[i].third[0];
        joint[i].thirdm[3][1]=joint[i].third[1];
        joint[i].thirdm[3][2]=joint[i].third[2];
        joint[i].position=0.0;
    }
}
```

```

    gettransform(i);
}
fclose(fp);
}

tform(x,y,z,outx,outy,outz,dof,th)
double x,y,z,*outx,*outy,*outz,*th;
int dof;
{
    double ct,st; int i;

    for (i=dof; i>=0; i--) {
        if (th[i] != joint[i].position) {
            joint[i].position=th[i];
            gettransform(i);
        }
        *outx=joint[i].xform[0][0]*x+
        joint[i].xform[1][0]*y+joint[i].xform[2][0]*z
        +joint[i].xform[3][0];
        *outy=joint[i].xform[0][1]*x+
        joint[i].xform[1][1]*y+joint[i].xform[2][1]*z
        +joint[i].xform[3][1];
        *outz=joint[i].xform[0][2]*x+
        joint[i].xform[1][2]*y+joint[i].xform[2][2]*z
        +joint[i].xform[3][2];
        x= *outx; y= *outy; z= *outz;
    }
}

gettransform(i)
int i;
{
    double tform[4][3],cz,sz;
    int j,k,l;

    if (joint[i].type[0] == 'p') {
        for (j=0; j<4; j++)
            for (k=0; k<3; k++)
                tform[j][k]=joint[i].thrdm[j][k];
        tform[3][2]=joint[i].position;
    }
    else {
        cz=cos(joint[i].position);
        sz=sin(joint[i].position);
        for (j=0; j<4; j++) {
            tform[j][2]=joint[i].thrdm[j][2];
            tform[j][0]=cz*joint[i].thrdm[j][0]
            -sz*joint[i].thrdm[j][1];
            tform[j][1]=sz*joint[i].thrdm[j][0]
            +cz*joint[i].thrdm[j][1];
        }
        for (j=0; j<3; j++)
            for (k=0; k<3; k++) {
                joint[i].xform[j][k]=0;
                for (l=0; l<3; l++)
                    joint[i].xform[j][k]+
                    =joint[i].firstm[l][k]
                    *tform[j][l];
            }
        for (j=0; j<3; j++) {
            joint[i].xform[3][j]=joint[i].firstm[3][j];
            for (k=0; k<3; k++)
                joint[i].xform[3][j]+=joint[i].firstm[k][j]*
                tform[3][k];
        }
    }
}

compress()
{
    int i,j,k,e,epr,v;

```

```

    for (i=0; i<nbounds; i++) {
        if (bound[i].compflag) {
            epr=0;
            for (j=0; j<bound[i].numloops; j++) {
                for (k=epr; k<bound[i].nedge[j]+epr; k+
                +) {
                    e=bound[i].edge[k];
                    if (e > 0)
                        e--;
                    else
                        e=(e+1)* -1;
                    v=edge[e].vert1;
                    vertice[v].x-=bound[i].a
                    *bound[i].compress;
                    vertice[v].y-=bound[i].b
                    *bound[i].compress;
                    vertice[v].z-=bound[i].c
                    *bound[i].compress;
                }
                epr+=k;
            }
            bound[i].d+=bound[i].compress;
        }
    }
}

uncompress()
{
    int i,j,k,l,epr,e,v;

    for (i=0; i<nbounds; i++) {
        if (bound[i].compflag) {
            epr=0;
            for (j=0; j<bound[i].numloops; j++) {
                for (k=epr; k<bound[i].nedge[j]+epr; k++) {
                    e=bound[i].edge[k];
                    if (e > 0)
                        e--;
                    else
                        e=(e+1)* -1;
                    v=edge[e].vert1;
                    vertice[v].x+=bound[i].a*bound[i].compress;
                    vertice[v].y+=bound[i].b*bound[i].compress;
                    vertice[v].z+=bound[i].c*bound[i].compress;
                }
                epr+=k;
            }
            bound[i].d-=bound[i].compress;
        }
    }
}

/* moved- Returns 1 if surf sptr will move when pt
is moved from x,y,z to lx,ly,lz. If obst is
compressed and move is from lx,ly,lz to x,y,z
is along surface, it will return 1 as well.
See August 2nd notes for description of this
algorithm */
moved(sptr,x,y,z,lx,ly,lz)
int sptr;
double x,y,z,lx,ly,lz;
{
    int i,s,k,inside,compress;
    double mx,my,mz,a,b,c,d,tsum,t;

    tsum=bound[sptr].a*x+bound[sptr].b*y+
    bound[sptr].c*z+bound[sptr].d;
    if (tsum > 0.000001)
        return(0);

    i=bound[sptr].obstacle;

```



```

mx=x-lx; my=y-ly; mz=z-lz; compress=1;
/* If last point is in obst, obst is/will
compress */
for (k=0; k<obst[i].nsides; k++) {
s=obst[i].side[k];
if (k != s)
if (bound[s].a*lx+bound[s].b*ly+bound[s].c
*lz+
bound[s].d > 0.000001)
compress=0;
}
if (compress == 1)
return(1);

a=bound[sptr].a; b=bound[sptr].b; c
=bound[sptr].c;
d=bound[sptr].d;
/* Is (lx,ly,lz) above surface */
if (a*lx+b*ly+c*lz+d > 0.000001) {
t=tsum/(a*mx+b*my+c*mz);
/* Use lx,ly,lz to store xstar,ystar,zstar */
lx=x-t*mx; ly=y-t*my; lz=z-t*mz; inside=1;
/* Find out if lx,ly,lz is inside side */
for (k=0; k<obst[i].nsides; k++) {
s=obst[i].side[k];
if ((s != sptr) &&
(bound[s].a*lx+bound[s].b*ly+bound[s].c*lz
+
bound[s].d
> 0.00001))
inside=0;
}
/* If inside, obstacle will compress */
if (inside == 1)
return(1);
}
return(0);
}

```

/* tessle-sphere.c- Some utility functions used
for a spherical data data structures. Stolen
wholesale from Chien-Huei Chen. */

```

#include <stdio.h>
#include <math.h>
#include "common.h"
double mod2pi();
double Theta[SNLEVEL][SNROW][SNCOL];
double Phi[SNLEVEL][SNROW][SNCOL];

```

```

read_icos(filename)
char *filename;
{
int fi;
if((fi=open(filename,0)) == -1) {
fprintf(stderr,"can't open %s\n",filename);
exit(1);
}
read(fi,Phi,NTOTAL*sizeof(double));
read(fi,Theta,NTOTAL*sizeof(double));
close(fi);
}

```

/* assign-tessle- Given spherical coordinate rpt,
find
nearest tessle ilrc to direction indicated by
rpt. */

```

assign_tessle(rpt,ilrc)
struct Scoord rpt;
struct lcoord *ilrc;
{

```

```

struct lcoord jlrc;
double phi,theta,aphi,atheta,phitmp,offset_phi;
double dottmp,rc2,max_dot,fr,fc,ratio;
int il, ir, ic;

```

```

theta = rpt.t; phi = rpt.p;
if(theta == 0.0)
/* zenith of icosphere */
{ ilrc->l = 5; ilrc->r = ilrc->c = 0; return;}
/* nadir of icosphere */
else if(theta >= PI)
{ ilrc->l = -1; ilrc->r = ilrc->c = 0;
return;}
/* belong to one of top triangles */
else if(theta <= ATAN2) {
il = phi / (72 * DEG);
offset_phi = phi - il * 72*DEG;
rc2 = FQ * theta / ATAN2;
fc = rc2 * offset_phi / (72*DEG);
fr = rc2 - fc - 1.0;
}
/* belong to bottom triangles */
else if(theta >= (180*DEG - ATAN2)) {
phitmp = mod2pi(phi - 36*DEG);
il = phitmp / (72*DEG);
offset_phi = phitmp - il * 72*DEG;
rc2 = FQ * (PI - theta) / ATAN2;
fr = FQ - rc2 * offset_phi / (72*DEG) - 1.0;
fc = 3*FQ - rc2 - fr - 1.0;
}
else {
ratio = (theta - ATAN2) / (PI - 2*ATAN2);
phitmp = mod2pi(phi - ratio * (36*DEG));
il = phitmp / (72*DEG);
offset_phi = phitmp - il * 72*DEG;
rc2 = ratio * FQ;
fr = FQ - FQ * offset_phi / (72*DEG) - 1.0;
fc = rc2 - fr + FQ - 1;
}
ic = fc; /* floor */
ir = 0.999 + fr; /* ceiling */
if(il < 0 || il >= SNLEVEL || ir < 0 ||
ir >= SNROW || ic < 0 || ic >= SNCOL) {
printf("Warning: lrc (%d %d %d) out of .sp -1

```

```

range at phi: %f; theta:%f\n",
il, ir, ic, phi, theta);
ilrc->l = mod(SNLEVEL,il);
ilrc->r = mod(SNROW,ir);
ilrc->c = mod(SNCOL,ic);
}
else {
ilrc->l = il; ilrc->r = ir; ilrc->c = ic;
}
get_tp(*ilrc,&atheta,&aphi);
max_dot = sin(atheta) * sin(theta) * cos(aphi
- phi) +
cos(atheta) * cos(theta);
while(get_closer(rpt,&max_dot,ilrc));
}

```

/* get-closer- given a spherical coord.rpt, and a
icoxel w index *ilrc on the ico., find among its

```

neighbors the one having largest dot product
with
rpt. ilrc is reassign to the index of the best
neighbor. Returns dot product */
get_closer(rpt,maxdot,ilrc)
struct Scoord rpt;
struct lcoord *ilrc;
double *maxdot;

```

```

{
    int flag, i, nn;
    double theta, phi, newdot;
    struct Icoord nlrc[6];

    flag = 0;
    nn=find_neighbors(*ilrc,nlrc);
    FOR(i,nn) {
        get_tp(nlrc[i],&theta,&phi);
        newdot = sin(theta) * sin(rpt.t) *
            cos(phi - rpt.p) +
            cos(theta) * cos(rpt.t);
        if(newdot > *maxdot) {
            flag = 1;
            ilrc->l = nlrc[i].l; ilrc->r = nlrc[i].r;
            ilrc->c = nlrc[i].c; *maxdot = newdot;
        }
    }
    return (flag);
}

```

/* get_tp- Look up theta and phi angles
corresponding
to tessle described by ilrc. */

```

get_tp(ilrc,theta,phi)
double *theta, *phi;
struct Icoord ilrc;
{
    if(ilrc.l >= 0 && ilrc.l < SNLEVEL) {
        *theta = Theta[ilrc.l][ilrc.r][ilrc.c];
        *phi = Phi[ilrc.l][ilrc.r][ilrc.c];
    }
    else {
        *phi = 0.0;
        if(ilrc.l < 0) *theta = PI;
        if(ilrc.l >= SNLEVEL) *theta = 0.0;
    }
    if (*theta < -3.14)
        *theta += TWOPI;
    if (*phi < -3.14)
        *phi += TWOPI;
}

```

/* get_thp- Look up theta and phi angles
corresponding
to tessle described by ilrc. */

```

get_thp(l,r,c,theta,phi)
double *theta, *phi;
int l,r,c;
{
    if(l >= 0 && l < SNLEVEL) {
        *theta = Theta[l][r][c];
        *phi = Phi[l][r][c];
    }
    else {
        *phi = 0.0;
        if(l < 0) *theta = PI;
        if(l >= SNLEVEL) *theta = 0.0;
    }
}

```

/* imed_neighbor- Returns 1 if ilrc is an
immediate
neighbor of jlrc on sphere, 0 otherwise */

```

imed_neighbor(ilrc,jlrc)
struct Icoord ilrc, jlrc;
{
    int in, il, ir, ic, nn;
    struct Icoord nlrc[6];

    il = ilrc.l; ir = ilrc.r; ic = ilrc.c;
    if(il == jlrc.l && ir == jlrc.r && ic ==
        jlrc.c)

```

```

    return (1);
    nn=find_neighbors(jlrc,nlrc);
    FOR(in,nn)
        if(il == nlrc[in].l && ir == nlrc[in].r &&
            ic == nlrc[in].c)
            return (1);
    return (0);
}

```

/* find_neighbors- Find the neighbors of the
tessle

indicated by ilrc on the sphere. */

```

find_neighbors(ilrc,nlrc)
struct Icoord ilrc, nlrc[];
{
    int nn; /* number of neighbors */
    register i,il,ir,ic,nr[6],nl[6],nc[6];
    nn = 6;
    il = ilrc.l; ir = ilrc.r; ic = ilrc.c;
    FOR(i,6) nl[i] = il;
    nr[0] = ir - 1; nc[0] = ic;
    nr[1] = ir - 1; nc[1] = ic + 1;
    nr[2] = ir; nc[2] = ic + 1;
    nr[3] = ir + 1; nc[3] = ic;
    nr[4] = ir + 1; nc[4] = ic - 1;
    nr[5] = ir; nc[5] = ic - 1;

```

if(il >= SNLEVEL) { /* zenith of icosphere */

```

    nn = 5;
    FOR(i,5) {
        nl[i] = i;
        nr[i] = nc[i] = 0;
    }
}

```

else if(il <= -1) { /* zadir of icosphere */

```

    nn = 5;
    FOR(i,5) {
        nl[i] = i;
        nr[i] = SNROW - 1;
        nc[i] = SNCOL - 1;
    }
}

```

else if(ir == 0 && ic == 0) {

```

    nl[0] = 5; nr[0] = 0; nc[0] = 0;
    nl[1] = mod(SNLEVEL,il + 1); nr[1] = 0; nc[1]
    = 0;
    nl[4] = mod(SNLEVEL,il - 1); nr[4] = 0; nc[4]
    = 1;
    nl[5] = mod(SNLEVEL,il - 1); nr[5] = 0; nc[5]
    = 0;
}

```

else if(ir == (SNROW-1) && ic == (SNCOL-1)) {

```

    nl[1] = mod(SNLEVEL,il+1); nr[1] = SNROW-1;
    nc[1] = SNCOL-1; nl[2] = -1; nr[2] = 0; nc[2] = 0;
    nl[3] = mod(SNLEVEL,il-1); nr[3] = SNROW-1;
    nc[3] = SNCOL-1; nl[4] = mod(SNLEVEL,il-1);
    nr[4] = SNROW-2; nc[4] = SNCOL-1;
}

```

else if(ir == 0)

```

    if(ic < FQ) {
        nl[0] = mod(SNLEVEL,il+1); nr[0] = ic-1; nc[0]
        = 0;
        nl[1] = mod(SNLEVEL,il+1); nr[1] = ic; nc[1]
        = 0;
    }
}

```

else {

```

    nl[0] = mod(SNLEVEL,il+1); nr[0] = SNROW-1;
    nc[0] = ic-FQ; nl[1] = mod(SNLEVEL,il+1);
    nr[1] = SNROW-1; nc[1] = ic-FQ+1;
    if(ic == (SNCOL-1)) {
        nl[2] = mod(SNLEVEL,il+1); nr[2]
        = SNROW-1;
        nc[2] = ic-FQ+2;
    }
}

```

```

    }
    else if(ic == (SNCOL-1)) {
        nl[1]=mod(SNLEVEL,il+1); nr[1]=SNROW-1;
        nc[1]=FQ+ir; nl[2]=mod(SNLEVEL,il+1);
        nr[2]=SNROW-1; nc[2]=FQ+ir+1;
    }
    else if(ic == 0) {
        nl[4]=mod(SNLEVEL,il-1); nr[4]=0; nc[4]=ir+1;
        nl[5]=mod(SNLEVEL,il-1); nr[5]=0; nc[5]=ir;
        if(ir == (SNROW-1)) {
            nn=5;
            nl[3]=mod(SNLEVEL,il-1); nr[3]=0; nc[3]=FQ;
            nl[4]=mod(SNLEVEL,il-1); nr[3]=0; nc[3]
            =FQ-1;
        }
    }
    else if(ir == (SNROW-1))
        if(ic > FQ) {
            nl[3]=mod(SNLEVEL,il-1); nr[3]=ic-FQ;
            nc[3]=SNCOL-1; nl[4]=mod(SNLEVEL,il-1);
            nr[4]=ic-FQ-1; nc[4]=SNCOL-1;
        }
    else if(ic == FQ) {
        nn=5;
        nl[3]=mod(SNLEVEL,il-1); nr[3]=0;
        nc[3]=SNCOL-1; nl[4]=il; nr[4]=ir; nc[4]
        =FQ-1;
    }
    else {
        nl[3]=mod(SNLEVEL,il-1); nr[3]=0;
        nc[3]=FQ+ic; nl[4]=mod(SNLEVEL,il-1);
        nr[4]=0; nc[4]=FQ+ic-1;
    }
    FOR(i,6) {
        nlrc[i].l=nl[i]; nlrc[i].r=nr[i]; nlrc[i].c
        =nc[i];
    }
    return(nn);
}

/* mod2pi- Returns angle between 0 and 2pi */
double mod2pi(x)
double x;
{
    while(x < 0.0) x += 2 * PI;
    while(x >= (2*PI)) x -= 2 * PI;
    return (x);
}

xyz_to_rpt(xyz,rpt)
double xyz[3];
struct Scoord *rpt;
{
    rpt->r=sqrt(sdot(xyz));
    rpt->p=(xyz[0] == 0.0 && xyz[1] == 0.0)
    ? 0.0
    : mod2pi(atan2(xyz[1],xyz[0]));
    rpt->t=acos(xyz[2] / rpt->r);
}

```

UNCERTANAL

```

/* Source code roughly corresponding to Phase 2,
Stage 1 */
/* Program UNCERTANAL */
/* Calling format- uncertanal dimension */
/* Input:

```

```

object- File describing the object, created
by the program twincon in this instance
which takes twin boundary reps and
converts them into a convenient
representation
for our purposes. The idea is that if a
different modeler were used a different
conversion program would be written to
convert
the breps into my standard format
obstacles- File describing the obstacles, also
created by twincon. The function build-input
in the assembly supervisor takes care of
of invoking twincon
uncobject- Same as object file, only with
object in
its most-material-condition.
uncobstacles- Same as obstacles file, only with
obstacles
in their most-material-condition.
soa.out- Output of path planner (reversed by
supervisor).
uncertainties- File containing center of mass
and positional
uncertainty of each obstacle, initial pose
uncertainty
and control uncertainty of grasped part. */
/* Output:
strategy- Characterization of path segments in
path upto first
collision point, or end of path, which ever
occurs
first. */
/* Description: This program takes the path from
Phase 1,
takes the object and obstacles used their and
incorporates
the 3 types of uncertainty into them, and then
characterizes
each segment of the path by simulating the
motion of the
grasped part along the path. This is further
described in
Section 1 of Chapter 6 in my thesis. */

```

```

#include <stdio.h>
#include <math.h>
#include "/home/rv14/q/research/pathplan
/common.h"
#define CONTACT_DIST 0.05 /* If a surface less
than
contact_dist away constrained by this surface
*/

struct obstacle *obst; /* number of obstacles */
/
struct side *bound; /* all sides of all
obstacles */
struct oside *findcolsurf();
struct mcside *mmcbound; /* bounding surf list for
mmc rep */
struct vertex *vertice; /* vertices of
obstacles */
struct line *edge; /* edges of obstacles */
struct oside *object; /* object being
considered
(trans uncerts included) */
struct oside *tempobj, *uncobject, *expobject;
/* Diff forms of object */
extern struct oside *inituncobject();
extern double *modify_soa();
extern double distunc();
double frame[3][3];

```

```

/* 1 if no rotational uncertainties in obj, obst */
/
int ROTSAMPOBJ, ROTSAMPOBST;

int **colsurf; /* surfaces that collide,
               1st dimension- which object side,
               2nd- which obstacle side */
int *clashes;
int nob, nbound, nedge, nvert, npoints, objside;
extern double fsqr();
extern int inside_obst();
/* Reads in mmc obstacle rep */
extern struct mcside *readuncmodel();
extern int readunc();
extern double uncpnt();
double **path, *point1, *point2, **contunc;
int **contacts; /* describe constraints at each pt
                */
double normal[3], motion[3];
double netmotion[3], mag; /* motion between 2
                           points */
double *uncert; /* pose uncertainty vector */
double *muncert; /* control uncertainty vector */
int *obstcolsurf; /* obstacle surfaces involved in
                  collision */
int ocs; /* # of obst surfaces involved in
          collision */
int obcs; /* # of object surfaces involved in
          collision */
int numcol; /* total number of collisions */
int ndim; /* dimension of problem, >6 if
          articulated */
int tranflag, rotflag;
int colltype; /* 0= no collision,
               1= need orient and trans unc to cause
                 collision,
               2= either trans or orient unc to cause
                 collision,
               3= just trans unc to cause collision,
               4= just orient unc to cause collision */
FILE *strategy;

/* main- Main routine for uncertainty analysis */
main(argc, argv)
int argc;
char *argv[];
{
    int i, j, mode, pathpt, flag, c1, fract, RCC;
    int type; /* 1=X and Y uncertainty, 2=RCC no
              Rotz,
              3= RCC with Rotz, 4=RotZ uncertainty,
              5=X,Y,Z,Rotx,Roty, 6=final orient,
              7= initial orient, 8=X,Y,Z,Rotx,Roty,Rotz
              9= X uncertainty, 11= Y uncertainty
              13= something involving joints */
    int rottype; /* 1= rotz; 2 rotx, rotz;
                  3 rotx, roty, rotz;
                  4= rotz, joints;
                  5= rotx, roty, joints;
                  6= rotx, roty, rotz, joints;
                  7= joints */
    struct oside *uncobject, *tempobj, *nobject;
    double dist, center[3], bottom[3], last[3]
    , mindist;
    double *tilt_axis;
    double *upper, *lower, *pp;
    double ccunc[6], ccunc1[6], ccunc2[6];

    /* Read in input, set up data structures */
    if (argc < 2)
        ndim=6;
    else
        ndim=atoi(argv[1]);

    point1=(double *)calloc(ndim,sizeof(double));
    point2=(double *)calloc(ndim,sizeof(double));
    upper=(double *)calloc(ndim,sizeof(double));
    lower=(double *)calloc(ndim,sizeof(double));
    uncert=(double *)calloc(ndim,sizeof(double));
    muncert=(double *)calloc(ndim,sizeof(double));
    initobstacles();
    initobject();
    readunc();
    readsoa(&npoints,&path);
    mmcbound=readuncmodel();
    uncobject=inituncobject();

    /* Look for first collision, don't bother w/
       starting and ending points */
    pathpt=1;
    for (i=0; i<ndim; i++) {
        lower[i]=path[i][pathpt];
        upper[i]=path[i][pathpt-1];
        ccunc[i]=contunc[i][pathpt];
    }
    last[0]=lower[0]; last[1]=lower[1]; last[2]
    =lower[2];
    /* Expand object by pose uncertainty ahead of
       time */
    expandobj(uncobject,&expobject,uncert[0],
    uncert[1],uncert[2]);
    /* Start out by considering all obstacles */
    for (i=0; i<nobs; i++)
        obst[i].consider=1;
    while (collision(lower,ccunc,expobject,last[0],
    last[1],last[2]) == 0) {
        /* Calculate constraints for each segment */
        get_contact_directions(upper,pathpt);
        pathpt++;
        if (pathpt >= npoints) {
            if ((strategy=fopen("strategy","w")) ==
            NULL)
                perror(strategy);
            fprintf(strategy,"%d0,npoints);
            dumpppath(strategy,path,npoints);
            fclose(strategy);
            exit(1);
        }
        last[0]=lower[0]; last[1]=lower[1];
        last[2]=lower[2];
        for (i=0; i<ndim; i++) {
            upper[i]=lower[i];
            lower[i]=path[i][pathpt];
            ccunc[i]=contunc[i][pathpt];
        }
        get_contact_directions(upper,pathpt);

        /* Now, find obstacles close enough to consider
           */
        mindist=1000;
        for (i=0; i<nobs; i++) {
            obst[i].dist=distunc(lower[0],lower[1],
            lower[2],i,mmcbound);
            if (obst[i].dist < mindist)
                mindist=obst[i].dist;
        }
        if (mindist < 1)
            mindist=2;
        for (i=0; i<nobs; i++)
            if (obst[i].dist > mindist+.1)
                obst[i].consider=0;
        if (pathpt == (npoints-1)) {
            lower[0]=lower[0]+0.05*(upper[0]-lower[0]);
            lower[1]=lower[1]+0.05*(upper[1]-lower[1]);
            lower[2]=lower[2]+0.05*(upper[2]-lower[2]);
            lower[3]=lower[3]+0.05*(upper[3]-lower[3]);
        }
    }
}

```

```

lower[4]=lower[4]+0.05*(upper[4]-lower[4]);
lower[5]=lower[5]+0.05*(upper[5]-lower[5]);
if (!collision(lower,ccunc,expobject,
last[0],last[1],last[2])) {
    if ((strategy=fopen("strategy","w")) ==
        NULL)
        perror(strategy);
    fprintf(strategy,"%d0,npoints);
    /* Print out pt-to-pt segment info */
    dumpppath(strategy,path,npoints-1);
    for (i=0; i<ndim; i++)
        fprintf(strategy,"%f ",
            path[i][npoints-1]);
    /* Force last motion to be guarded, probably
    bringing parts into contact here, nothing
    to be lost otherwise */
    fprintf(strategy,"1 Z ");
    for (i=0; i<6; i++)
        fprintf(strategy,"%d ",
            contacts[i][npoints-1]);
    fprintf(strategy,"0");
    fclose(strategy);
    exit(1);
}
/* If collision, find pt where it starts as
near as possible */
findcolpoint(upper,lower,point1,point2,pathpt,
ccunc1,ccunc2,expobject);
ccunc2[0]+=0.05; ccunc2[1]+=0.05;
/* Check to see if all is needed is a rcc */
RCC=collision(point1,ccunc2,expobject,last[0],
last[1],last[2]);
ccunc2[0]-=0.05; ccunc2[1]-=0.05;
/* Add new point into path, just after collision
point */
pp=modify_soa(npoints,path,point2,pathpt);
/* Consider just rotational uncertainties */
rotflag=rotcollision(point1,point2,ccunc2,
uncobject);
/* -1, nonrot coll. */
/* Consider just translational uncertainties */
tranflag=trancollision(point1,point2,ccunc2,
expobject);
/* 0, notran col. */
if (rotflag != -1 && tranflag != 0) {
    colltype=2; /* must reduce trans and rot
    unc */
}
else if (rotflag != -1) {
    colltype=4; /* must reduce rot unc */
}
else if (tranflag != 0) {
    colltype=3; /* must reduce trans or unc */
}
else
    colltype=1; /* must reduce trans unc */

/* Calculate motion vector */
for (i=0; i<3; i++)
    netmotion[i]=point2[i]-point1[i];
mag=sqrt(fsqr(netmotion[0])+fsqr(netmotion[1])+
fsqr(netmotion[2]));
netmotion[0]/=mag; netmotion[1]/=mag;
netmotion[2]/=mag;

/* Find surfaces involved in collision,
ignore flexibilities if there are any */
compress();
nobject=findcolsurf(point1,point2,ccunc1,ccunc2,
expobject,colltype,rotflag);
uncompress();

```

```

/* Find out # of obj surfs are involved in
collision */
obcs=0; numcol=0;
for (i=0; i<objside; i++) {
    numcol+=colsurf[i][0];
    if (colsurf[i][0] > 0)
        obcs++;
}
obstcolsurf=(int *)calloc(numcol,sizeof(int));

/* Develop list of surfaces of obst involved
in collision */
ocs=0;
for (i=0; i<objside; i++)
    for (j=1; j<colsurf[i][0]+1; j++)
        if (member(colsurf[i][j],obstcolsurf,ocs) ==
            0) {
            obstcolsurf[ocs]=colsurf[i][j];
            ocs++;
        }

/* must reduce rot unc */
if ((colltype == 2) || (colltype == 4)) {
    tempobj=(struct oside *)
    calloc(objside,sizeof(struct oside));
    fract=(int)(ROTSAMPOBJ-1)/2;
    if (fract != 0) {
        c1=rotobj(uncobject,&tempobj,
        ccunc2[3],ccunc2[4],ccunc2[5],
        &(uncert[6]),(double)((rotflag-fract)/fract));
    }
    else {
        c1=0;
        tempobj=uncobject;
    }
    rotype=calcrotype(lower,ccunc2,uncobject,
        tempobj,last,netmotion,
        (double)((rotflag-((ROTSAMPOBJ-1)/2.0))/
        ((ROTSAMPOBJ-1)/2.0)));
    if (c1) free(tempobj);
}
if (colltype != 4) {
    /* will try to reduce trans unc */
    type=whichtransunc(uncobject,expobject,
        ccunc,point2,last);
}
else if (colltype == 4) {
    /* need to reduce rot unc only */
    if (rotype == 2)
        type=5; /* must reduce rotx, roty */
    if (rotype == 3)
        type=8; /* must reduce rotx, roty, rotz */
    if (rotype == 1)
        type=4; /* must reduce rotz */
    if (rotype > 3)
        type=13; /* must reduce joints & possibly
        others */
}
if (colltype == 2) { /* must reduce both */
    if (rotype == 2) /* if rotx,roty */
        type=5; /* trans, rotx, roty */
    else {
        if (rotype < 4)
            type=8; /* trans, rotx, roty, rotz */
        else
            type=13; /* must reduce joints & poss
            others */
    }
}
}
if ((strategy=fopen("strategy","w")) ==
    NULL)

```

```

perror("strategy");

fprintf(strategy,"%d0,pathpt+1);
/* Print out pt-to-pt segment info */
dumppath(strategy,path,pathpt);
if (pathpt > 0) {
    if (pp == NULL)
        for (i=0; i<ndim; i++)
            fprintf(strategy,"%f ",path[i][pathpt]);
    else
        for (i=0; i<ndim; i++)
            fprintf(strategy,"%f ",pp[i]);
}
else
    for (i=0; i<ndim; i++)
        fprintf(strategy,"%f ",path[i][0]);

/* Print out unc_red_axis */
switch (type) {
case 1:
    fprintf(strategy,"2 X Y ");
    break;
case 2:
    fprintf(strategy,"4 X Y Rotx Roty ");
    break;
case 3:
    fprintf(strategy,"5 X Y Rotx Roty Rotz ");
    break;
case 4:
    fprintf(strategy,"1 Rotz ");
    break;
case 5:
    fprintf(strategy,"4 X Y Rotx Roty ");
    break;
case 8:
    fprintf(strategy,"5 X Y Rotx Roty Rotz ");
    break;
case 9:
    fprintf(strategy,"1 X ");
    break;
case 11:
    fprintf(strategy,"1 Y ");
    break;
case 13:
    switch (rottype) {
case 4:
        fprintf(strategy,"2 Rotz Joints ");
        break;
case 5:
        fprintf(strategy,
            "3 Rotx Roty Joints ");
        break;
case 6:
        fprintf(strategy,
            "4 Rotx Roty Rotz Joints ");
        break;
case 7:
        fprintf(strategy,"1 Joints ");
        break;
}
break;
}
/* Print out constraint information */
for (i=0; i<6; i++)
    fprintf(strategy,"%d ",contacts[i][pathpt]);

/* RCC flag or no */
fprintf(strategy,"152d0,RCC);
/* List obstacles involved in collision */
/* This info will be converted into misc
parm discussed in Chap6 of thesis */
for (i=0; i<ocs; i++) {
    flag=0;

```

```

    for (j=0; j<i; j++)
        if (bound[obstcolsurf[i]].obstacle ==
            bound[obstcolsurf[j]].obstacle)
            flag=1;
    if (flag != 1)
        fprintf(strategy,"%d0,
            bound[obstcolsurf[i]].obstacle);
    }
    fprintf(strategy,"%-10);
    fprintf(strategy,"%f %f %f0,
        netmotion[0],netmotion[1],netmotion[2]);
    for (i=0; i<ndim; i++)
        fprintf(strategy,"%f ",point1[i]);
    fprintf(strategy,"0);
    fclose(strategy);
}

/* findcolsurf- Given 2 pts, which surfs are coll
surfaces */
struct oside *findcolsurf(point1,point2,ccunc1
,ccunc2,
    object,flag,rotsample)
double point1[],point2[];
double ccunc1[],ccunc2[];
oside object[];
int flag,rotsample;
{
    double t,ox,oy,oz,tx,ty,tz,nx1,ny1,nz1,
        nx2,ny2,nz2,tdist;
    int i,j,k,dof,c1,c2,t1,fract,ptr,next;
    oside *outobj1,*outobj2,*tempobj;

    outobj1=(struct oside *)
        calloc(objside,sizeof(struct oside));
    outobj2=(struct oside *)
        calloc(objside,sizeof(struct oside));
    tempobj=(struct oside *)
        calloc(objside,sizeof(struct oside));

    /* Expand & rotate object by amount of
    uncertainty */
    colsurf=(int **)calloc(objside,sizeof(int));
    for (i=0; i<objside; i++)
        colsurf[i]=(int *)calloc(objside,sizeof(int));
    if ((flag == 3) || (flag == 1)) {
        c1=expandobj(object,&outobj1,ccunc1[0],
            ccunc1[1],ccunc1[2]);
        c2=expandobj(object,&outobj2,ccunc2[0],
            ccunc2[1],ccunc2[2]);
    }
    else if (flag == 4) {
        fract=(int)(ROTSAMP OBJ-1)/2;
        if (fract != 0) {
            c1=rotobj(object,&outobj1,ccunc1[3],
                ccunc1[4],ccunc1[5],
                &(uncert[6]),(double)((rotsample-fract)
                    /fract));
        }
        else {
            c1=0;
            outobj1=object;
        }
        if (fract != 0) {
            c2=rotobj(object,&outobj2,ccunc2[3],
                ccunc2[4],ccunc2[5],
                &(uncert[6]),(double)((rotsample-fract)
                    /fract));
        }
        else {
            c2=0;
            outobj2=object;
        }
    }
}

```

```

else {
    expandobj(object,&tempobj,uncert[0]+ccunc1[0],
        uncert[1]+ccunc1[1],uncert[2]+ccunc1[2]);
    fract=(int)(ROTSAMPOBJ-1)/2;
    if (fract != 0) {
        c1=rotobj(object,&outobj1,ccunc2[3],
            ccunc2[4],ccunc2[5],
            &(uncert[6]),(double)((rotsample-fract)/fract));
    }
    else {
        c1=0;
        outobj1=object;
    }
    t1=expandobj(object,&tempobj,uncert[0]
        +ccunc1[0],
        uncert[1]+ccunc1[1],uncert[2]+ccunc1[2]);
    if (fract != 0) {
        c2=rotobj(object,&outobj2,ccunc2[3],
            ccunc2[4],ccunc2[5],
            &(uncert[6]),(double)((rotsample-fract)/fract));
    }
    else {
        c2=0;
        outobj2=object;
    }
}

clashes=NULL;
/* Find surfaces where the object is one side of
it at
first pt and on the other side of it at the
second point */
for (j=0; j<objside; j++) {
    colsurf[j]=(int *)calloc(1,sizeof(int));
    dof=movable(j);
    for (k=0; k<object[j].numsamp; k++) {
        tx=ox=outobj1[j].samples[k][0];
        ty=oy=outobj1[j].samples[k][1];
        tz=oz=outobj1[j].samples[k][2];
        if (dof != -1)
            tform(tx,ty,tz,&ox,&oy,&oz,dof,
                &(point1[6]));
        tpoint(point1[0],point1[1],point1[2],
            point1[3],
            point1[4],point1[5],ox,oy,oz,&nx1,&ny1
            ,&nz1);
        tx=ox=outobj2[j].samples[k][0];
        ty=oy=outobj2[j].samples[k][1];
        tz=oz=outobj2[j].samples[k][2];
        if (dof != -1)
            tform(tx,ty,tz,&ox,&oy,&oz,dof,
                &(point2[6]));
        tpoint(point2[0],point2[1],point2[2],
            point2[3],
            point2[4],point2[5],ox,oy,oz,&nx2,&ny2
            ,&nz2);
        if (clashes != NULL)
            free(clashes);
        colpot(nx1,ny1,nz1,nx2,ny2,nz2,mmcbound);
        if (clashes[0] != 0) {
            for (i=1; i<clashes[0]+1; i++) {
                if (member2d(clashes[i],j,colsurf[j][0])
                    == 0) {
                    colsurf[j][0]++;
                    colsurf[j]=(int *)realloc(colsurf[j],
                        (colsurf[j][0]+1)*sizeof(int));
                    colsurf[j][colsurf[j][0]]=clashes[i];
                }
            }
        }
    }
}
}
}

```

```

if (c1) free(outobj1);
if (t1) free(tempobj);
return(outobj2);
}

/* findcolpoint- Try to locate collision point
within 0.0025" */
findcolpoint(upper,lower,point1,point2,pathpt,
    ccunc1,ccunc2,object)
double *upper, *lower, *ccunc1, *ccunc2;
double *point1, *point2;
oside *object;
int pathpt;
{
    double x1,x2,y1,y2,z1,z2,rotx1,rotx2,roty1,roty2
        ,
        rotz1,rotx2;
    double *point,*ccunc;
    int dim;

    point=(double *)calloc(ndim,sizeof(double));
    ccunc=(double *)calloc(ndim,sizeof(double));
    for (dim=0; dim<ndim; dim++) {
        point1[dim]=upper[dim];
        point2[dim]=lower[dim];
        ccunc1[dim]=contunc[dim][pathpt-1];
        ccunc2[dim]=contunc[dim][pathpt];
    }
    while (fsqr(point1[0]-point2[0])+
        fsqr(point1[1]-point2[1])+
        fsqr(point1[2]-point2[2]) > 0.0025) {
        for (dim=0; dim<ndim; dim++) {
            point[dim]=(point1[dim]+point2[dim])/2.0;
            ccunc[dim]=(ccunc1[dim]+ccunc2[dim])/2.0;
        }
        if (collision(point,ccunc,object,
            upper[0],upper[1],upper[2]))
            for (dim=0; dim<ndim; dim++) {
                point2[dim]=point[dim];
                ccunc2[dim]=ccunc[dim];
            }
        else
            for (dim=0; dim<ndim; dim++) {
                point1[dim]=point[dim];
                ccunc1[dim]=ccunc[dim];
            }
    }
}

/* trancollision- Returns 1 if collision occurs
when just translational uncertainties are
considered. */
int trancollision(last,point,ccunc,inobj)
double last[],point[],ccunc[];
oside *inobj;
{
    double t,ox,oy,oz,nx,ny,nz;
    double x,y,z,rotx,roty,rotz,tx,ty,tz,mag;
    int i,j,k,dof,c1,value,ptr;
    oside *outobj;

    x=point[0]; y=point[1]; z=point[2]; rotx
        =point[3];
    roty=point[4]; rotx=point[5];
    outobj=(struct oside *)
        calloc(objside,sizeof(struct oside));
    c1=expandobj(inobj,&outobj,
        ccunc[0],ccunc[1],ccunc[2]);
    value=0;
    if (c1)
        j=0;
    else
        j=objside;

```

```

compressed(x,y,z,rotx,roty,rotz,&(point[6]),
  last[0],last[1],last[2],outobj);
for (j; j<objside; j++) {
  dof=movable(j);
  ptr=0;
  /* Then check interior face points */
  for (k=0; k<(outobj)[j].numsamp; k++) {
    ox=outobj[j].samples[k][0];
    oy=outobj[j].samples[k][1];
    oz=outobj[j].samples[k][2];
    if (dof != -1) {
      tx=ox; ty=oy; tz=oz;
      tform(tx,ty,tz,&ox,&oy,&oz,dof
        ,&(point[6]));
    }
    tpoint(x,y,z,rotx,roty,rotz,ox,oy,oz,
      &nx,&ny,&nz);
    value=inside_obst(nx,ny,nz);
    if (value == 1) {
      uncompressed();
      free(outobj);
      return(1);
    }
  }
}
if (c1) free(outobj);
uncompressed();
return(0);
}

/* rotcollision- Returns -1 if collision doesn't
occur when just rotational uncertainties are
considered, else returns # of sampling slice
where collision first found */
int rotcollision(last,point,ccunc,inobj)
double last[],point[],ccunc[];
oside *inobj;
{
  double t,ox,oy,oz,nx,ny,nz,mag;
  double oxin,oyin,ozin,tx,ty,tz;
  double nxin,nyin,nzin;
  double x,y,z,rotx,roty,rotz;
  int i,j,k,dof,c1,frac,rs,value,ptr;
  oside *outobj;

  x=point[0]; y=point[1]; z=point[2]; rotx
    =point[3];
  roty=point[4]; rotz=point[5];
  if (ROTSAMPOBJ > 1)
    rs=ROTSAMPOBJ;
  else
    rs=1;
  (outobj)=(struct oside *)
    calloc(objside,sizeof(struct oside));
  frac=(int)(rs-1)/2;
  for (i=0; i<rs; i++) {
    value=0;
    if (frac != 0) {
      c1=rotobj(inobj,&outobj,ccunc[3],ccunc[4]
        ,ccunc[5],
        &(uncert[6]),(double)((i-frac)/frac));
    }
    else {
      c1=0; outobj=inobj;
    }
    compressed(x,y,z,rotx,roty,rotz,&(point[6])
      ,last[0],
        last[1],last[2],outobj);
    for (j=0; j<objside; j++) {
      dof=movable(j);
      /* Then check interior face points */
      for (k=0; k<(outobj)[j].numsamp; k++) {
        ox=outobj[j].samples[k][0];

```

```

        oy=outobj[j].samples[k][1];
        oz=outobj[j].samples[k][2];
        oxin=inobj[j].samples[k][0];
        oyin=inobj[j].samples[k][1];
        ozin=inobj[j].samples[k][2];
        if (dof != -1) {
          tx=ox; ty=oy; tz=oz;
          tform(tx,ty,tz,&ox,&oy,&oz,dof,
            &(point[6]));
          tx=oxin; ty=oyin; tz=ozin;
          tform(tx,ty,tz,&oxin,&oyin,
            &ozin,dof,&(point[6]));
        }
        tpoint(x,y,z,rotx,roty,rotz,ox,oy,oz,
          &nx,&ny,&nz);
        tpoint(x,y,z,rotx,roty,rotz,oxin,
          oyin,ozin,&nxin,&nyin,&nzin);
        value=inside_obst(nx,ny,nz);
        if (value == 1) {
          if (inside_obst(nxin,nyin,nzin) == 0) {
            if (c1) free(outobj);
            uncompressed();
            return(i);
          }
        }
      }
    }
    if (c1) free(outobj);
  }
  uncompressed();
  return(-1);
}

/* whichtransunc- Differentiates between
uncertainty axes
of X or Y and X and Y */
whichtransunc(nounc,object,ccunc,point,last)
struct oside *nounc, *object;
double *ccunc, *point, *last;
{
  int i,copied;
  double unxc,uncy,uncz;
  struct oside *tempobj;
  double temppoint[6],tunc;

  for (i=0; i<6; i++)
    temppoint[i]=point[i];
  unxc=uncert[0]+ccunc[0];
  uncy=uncert[1]+ccunc[1];
  uncz=uncert[2]+ccunc[2];
  tunc=ccunc[0]; ccunc[0]=0;
  copied=expandobj(nounc,&tempobj,0.0,uncy,uncz);
  /* If eliminating uncert X gets rid of collis,
    unc_axis=X */
  if (!collision(point,ccunc,tempobj,
    last[0],last[1],last[2])) {
    if (copied) free(tempobj);
    return(9);
  }
  if (copied) free(tempobj);
  copied=expandobj(nounc,&tempobj,uncx,0.0,uncz);
  ccunc[0]=tunc; tunc=ccunc[1]; ccunc[1]=0;
  /* If eliminating uncertainty in Y gets rid
    of collision, unc_axis=Y */
  if (!collision(point,ccunc,tempobj,
    last[0],last[1],last[2])) {
    return(11);
  }
  ccunc[1]=tunc;
  /* Else unc_axis=X,Y */
  if (copied) free(tempobj);
  return(1);
}

```



```
/* calctoolframe- Calculates tool frame based on
   lining the Z axis up with the axis of
   motion. */
```

```
calctoolframe(x,y,z)
```

```
double x,y,z;
```

```
{
  /* See Eq. 2.20 for RPY transform in Paul */
  frame[0][0]=cos(z)*cos(y);
  frame[1][0]=sin(z)*cos(y);
  frame[2][0]=-1*sin(y);
  frame[0][1]=cos(z)*sin(y)*sin(x)-sin(z)*cos(x);
  frame[1][1]=sin(z)*sin(y)*sin(x)+cos(z)*cos(x);
  frame[2][1]=cos(y)*sin(x);
  frame[0][2]=cos(z)*sin(y)*cos(x)+sin(z)*sin(x);
  frame[1][2]=sin(z)*sin(y)*cos(x)-cos(z)*sin(x);
  frame[2][2]=cos(y)*cos(x);
}
```

```
/* dumpppath- Dump out altered path */
```

```
dumpppath(fp,path,num)
```

```
FILE *fp;
```

```
double **path;
```

```
int num;
```

```
{
  int i,j;

  for (i=0; i<num; i++) {
    /* Print out end point of segment */
    for (j=0; j<ndim; j++)
      fprintf(fp,"%f",path[j][i]);
    /* unc_red_axis= NULL */
    fprintf(fp,"0 ");
    /* Print out constraints */
    for (j=0; j<6; j++)
      fprintf(fp,"%d",contacts[j][i]);
    fprintf(fp,"0");
  }
}
```

```
/* get_contact_directions-
```

```
Calculate value of constraints attribute */
```

```
get_contact_directions(point,num)
```

```
double *point;
```

```
int num;
```

```
{
  int i,plus,minus;
  double dd;

  dd=CONTACT_DIST;
  move_object(object,dd,0.0,0.0);
  plus=contact(point,object);
  move_object(object,-2.0*dd,0.0,0.0);
  minus=contact(point,object);
  move_object(object,dd,0.0,0.0);
  if (plus && minus) {
    contacts[0][num]=1; contacts[4][num]=1;
  }
  else if (plus) {
    contacts[0][num]=2; contacts[4][num]=0;
  }
  else if (minus) {
    contacts[0][num]=3; contacts[4][num]=0;
  }
  else {
    contacts[0][num]=0; contacts[4][num]=0;
  }

  move_object(object,0.0,dd,0.0);
  plus=contact(point,object);
  move_object(object,0.0,-2.0*dd,0.0);
  minus=contact(point,object);
  move_object(object,0.0,dd,0.0);
}
```

```
if (plus && minus) {
  contacts[1][num]=1; contacts[3][num]=1;
}
else if (plus) {
  contacts[1][num]=2; contacts[4][num]=0;
}
else if (minus) {
  contacts[1][num]=3; contacts[4][num]=0;
}
else {
  contacts[1][num]=0; contacts[3][num]=0;
}
```

```
move_object(object,0.0,0.0,dd);
plus=contact(point,object);
move_object(object,0.0,0.0,-2.0*dd);
minus=contact(point,object);
move_object(object,0.0,0.0,dd);
if (plus & minus)
  contacts[2][num]=1;
else if (plus)
  contacts[2][num]=2;
else if (minus)
  contacts[2][num]=3;
else
  contacts[2][num]=0;
```

```
if (contacts[4][num] && contacts[3][num])
  contacts[5][num]=1;
}
```

```
/* move_object- Perturb position of object by
   dx, dy, dz */
```

```
move_object(object,dx,dy,dz)
```

```
struct oside *object;
```

```
double dx,dy,dz;
```

```
{
  int j,k;

  for (j=0; j<objside; j++) {
    for (k=0; k<object[j].numsamp; k++) {
      object[j].samples[k][0]+=dx;
      object[j].samples[k][1]+=dy;
      object[j].samples[k][2]+=dz;
    }
  }
}
```

```
/* This file contains code for calculating
```

```
unc_red_unc
```

```
for rotational degrees of freedom. */
```

```
#include <stdio.h>
```

```
#include <math.h>
```

```
#include "/home/rvl4/q/research/pathplan
/common.h"
```

```
extern oside object[]; /* object being
   considered
   (trans uncerts included) */
```

```
extern int objside;
extern int ndim;
extern int ROTSAMPOBJ;
extern double fsqr();
extern int *obstcalsurf,ocs;
extern int rigid;
```

```
extern side *bound; /* all sides of all obstacles
   */
```

```
extern struct vertex *vertice; /* vertices of
   obstacles */
```

```
extern line *edge; /* edges of obstacles */
```

```
extern double netmotion[3];
```

```

extern double *uncert, *muncert;

/* calcrottype- Decides what axes of rot are
   causing collision */
int calcrottype(point, ccunc, object, unobject, last,
               motion, fract)
double point[], ccunc[], *last, *motion, fract;
oside *object, *unobject;
{
    oside *tempobj;
    int rottype, rotsamp, i;
    double alpha, *tuncert, *tmuncert;

    /* First, figure out what problem is, is it
       because rotation about motion vector,
       rotations
       other than rotation about motion vector, or
       both */
    tempobj=(struct oside *)
        calloc(objside, sizeof(struct oside));
    tuncert=(double *)calloc(ndim, sizeof(double));
    tmuncert=(double *)calloc(ndim, sizeof(double));
    for (i=3; i<ndim; i++) {
        tuncert[i]=uncert[i];
        tmuncert[i]=muncert[i];
    }
    alpha=ccunc[5]*fract;
    justrotz(netmotion, object, tempobj, alpha);
    rotsamp=ROTSAMPOBJ; ROTSAMPOBJ=1;
    if (collision(point, ccunc, tempobj,
                 last[0], last[1], last[2])) {
        free(tempobj);
        tempobj=(struct oside *)
            calloc(objside, sizeof(struct oside));
        notrotz(unobject, netmotion, tempobj, alpha);
        if (collision(point, ccunc, tempobj,
                     last[0], last[1], last[2])) {
            if (rigid == 0) return(3); /* rotx, roty,
               rotz */
            ROTSAMPOBJ=rotsamp;
            justjoints(tuncert, tmuncert);
            if (collision(point, ccunc, object,
                         last[0], last[1], last[2])) {
                notjoints(tuncert, tmuncert);
                if (collision(point, ccunc, object,
                             last[0], last[1], last[2]))
                    rottype=6; /* rotx, roty, rotz,
                       joints */
            }
            else
                rottype=4; /* rotx, joints */
        }
        else
            rottype=3; /* rotx, roty, rotz */
    }
    else
        rottype=1; /* rotx */
    }
    else {
        ROTSAMPOBJ=rotsamp;
        justjoints(tuncert, tmuncert);
        if (collision(point, ccunc, object,
                     last[0], last[1], last[2])) {
            if (rigid == 0) return(2); /* rotx, roty */
            notjoints(tuncert, tmuncert);
            if (collision(point, ccunc, object,
                         last[0], last[1], last[2]))
                rottype=5; /* rotx, roty, joints */
            else
                rottype=7; /* joints */
        }
        else
            rottype=2; /* rotx, roty */
    }
}

```

```

for (i=3; i<ndim; i++) {
    uncert[i]=tuncert[i];
    muncert[i]=tmuncert[i];
}
free(tuncert); free(tmuncert);
free(tempobj);
ROTSAMPOBJ=rotsamp;
return(rottype);
/* Now, decide what to do */
}

/* notrotz- Essentially subtracts out uncertainty
   about
   object frame Z axis out of uncertainty model.
   The
   input is current uncertainty model, output is a
   model of uncertainties not including
   uncertainties about
   axis of motion. */
notrotz(inobj, m, outobj, alpha)
oside inobj[], outobj[];
double m[], alpha;
{
    double u[3][3];

    u[0][0]=fsqr(m[0])+cos(alpha)*(1-fsqr(m[0]));
    u[0][1]=m[0]*m[1]*(1-cos(alpha))-m[2]
        *sin(alpha);
    u[0][2]=m[2]*m[0]*(1-cos(alpha))+m[1]
        *sin(alpha);
    u[1][0]=m[0]*m[1]*(1-cos(alpha))+m[2]
        *sin(alpha);
    u[1][1]=fsqr(m[1])+cos(alpha)*(1-fsqr(m[1]));
    u[1][2]=m[1]*m[2]*(1-cos(alpha))-m[0]
        *sin(alpha);
    u[2][0]=m[2]*m[0]*(1-cos(alpha))-m[1]
        *sin(alpha);
    u[2][1]=m[1]*m[2]*(1-cos(alpha))+m[0]
        *sin(alpha);
    u[2][2]=fsqr(m[2])+cos(alpha)*(1-fsqr(m[2]));

    gen_new_obj(inobj, outobj, u);
}

/* justrotz- Develops a model of uncertainty that
   only considers the uncertainty about the obj
   frame Z axis. The input is the rotational-
   uncertaintyless model of the object, the output
   is a model of uncertainties about the axis
   of motion. */
justrotz(m, inobj, outobj, alpha)
oside inobj[], outobj[];
double m[], alpha;
{
    int i, j, k, total;
    double u[3][3];

    alpha *= -1;
    u[0][0]=fsqr(m[0])+cos(alpha)*(1-fsqr(m[0]));
    u[0][1]=m[0]*m[1]*(1-cos(alpha))-m[2]
        *sin(alpha);
    u[0][2]=m[2]*m[0]*(1-cos(alpha))+m[1]
        *sin(alpha);
    u[1][0]=m[0]*m[1]*(1-cos(alpha))+m[2]
        *sin(alpha);
    u[1][1]=fsqr(m[1])+cos(alpha)*(1-fsqr(m[1]));
    u[1][2]=m[1]*m[2]*(1-cos(alpha))-m[0]
        *sin(alpha);
    u[2][0]=m[2]*m[0]*(1-cos(alpha))-m[1]
        *sin(alpha);

```

```

u[2][1]=m[1]*m[2]*(1-cos(alpha))+m[0]
*sin(alpha);
u[2][2]=fsqr(m[2])+cos(alpha)*(1-fsqr(m[2]));

gen_new_obj(inobj,outobj,u);
}

/* Gnerate new object rotated by matrix u */
gen_new_obj(inobj,outobj,u)
oside inobj[],outobj[];
double u[3][3];
{
    int i,j,k,total;

    for (i=0; i<obside; i++) {
        outobj[i].numsamp=inobj[i].numsamp;
        outobj[i].samples=(double **)
        calloc(inobj[i].numsamp,sizeof(double *));
        for (k=0; k<inobj[i].numsamp; k++) {
            outobj[i].samples[k]=(double *)
            calloc(3,sizeof(double));
            outobj[i].samples[k][0]=u[0][0]
            *inobj[i].samples[k][0]+
            u[0][1]*inobj[i].samples[k][1]+
            u[0][2]*inobj[i].samples[k][2];
            outobj[i].samples[k][1]=u[1][0]
            *inobj[i].samples[k][0]+
            u[1][1]*inobj[i].samples[k][1]+
            u[1][2]*inobj[i].samples[k][2];
            outobj[i].samples[k][2]=u[2][0]
            *inobj[i].samples[k][0]+
            u[2][1]*inobj[i].samples[k][1]+
            u[2][2]*inobj[i].samples[k][2];
        }
    }
}

/* justjoints- This routine resets uncert and
muncert
for just joint uncertainty */
justjoints(tuncert,tmuncert)
double *tuncert, *tmuncert;
{
    int i;

    uncert[3]=uncert[4]=uncert[5]=0;
    muncert[3]=muncert[4]=muncert[5]=0;

    for (i=6; i<ndim; i++) {
        uncert[i]=tuncert[i];
        muncert[i]=tmuncert[i];
    }
}

/* notjoints- Uses original uncert, muncert,
only ignores joint uncertainty */
notjoints(tuncert,tmuncert)
double *tuncert, *tmuncert;
{
    int i;

    for (i=3; i<6; i++) {
        uncert[i]=tuncert[i];
        muncert[i]=tmuncert[i];
    }

    for (i=6; i<ndim; i++) {
        uncert[i]=muncert[i]=0;
    }
}

/* This file contains general routines used by
uncertanal. Mostly for representing potential

```

```

field with all or some of the uncertainties
included */
#include <stdio.h>
#include <math.h>
#include "/home/rvl4/q/research/pathplan
/common.h"

#define FMPDELTA 0.000001

extern int ROTSAMPOBJ, ROTSAMPOBST;
extern int ndim;
extern int expandobj();
extern obstacle *obst; /* number of obstacles */
/
extern side *bound; /* all sides of all
obstacles */
extern mcside *mmcbound;
extern struct vertex *vertice, *something;
/* verts of obsts */
extern line *edge; /* edges of obstacles */
extern oside *object;
extern oside *tempobj, *uncobject;
extern int nobs, nbounds, nedge, nvert, npoints,
obside;
double fsqr();
int inside_obst();
int calcnompot();
int contact();
double md();
extern double **path;
extern double *uncert;
extern double *muncert;

double distnom(), distunc(), dist();

/* Calculate potential of single point, potential
is taken to be binary for uncertainty analysis,

potential=1 ==> inside fixtured part,
potential=0 ==> outside. */
int inside_obst(x,y,z)
double x,y,z;
{
    int i,j,hole,chole,result;
    double sum;
    double urx,ury,urz;
    mcside *model;

    result=0;
    model=(struct mcside *)
    calloc(nbounds,sizeof(struct mcside));
    for (i=0; i<nobs; i++) {
        if ((obst[i].shape >= 0) &&
        (obst[i].consider == 1)) {
            for (j=0; j<ROTSAMPOBST; j++) {
                if (j != 0) {
                    urx=((double)(j-(ROTSAMPOBST/2)))/
                    (ROTSAMPOBST/2))*
                    obst[i].uncert[3];
                    ury=((double)(j-(ROTSAMPOBST/2)))/
                    (ROTSAMPOBST/2))*
                    obst[i].uncert[4];
                    urz=((double)(j-(ROTSAMPOBST/2)))/
                    (ROTSAMPOBST/2))*
                    obst[i].uncert[5];
                }
            }
        }
        else {
            urx=ury=urz=0.0;
        }
    }
    calcrotmodel(mmcbound,model,i,urx,ury,
    urz);
    /* Worry first only about solid obstacles
    */
}

```

```

sum=distunc(x,y,z,i,model);
if (sum < 0.0001) {
    result=1;
    for (hole=0; hole<obst[i].nholes; hole++) {
        chole=obst[i].holes[hole];
        sum=distunc(x,y,z,chole,model);
        /* Inside hole, no reason to worry */
        if (sum < 0.0001)
            result=0;
    }
    if (result == 1) {
        j=ROTSAMPOBST; i=nobs;
    }
}
free(model);
return(result);
}

inside(dist)
double dist[];
{
    int i,flagp,flagn;

    flagp=flagn=0;
    for (i=0; i<nobs; i++) {
        if ((obst[i].shape == 1) && (dist[i] <= 0))
            flagp=1;
        if ((obst[i].shape == -1) && (dist[i] <= 0))
            flagn=1;
    }
    if ((flagp == 1) && (flagn == 0))
        return(1);
    else
        return(0);
}

/* copymodel- Copy uncert model of obst i
   from inobst to model */
copymodel(inobst,i,model)
mcside *inobst, *model;
int i;
{
    int j,sptr,k,hptr;
    for (j=0; j<obst[i].nsides; j++) {
        sptr=obst[i].side[j];
        model[sptr].a=inobst[sptr].a;
        model[sptr].b=inobst[sptr].b;
        model[sptr].c=inobst[sptr].c;
        if (bound[sptr].compressed == 1)
            model[sptr].d=inobst[sptr].d+
                bound[sptr].compress;
        else
            model[sptr].d=inobst[sptr].d;
    }
    for (j=0; j<obst[i].nholes; j++) {
        hptr=obst[i].holes[j];
        for (k=0; k<obst[hptr].nsides; k++) {
            sptr=obst[hptr].side[k];
            model[sptr].a=inobst[sptr].a;
            model[sptr].b=inobst[sptr].b;
            model[sptr].c=inobst[sptr].c;
            if (bound[sptr].compressed == 1)
                model[sptr].d=inobst[sptr].d+
                    bound[sptr].compress;
            else
                model[sptr].d=inobst[sptr].d;
        }
    }
}

```

```

/* calcrotmodel- For a given amount of
   orientational
   uncert, calculate a model of obst i rotated by

   this much uncert */
calcrotmodel(inobst,model,i,urx,ury,urz)
mcside *inobst, *model;
int i;
double urx,ury,urz;
{
    int sptr, j;
    double trans[4][4], cx, cy, cz;

    if (fabs(urx)+fabs(ury)+fabs(urz) < 0.0001) {
        copymodel(inobst,i,model);
        return;
    }
    trans[0][0]=cos(urz)*cos(ury);
    trans[0][1]=cos(urz)*sin(ury)*sin(urx)-sin(urz)
        *cos(urx);
    trans[0][2]=cos(urz)*sin(ury)*cos(urx)+sin(urz)
        *sin(urx);
    trans[1][0]=sin(urz)*cos(ury);
    trans[1][1]=sin(urz)*sin(ury)*sin(urx)+cos(urz)
        *cos(urx);
    trans[1][2]=sin(urz)*sin(ury)*cos(urx)-cos(urz)
        *sin(urx);
    trans[2][0]=-sin(ury);
    trans[2][1]=cos(ury)*sin(urx);
    trans[2][2]=cos(ury)*cos(urx);
    cx=obst[i].com[0]; cy=obst[i].com[1];
    cz=obst[i].com[2];
    trans[0][3]=cx-(cx*trans[0][0]+cy*trans[0][1]+
        cz*trans[0][2]);
    trans[1][3]=cy-(cx*trans[1][0]+cy*trans[1][1]+
        cz*trans[1][2]);
    trans[2][3]=cz-(cx*trans[2][0]+cy*trans[2][1]+
        cz*trans[2][2]);

    /* Get the new rotated 4 vector */
    for (j=0; j<obst[i].nsides; j++) {
        sptr=obst[i].side[j];
        model[sptr].a=trans[0][0]*inobst[sptr].a+
            trans[0][1]*inobst[sptr].b+
            trans[0][2]*inobst[sptr].c+
            trans[0][3]*inobst[sptr].d;
        model[sptr].b=trans[1][0]*inobst[sptr].a+
            trans[1][1]*inobst[sptr].b+
            trans[1][2]*inobst[sptr].c+
            trans[1][3]*inobst[sptr].d;
        model[sptr].c=trans[2][0]*inobst[sptr].a+
            trans[2][1]*inobst[sptr].b+
            trans[2][2]*inobst[sptr].c+
            trans[2][3]*inobst[sptr].d;
        if (bound[sptr].compressed == 1)
            model[sptr].d=inobst[sptr].d
                +bound[i].compress;
        else
            model[sptr].d=inobst[sptr].d;
    }
}

/* collision- If uncobj causes a collision when
   located
   at point p, which motion originating from start
   and last point being (lx,ly,lz) */
int collision(p,ccunc,uncobject,lx,ly,lz)
double *p,ccunc[],lx,ly,lz;
oside *uncobject;
{
    double t,ox,oy,oz,nx,ny,nz;
    double tx,ty,tz,mag;
}

```

```

int i,j,k,ptr,dof,copiedt,copiedo,fract,value;
oside *tempobj, *object;
mcside *model;

tempobj=(struct oside *)
    calloc(objside,sizeof(struct oside));
object=(struct oside *)
    calloc(objside,sizeof(struct oside));
copiedt=expandobj(uncobject,&tempobj,
    ccunc[0],ccunc[1],ccunc[2]);
fract=(int)(ROTSAMPOBJ-1)/2;
for (i=0; i<ROTSAMPOBJ; i++) {
    value=0;
    if (fract != 0) {
        copiedo=rotobj(tempobj,&object,ccunc[3],
            ccunc[4],
            ccunc[5],&(uncert[6]),(double)((i-fract)
                /fract));
    }
    else {
        object=tempobj;
        copiedo=0;
    }
    compressed(p[0],p[1],p[2],p[3],p[4],p[5],
        &(p[6]),
        lx,ly,lz,object);
    for (j=0; j<objside; j++) {
        dof=movable(j);
        for (k=0; k<object[j].numsamp; k++) {
            ox=object[j].samples[k][0];
            oy=object[j].samples[k][1];
            oz=object[j].samples[k][2];
            if (dof != -1) {
                tx=ox; ty=oy; tz=oz;
                tform(ox,oy,oz,&ox,&oy,&oz,dof,&(p[6]));
            }
            tpoint(p[0],p[1],p[2],p[3],p[4],p[5],
                ox,oy,oz,&nx,&ny,&nz);
            value=inside_obst(nx,ny,nz);
            if (value == 1) {
                if (copiedo) free(object);
                if (copiedt) free(tempobj);
                uncompressed();
                return(1);
            }
        }
    }
    if (copiedo == 1)
        free(object);
}
if (copiedt == 1)
    free(tempobj);
uncompressed();
return(0);
}

/* distunc- This funct is used to calculate the
distance between the point (x,y,z)
and the uncert model specified by bound */
double distunc(x,y,z,i,bound)
double x,y,z; /* Position of point */
int i; /* Consider the ith obstacle */
mcside bound[]; /* uncertainty model bound to
use */
{
    double sum;
    int j,sptr;

    sum=0;
    for (j=0; j < obst[i].nsides; j++) {
        sptr=obst[i].side[j];
        sum+=bound[sptr].a*x+bound[sptr].b*y+
            bound[sptr].c*z+bound[sptr].d;
    }
}

```

```

sum+=fabs(bound[sptr].a*x+bound[sptr].b*y+
    bound[sptr].c*z+bound[sptr].d);
}
return(sum);
}

/* Initialize mmc object representation */
oside *inituncobject()
{
    oside *uncobject;
    int i,j,k,ptr,nvert,total,trash,nsamp;
    double ex1,ey1,ez1,cx2,cy2,cz2,mag,**vert,*
        *samp;
    FILE *fp;

    if ((fp=fopen("uncobject","r")) != NULL) {
        fscanf(fp,"%d",&trash);
        fscanf(fp,"%d",&objside);
        uncobject=(struct oside *)
            calloc(objside,sizeof(struct oside));
        for (i=0; i<objside; i++) {
            fscanf(fp,"%d",&nsamp);
            samp=(double **)calloc(nsamp,sizeof(double
                *));
            for (k=0; k < nsamp; k++) {
                samp[k]=(double *)calloc(3,
                    sizeof(double));
                fscanf(fp,"%lf %lf %lf %lf %lf",
                    &samp[k][0],
                    &samp[k][1],&samp[k][2],&ex1,&ey1);
            }
            for (k=0; k<3; k++) {
                fscanf(fp,"%lf %lf %lf",
                    &uncobject[i].r[k][0],
                    &uncobject[i].r[k][1],
                    &uncobject[i].r[k][2]);
            }

            fscanf(fp,"%d",&uncobject[i].numloops);
            uncobject[i].nvert=(int *)
                calloc(objside,sizeof(int));
            ptr=0; total=0;
            for (k=0; k<uncobject[i].numloops; k++) {
                fscanf(fp,"%d",
                    &uncobject[i].nvert[k]);
                total+=uncobject[i].nvert[k];
            }
            vert=(double **)calloc(total+nsamp,
                sizeof(double *));
            for (k=0; k<total; k++) {
                vert[k]=(double *)calloc(3,
                    sizeof(double));
                fscanf(fp,"%lf %lf %lf",&vert[k][0],
                    &vert[k][1],&vert[k][2]);
            }
            uncobject[i].samples=(double **)
                calloc(total+nsamp,sizeof(double *));
            for (k=0; k<nsamp; k++) {
                uncobject[i].samples[k]=(double *)
                    calloc(3,sizeof(double));
                uncobject[i].samples[k][0]=samp[k][0];
                uncobject[i].samples[k][1]=samp[k][1];
                uncobject[i].samples[k][2]=samp[k][2];
            }
            for (k=0; k<total; k++) {
                uncobject[i].samples[k+nsamp]=
                    (double *)calloc(3,sizeof(double));
                uncobject[i].samples[k+nsamp][0]=vert[k][0];
                uncobject[i].samples[k+nsamp][1]=vert[k][1];
                uncobject[i].samples[k+nsamp][2]=vert[k][2];
            }
            free(vert[k]);
        }
        free(vert);
    }
}

```

```

uncobject[i].numsamp=nsamp+total;
ex1=uncobject[i].samples[nsamp+1][0]-
uncobject[i].samples[nsamp][0];
ey1=uncobject[i].samples[nsamp+1][1]-
uncobject[i].samples[nsamp][1];
ez1=uncobject[i].samples[nsamp+1][2]-
uncobject[i].samples[nsamp][2];
ex2=uncobject[i].samples[nsamp+2][0]-
uncobject[i].samples[nsamp+1][0];
ey2=uncobject[i].samples[nsamp+2][1]-
uncobject[i].samples[nsamp+1][1];
ez2=uncobject[i].samples[nsamp+2][2]-
uncobject[i].samples[nsamp+1][2];
object[i].nx=ey1*ez2-ey2*ez1;
object[i].ny=ez1*ex2-ez2*ex1;
object[i].nz=ex1*ey2-ex2*ey1;
mag=sqrt(fsqr(object[i].nx)
+fsqr(object[i].ny)+
fsqr(object[i].nz));
uncobject[i].nx=object[i].nx/=mag;
uncobject[i].ny=object[i].ny/=mag;
uncobject[i].nz=object[i].nz/=mag;
nsamp=object[i].numsamp;
for (k=0; k<nsamp; k++) {
samp[k][0]=object[i].samples[k][0];
samp[k][1]=object[i].samples[k][1];
samp[k][2]=object[i].samples[k][2];
free(object[i].samples[k]);
}
free(object[i].samples);
object[i].samples=(double **)
calloc(total+nsamp,sizeof(double *));
for (k=0; k<nsamp; k++) {
object[i].samples[k]=(double *)
calloc(3,sizeof(double));
object[i].samples[k][0]=samp[k][0];
object[i].samples[k][1]=samp[k][1];
object[i].samples[k][2]=samp[k][2];
free(samp[k]);
}
free(samp);
for (k=0; k<total; k++) {
object[i].samples[k+nsamp]=(double *)
calloc(3,sizeof(double));
object[i].samples[k+nsamp][0]=object[i].x[k];
object[i].samples[k+nsamp][1]=object[i].y[k];
object[i].samples[k+nsamp][2]=object[i].z[k];
}
free(object[i].x); free(object[i].y);
free(object[i].z);
object[i].numsamp+=total;
object[i].numloops=0;
}
fclose(fp);
}
else {
uncobject=(struct oside *)
calloc(objside,sizeof(struct oside));
for (i=0; i<objside; i++) {
uncobject[i].numloops=object[i].numloops;
uncobject[i].nvert=(int *)
calloc(objside,sizeof(int));
ptr=0; total=0;
for (k=0; k<object[i].numloops; k++) {
uncobject[i].nvert[k]=object[i].nvert[k];
total+=uncobject[i].nvert[k];
}
uncobject[i].x=(double *)
calloc(total,sizeof(double));
uncobject[i].y=(double *)
calloc(total,sizeof(double));
uncobject[i].z=(double *)
calloc(total,sizeof(double));
}

```

```

for (k=0; k<object[i].numloops; k++) {
for (j=ptr; j<ptr+object[i].nvert[k]; j++) {
uncobject[i].x[j]=object[i].x[j];
uncobject[i].y[j]=object[i].y[j];
uncobject[i].z[j]=object[i].z[j];
}
ptr=j;
}
ex1=uncobject[i].x[1]-uncobject[i].x[0];
ey1=uncobject[i].y[1]-uncobject[i].y[0];
ez1=uncobject[i].z[1]-uncobject[i].z[0];
ex2=uncobject[i].x[2]-uncobject[i].x[1];
ey2=uncobject[i].y[2]-uncobject[i].y[1];
ez2=uncobject[i].z[2]-uncobject[i].z[1];
object[i].nx=ey1*ez2-ey2*ez1;
object[i].ny=ez1*ex2-ez2*ex1;
object[i].nz=ex1*ey2-ex2*ey1;
mag=sqrt(fsqr(object[i].nx)
+fsqr(object[i].ny)+
fsqr(object[i].nz));
uncobject[i].nx=object[i].nx/=mag;
uncobject[i].ny=object[i].ny/=mag;
uncobject[i].nz=object[i].nz/=mag;
}
return(uncobject);
}

/* readunc- Read file "uncertainties" in format:
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Positional uncertainties of first obstacle
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Center of mass of first obstacle
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Positional uncertainties of second obstacle
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Center of mass of second obstacle
...
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Center of mass of obs-th obstacle
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Positional uncertainties of object
f.f.f.f.f.f.f.f.f.f.f.f.f.f.f.f
Control uncertainties of object */
readunc()
{
FILE *fp;
int i,j;

if ((fp=fopen("uncertainties","r")) == NULL)
perror("uncert");
for (i=0; i<nobs; i++) {
for (j=0; j<6; j++)
fscanf(fp,"%lf",&obst[i].uncert[j]);
for (j=0; j<3; j++)
fscanf(fp,"%lf",&obst[i].com[j]);
}
for (j=0; j<ndim; j++)
fscanf(fp,"%lf",&uncert[j]);
for (j=0; j<6; j++)
fscanf(fp,"%lf",&muncert[j]);
fscanf(fp,"%d %d",&ROTSAMPOBJ,&ROTSAMPOBST);
}

/* rotobj- Calculates uncertainty object per the
given
rotational uncertainty */
rotobj(inobj,outobj,uncrotx,uncroty,uncrotz,ujoint
fraction)
oside *inobj, **outobj;
double uncrotx,uncroty,uncrotz,*ujoint,fraction;

```

```

{
double trans[3][3],tempx,tempy,tempz,tot;
int i,j,k,dof,total;

tot=fsqr(uncrotx)+fsqr(uncroty)+fsqr(uncrotx);
for (i=6; i<ndim; i++)
tot+=fsqr(ujoint[i-6]);
if ((tot < 0.00001) || (fraction == 0)) {
(*outobj)=inobj;
return(0);
}

uncrotx*=fraction; uncroty*=fraction; uncrotx*=
=fraction;
for (i=0; i<ndim-6; i++)
ujoint[i]*=fraction;
/* Calculate RPY xform, from pg. 47 of Paul's
book. */
trans[0][0]=cos(uncrotx)*cos(uncroty);
trans[0][1]=cos(uncrotx)*sin(uncroty)
*sin(uncrotx)-
sin(uncrotx)*cos(uncrotx);
trans[0][2]=cos(uncrotx)*sin(uncroty)
*cos(uncrotx)+
sin(uncrotx)*sin(uncrotx);
trans[1][0]=sin(uncrotx)*cos(uncroty);
trans[1][1]=sin(uncrotx)*sin(uncroty)
*sin(uncrotx)+
cos(uncrotx)*cos(uncrotx);
trans[1][2]=sin(uncrotx)*sin(uncroty)
*cos(uncrotx)-
cos(uncrotx)*sin(uncrotx);
trans[2][0]=-sin(uncroty);
trans[2][1]=cos(uncroty)*sin(uncrotx);
trans[2][2]=cos(uncroty)*cos(uncrotx);

/* Now replace each vertice v with v'=RPYv */
(*outobj)=(struct oside *)
calloc(objside,sizeof(struct oside));
for (i=0; i<objside; i++) {
dof=movable(i);
(*outobj)[i].numsamp=inobj[i].numsamp;
(*outobj)[i].samples=(double **)
calloc(inobj[i].numsamp,sizeof(double
**));
for (k=0; k<inobj[i].numsamp; k++) {
(*outobj)[i].samples[k]=(double *)
calloc(3,sizeof(double));
tempx=inobj[i].samples[k][0];
tempy=inobj[i].samples[k][1];
tempz=inobj[i].samples[k][2];
(*outobj)[i].samples[k][0]=
trans[0][0]*tempx+trans[0][1]*tempy+
trans[0][2]*tempz;
(*outobj)[i].samples[k][1]=
trans[1][0]*tempx+trans[1][1]*tempy+
trans[1][2]*tempz;
(*outobj)[i].samples[k][2]=
trans[2][0]*tempx+trans[2][1]*tempy+
trans[2][2]*tempz;
if (dof != -1) {
tform((*outobj)[i].samples[k][0],
(*outobj)[i].samples[k][1],
(*outobj)[i].samples[k][2],
&tempx,&tempy,&tempz,dof,ujoint);
(*outobj)[i].samples[k][0]=tempx;
(*outobj)[i].samples[k][1]=tempy;
(*outobj)[i].samples[k][2]=tempz;
}
}
}
return(1);
}

```

```

/* expandobj- Produce new obj grown by amount of
specified uncert. Note- This routine assumes
center of object is (0,0,0) */
int expandobj(inobj,outobj,uncx,uncy,uncz)
oside *inobj, **outobj;
double uncx,uncy,uncz;
{
int i,k,total;
double mag,vector[3];

if (fsqr(uncx)+fsqr(uncy)+fsqr(uncz) < 0.0001) {
(*outobj)=inobj;
return(0);
}

(*outobj)=(struct oside *)
calloc(objside,sizeof(struct oside));
for (i=0; i<objside; i++) {
(*outobj)[i].numsamp=inobj[i].numsamp;
(*outobj)[i].samples=(double **)
calloc(inobj[i].numsamp,sizeof(double
**));
for (k=0; k<inobj[i].numsamp; k++) {
(*outobj)[i].samples[k]=(double *)
calloc(3,sizeof(double));
(*outobj)[i].samples[k][0]=
inobj[i].samples[k][0]+uncx*inobj[i].nx;
(*outobj)[i].samples[k][1]=
inobj[i].samples[k][1]+uncy*inobj[i].ny;
(*outobj)[i].samples[k][2]=
inobj[i].samples[k][2]+uncz*inobj[i].nz;
}
(*outobj)[i].numloops=inobj[i].numloops;
(*outobj)[i].nvert=(int *)
calloc(inobj[i].numloops,sizeof(int));
}
return(1);
}

/* Flags sides of obstacles that will be
compressed if point
(x,y,z,rotx,roty,rotz) is obtained from (lx,ly,
,lz) */
compressed(x,y,z,rotx,roty,rotz,th,lx,ly,lz
,object)
double x,y,z,lx,ly,lz,rotx,roty,rotz,*th;
struct oside *object;
{
int sptr,j,k,l,dof;
double nx,ny,nz,lx1,ly1,lz1,ox,oy,oz,t;

for (sptr=0; sptr<nbounds; sptr++) {
if (bound[sptr].compflag == 1) {
for (j=0; j<objside; j++) {
dof=movable(j);
bound[sptr].compressed=0;
for (k=0; k<object[j].numsamp; k++) {
if (dof != -1)
tform(object[j].samples[k][0],
object[j].samples[k][1],
object[j].samples[k][2],
&ox,&oy,&oz,dof,th);
else {
ox=object[j].samples[k][0];
oy=object[j].samples[k][1];
oz=object[j].samples[k][2];
}
tpoint(x,y,z,rotx,roty,rotz,ox,oy,oz,
&nx,&ny,&nz);
tpoint(lx,ly,lz,rotx,roty,rotz,ox,oy,oz,
&lx1,&ly1,&lz1);
if (moved(sptr,nx,ny,nz,lx1,ly1,lz1) ==
1) {

```

```

        bound[sptr].compressed=1;
        k=object[j].numsamp;
    }
}
}
}

/* Uncompresses object */
uncompressed()
{
    int i;

    for (i=0; i<nbounds; i++)
        bound[i].compressed=0;
}

/* agreeable_axis- Finds axis about which obj may
be
tilted w/o causing a collision. */
double *agreeable_axis(object,point,ccunc,last)
double *point,*ccunc,*last;
struct oside *object;
{
    double u[3][3],m[4];
    struct oside *tempobj;

    tempobj=(struct oside *)
        calloc(objside,sizeof(struct oside));
    m[0]=1; m[1]=m[2]=0; m[3]=.1745;
    get_transform(m,u);
    transform_object(object,tempobj,u);
    if (!collision(point,ccunc,tempobj,
        last[0],last[1],last[2])) {
        free(tempobj);
        return(m);
    }
    m[3]= -.1745;
    get_transform(m,u);
    transform_object(object,tempobj,u);
    if (!collision(point,ccunc,tempobj,
        last[0],last[1],last[2])) {
        free(tempobj);
        return(m);
    }
    m[1]=1; m[0]=m[2]=0; m[3]=.1745;
    get_transform(m,u);
    transform_object(object,tempobj,u);
    if (!collision(point,ccunc,tempobj,
        last[0],last[1],last[2])) {
        free(tempobj);
        return(m);
    }
    m[3]= -.1745;
    get_transform(m,u);
    transform_object(object,tempobj,u);
    if (!collision(point,ccunc,tempobj,
        last[0],last[1],last[2])) {
        m[1]=1; m[0]=m[2]=0; m[3]= -.1745;
        free(tempobj);
        return(m);
    }
    m[0]=m[1]=m[2]=m[3]=0.0;
    return(m);
}

/* get_transform- Calculates an arbitrary
homogeneous xform for
quaternion. Simple routine detailed in IEEE
Trans
R&A June 1990 */
get_transform(m,u)

```

```

double m[4],u[3][3];
{
    double kx,ky,kz,ct,st,vt;

    kx=m[0]; ky=m[1]; kz=m[2]; ct=cos(m[3]); st
        =sin(m[3]);
    vt=1-ct; u[0][0]=fsqr(kx)*vt+ct;
    u[1][0]=kx*ky*vt+kz*st; u[2][0]= kx*kz*vt-ky*st;
    u[0][1]=kx*ky*vt-kz*st; u[1][1]=fsqr(ky)*vt+ct;
    u[2][1]=ky*kz*vt+kx*st; u[0][2]=kz*kx*vt+ky*st;
    u[1][2]=kz*ky*vt-kx*st; u[2][2]=fsqr(kz)*vt+ct;
}

/* transform_object- Xforms obj into tobject via
xform trans. */
transform_object(object,tobject,trans)
struct oside *object,*tobject;
double trans[3][3];
{
    int i,k;
    double tempx,tempy,tempz;

    for (i=0; i<objside; i++) {
        tobject[i].numsamp=object[i].numsamp;
        tobject[i].samples=(double **)
            calloc(object[i].numsamp,sizeof(double
                *));
        for (k=0; k<object[i].numsamp; k++) {
            tobject[i].samples[k]=(double *)
                calloc(3,sizeof(double));
            tempx=object[i].samples[k][0];
            tempy=object[i].samples[k][1];
            tempz=object[i].samples[k][2];
            tobject[i].samples[k][0]=trans[0][0]*tempx+
                trans[0][1]*tempy+trans[0][2]*tempz;
            tobject[i].samples[k][1]=trans[1][0]*tempx+
                trans[1][1]*tempy+trans[1][2]*tempz;
            tobject[i].samples[k][2]=trans[2][0]*tempx+
                trans[2][1]*tempy+trans[2][2]*tempz;
        }
    }

    /* contact- If object causes a collision
when located at point p, return 1 */
int contact(p,object)
double *p;
oside *object;
{
    double nx,ny,nz;
    double tx,ty,tz,mag,value,t,ox,oy,oz;
    int i,j,k,l,dof;

    value=0;
    for (j=0; j<objside; j++) {
        dof=movable(j);
        for (k=0; k<object[j].numsamp; k++) {
            ox=object[j].samples[k][0];
            oy=object[j].samples[k][1];
            oz=object[j].samples[k][2];
            if (dof != -1) {
                tx=ox; ty=oy; tz=oz;
                tform(ox,oy,oz,&ox,&oy,&oz,dof,&(p[6]));
            }
            tpoint(p[0],p[1],p[2],p[3],p[4],p[5],
                ox,oy,oz,&nx,&ny,&nz);
            value=calcnompot(nx,ny,nz);
            if (value == 1)
                return(1);
        }
    }
    return(0);
}

```



```

/* Calculate potential of a single point,
potential is taken to be binary for uncertainty
analysis, potential=1 ==> in fixtured part,
potential=0 ==> outside fixtured part. */
int calcompot(x,y,z)
double x,y,z;
{
    int i,j,hole,chole,result,sptr,sum;

    result=0;
    for (i=0; i<nobs; i++) {
        if ((obst[i].shape >= 0) &&
            (obst[i].consider == 1)) {
            sum=0;
            for (j=0; j<obst[i].nsides; j++) {
                sptr=obst[i].side[j];
                if (bound[sptr].a*x+bound[sptr].b*y+
                    bound[sptr].c*z+bound[sptr].d > 0) {
                    sum=1;
                    j=obst[i].nsides;
                }
            }
            if (sum == 0) {
                result=1;
                for (hole=0; hole < obst[i].nholes; hole++) {
                    choleside=obst[i].holes[hole];
                    for (j=0; j<obst[choleside].nsides; j++) {
                        sptr=obst[choleside].side[j];
                        if (bound[sptr].a*x+bound[sptr].b*y+
                            bound[sptr].c*z+ bound[sptr].d > 0) {
                            sum=1;
                            j=obst[choleside].nsides;
                        }
                    }
                    if (sum == 0) {
                        result=0;
                        hole=obst[i].nholes;
                    }
                }
            }
            if (result == 1)
                i=nobs;
        }
    }
    return(result);
}

```

```

/* This file has utility routines used by Phase 2
Stage 1 */
#include <stdio.h>
#include <math.h>
#include "/home/rvl4/q/research/pathplan
/common.h"

```

```

extern struct obstacle *obst; /* number of
obstacles */
extern struct side *bound; /* all sides of all
obstacles */
extern struct vertex *vertec; /* vertices of
obstacles */
extern struct line *edge; /* edges of obstacles */
extern struct oside realobj[]; /* sides of real
obj
(described by vertices) */
extern int *clashes;
extern double distunc();
extern struct oside object[];
extern struct mcside *mmcbound;
extern int expandobj(), rotobj();
extern int ndim;

```

```

extern struct vertex *something;

```

```

extern int **colsurf; /* surfaces that collide,
1st dimension- which obj side,
2nd- which obst side */
extern int nobs, nbounds, nedge, nvert, npoints,
objside;
extern double fsqr();
extern double **path, *uncert, *muncert, *
*contunc;
extern double frame[3][3];
extern int **contacts;
mcside *readuncmodel();

```

```

/* member2d- Member function for 2d array, colsurf
*/

```

```

member2d(i,comp,j)
int i,comp,j;
{
    int k;

    for (k=0; k<j; k++)
        if (colsurf[comp][k+1]==i)
            return(1);
    return(0);
}

```

```

/* member- Is i a member of the first j elements
of list? */

```

```

member(i,list,j)
int i,*list,j;
{
    int k;

    for (k=0; k<j; k++)
        if (list[k]==i)
            return(1);
    return(0);
}

```

```

/* readsoa- Read soa path */

```

```

readsoa(npoints,path)
int *npoints;
double ***path;
{
    int i,j;
    FILE *fp;
    double dx,dy,dz;

```

```

    if ((fp=fopen("soa.out","r")) == NULL)
        perror("soa.out");
    fscanf(fp,"%d0,npoints);

```

```

/* Allocate arrays */

```

```

*path=(double **)calloc(ndim,sizeof(double *));
contunc=(double **)calloc(ndim,sizeof(double
*));
contacts=(int **)calloc(ndim,sizeof(int *));
for (i=0; i<ndim; i++) {
    (*path)[i]=(double *)
    calloc(npoints+1,sizeof(double));
    (contunc)[i]=(double *)
    calloc(npoints+1,sizeof(double));
    (contacts)[i]=(int *)
    calloc(npoints+1,sizeof(int));
}

```

```

for (i=0; i<(*npoints); i++)
    for (j=0; j<ndim; j++)
        fscanf(fp,"%lf",&(*path)[j][i]);
fclose(fp);
contunc[0][0]=0; contunc[1][0]=0; contunc[2][0]
=0;

```

```

/* Can't take pose uncertainty into account
ahead of time as for translational components
*/
contunc[3][0]=uncert[3]; contunc[4][0]
=uncert[4];
contunc[5][0]=uncert[5];
/* Calculate control uncertainty at each point */
for (i=1; i<(*npoints); i++) {
    calctoolframe((*path)[3][i],(*path)[4][i],
    (*path)[5][i]);
    dx=fabs((*path)[0][i]-(*path)[0][i-1]);
    dy=fabs((*path)[1][i]-(*path)[1][i-1]);
    dz=fabs((*path)[2][i]-(*path)[2][i-1]);
    contunc[0][i]=contunc[0][i-1]+fabs(muncert[0]*
    (frame[0][0]*dx+frame[0][1]*dy
    +frame[0][2]*dz));
    contunc[1][i]=contunc[1][i-1]+fabs(muncert[1]*
    (frame[1][0]*dx+frame[1][1]*dy
    +frame[1][2]*dz));
    contunc[2][i]=contunc[2][i-1]+fabs(muncert[2]*
    (frame[2][0]*dx+frame[2][1]*dy
    +frame[2][2]*dz));

    dx=fabs((*path)[3][i]-(*path)[3][i-1]);
    dy=fabs((*path)[4][i]-(*path)[4][i-1]);
    dz=fabs((*path)[5][i]-(*path)[5][i-1]);
    contunc[3][i]=contunc[3][i-1]+fabs(muncert[3]*
    (frame[0][0]*dx+frame[0][1]*dy
    +frame[0][2]*dz));
    contunc[4][i]=contunc[4][i-1]+fabs(muncert[4]*
    (frame[1][0]*dx+frame[1][1]*dy
    +frame[1][2]*dz));
    contunc[5][i]=contunc[5][i-1]+fabs(muncert[5]*
    (frame[2][0]*dx+frame[2][1]*dy
    +frame[2][2]*dz));
}

/* modify-soa- Add a point to the soa file if
there is a
large dist between the collision point and the
next
point on the path, else another collision may
be
missed. */
double *modify_soa(npoints,path,point,pathpt)
int npoints;
double **path;
double *point;
int pathpt;
{
    int i,j,flag;
    FILE *fp;
    double distance,*new;

    flag=0; new=NULL;
    if ((fp=fopen("soa.out","w")) == NULL)
        perror("soa.out");
    distance=fsqr(path[0][pathpt]-point[0])+
    fsqr(path[1][pathpt]-point[1])+
    fsqr(path[2][pathpt]-point[2]);
    if ((distance > 0.1) || (pathpt == npoints))
        flag=1;
    fprintf(fp,"%d0,(npoints-pathpt)+flag);
    if (flag == 1) {
        new=(double *)calloc(ndim,sizeof(double));
        if (distance > 0.1) {
            for (j=0; j<ndim; j++) {
                new[j]=point[j]+((0.10/distance)*
                (path[j][pathpt]-point[j]));
                fprintf(fp,"%f ",new[j]);
            }
        }
    }
}

```

```

}
else {
    for (j=0; j<ndim; j++) {
        new[j]=point[j]+(0.5*
        (path[j][pathpt]-point[j]));
        fprintf(fp,"%f ",new[j]);
    }
    fprintf(fp,"0");
}
for (i=pathpt; i<npoints; i++) {
    for (j=0; j<ndim; j++)
        fprintf(fp,"%f ",path[j][i]);
    fprintf(fp,"0");
}
fclose(fp);
return(new);
}

/* colpot- For 2 p1 & p2, if p2 is on inside of
surf but
p1 is on outside, flag this surface in data
structure.
Used to tell what object surfaces and obstacle
surfaces involved in collision */
int colpot(x1,y1,z1,x2,y2,z2,unbound)
double x1,y1,z1,x2,y2,z2;
struct mcside *unbound;
{
    int i,j,k,sptr,clashstart,nclash;
    double sum1, sum2, sum;

    clashes=(int *)calloc(10,sizeof(int));
    nclash=clashes[0]=0;
    for (i=0; i<nobs; i++) {
        sum=0;
        clashstart=nclash;
        for (j=0; j<obst[i].nsides; j++) {
            sptr=obst[i].side[j];
            if (obst[i].shape >= 0) {
                sum1=unbound[sptr].a*x1+unbound[sptr].b*y1
                +
                unbound[sptr].c*z1 +unbound[sptr].d;
                sum2=unbound[sptr].a*x2+unbound[sptr].b*y2
                +unbound[sptr].c*z2+unbound[sptr].d;
                sum+=unbound[sptr].a*x2+unbound[sptr].b*y2
                +unbound[sptr].c*z2+unbound[sptr].d;
                sum+=fabs(unbound[sptr].a*x2
                +unbound[sptr].b*y2+unbound[sptr].c*z2
                +unbound[sptr].d);
            }
            else {
                sum2=unbound[sptr].a*x1+unbound[sptr].b*y1
                +unbound[sptr].c*z1+unbound[sptr].d;
                sum1=unbound[sptr].a*x2+unbound[sptr].b*y2
                +unbound[sptr].c*z2+unbound[sptr].d;
                sum+=unbound[sptr].a*x1+unbound[sptr].b*y1
                +unbound[sptr].c*z1+unbound[sptr].d;
                sum+=fabs(unbound[sptr].a*x1
                +unbound[sptr].b*y1+unbound[sptr].c*z1
                +unbound[sptr].d);
            }
        }
        if ((sum1 > 0) && (sum2 <= 0))
            if ((obst[bound[sptr].obstacle].holes ==
            0) ||
            (real_clash(x2,y2,z2
            ,bound[sptr].obstacle,
            unbound) == 1)) {
                nclash++;
                if (nclash > 9)
                    clashes=(int *)realloc(clashes,(nclash+1)
                    *sizeof(int));
                clashes[nclash]=sptr;
            }
    }
}

```

```

    }
    if (sum > 0.00001)
        nclash=clashstart;
    }
    clashes[0]=nclash;
}

/* real_clash- */
real_clash(x,y,z,theobst,model)
double x,y,z;
int theobst;
struct mcside *model;
{
    int i,optr;
    double value;

    for (i=0; i<obst[theobst].nholes; i++) {
        optr=obst[theobst].holes[i];
        if (distunc(x,y,z,optr,model) < 0.000001)
            return(0);
    }
    return(1);
}

/* readuncmodel- Most everything here is thrown
out
because its redundant info already stored in
obst
data structures, only mcside info is needed. */
mcside *readuncmodel()
{
    int i,j,sptr;
    mcside *outmodel;
    FILE *fp;
    double uncx, uncy, uncx;
    double ftrash;
    int nob, nvert, trash, nedge, nfaces, total;

    outmodel=(struct mcside *)
        calloc(nbounds, sizeof(struct mcside));

    if ((fp=fopen("uncobstacles", "r")) == NULL)
        perror("uncobstacles");
    fscanf(fp, "%d", &nob);
    for (i=0; i<nob; i++) {
        fscanf(fp, "%d", &trash);
        for (j=0; j<obst[i].nsides; j++)
            fscanf(fp, "%d", &trash);
        fscanf(fp, "%d %d", &trash, &trash);
    }

    fscanf(fp, "165d0", &nvert);
    for (i=0; i<nvert; i++) {
        fscanf(fp, "%f %f %f", &ftrash, &ftrash, &ftrash);
        fscanf(fp, "%d", &nedge);
        for (j=0; j<nedge; j++)
            fscanf(fp, "%d", &trash);
    }

    fscanf(fp, "165d0", &nedge);
    for (i=0; i<nedge; i++) {
        fscanf(fp, "%lf %lf %lf %d %d0", &ftrash, &ftrash, &ftrash, &trash, &trash);
        fscanf(fp, "%d", &nfaces);
        for (j=0; j<nfaces; j++)
            fscanf(fp, "%d", &trash);
    }

    fscanf(fp, "%d0", &nbounds);
    for (i=0; i<nbounds; i++) {
        fscanf(fp, "%lf %lf %lf %lf %lf0",

```

```

        &outmodel[i].a, &outmodel[i].b,
        &outmodel[i].c, &outmodel[i].d);
    fscanf(fp, "%d %d %d %d"
        , &outmodel[i].obstacle,
        &nedge, &trash, &nvert);
    total=0;
    for (j=0; j<nedge; j++) {
        fscanf(fp, "%d", &nfaces);
        total+=nfaces;
    }
    for (j=0; j<total; j++)
        fscanf(fp, "%d", &trash);
    fscanf(fp, "%d %lf", &trash, &ftrash);
}
fclose(fp);
return(outmodel);
}

```

SUPERVISE

```

;; top.cl- This is the top level code for the
;; assembly
;; motion planning supervisor. The supervisor
;; first
;; calls converts the assembly representation
;; into a
;; potential field representation and
;; generates other
;; extraneous input needed (such as start
;; point, goal
;; point, etc.). It then invokes Phase 1 and
;; Phase 2 as
;; needed. It finally generates a file of
;; motions, to be
;; executed by the execution unit later on.
;;
;; supervise- This routine supervises motion
;; planning.
(defun supervise (assem)
  (let ((list) (item) (result) (flag t)
        (motionfile)
        (gp (assembly-grasped-part assem))
        (fp (assembly-fixtured-part assem)))
    ;; Delete old motion file
    (excl:run-shell-command "rm /home/rvl/q
/motions")
    ;; Open up file where motion plan is dumped
    (setq motionfile (open '/home/rvl/q/motions
:direction :output
:if-does-not-exist :create
:if-exists :overwrite))
    ;; If this (sub)assembly has already been formed
    ;; quit.
    (cond ((assembly-done assem) t)

    ;; If not done, first make sure a plan has been
    ;; generated for the left and right
    ;; subassemblies
    ;; are done.
    (t (if (and (typep gp 'assembly)
                (null (assembly-done gp)))
            (supervise gp))
        (if (and (typep fp 'assembly)
                (null (assembly-done fp)))
            (supervise fp))

```

```

;; Set up starting position for mating oper,
;; if
;; hasn't already been set, a method
;; associated
;; w assemblies will automatically be invoked
;;
;; to calculate a good starting position.

;; Develop motion plan
(setq *first* nil)
(setq list (motion-plan assem))
(setq *list* list)

;; Set up motion file
(format motionfile "~s ~%" (length list))
(close motionfile)
;; Write each motion into file
(dolist (item list result)
  (cond (flag
    (setq result (execute item))
    ;; If motion invalid there is a problem
    (cond ((null result)
      (setq flag nil)
      (format t
        "Unrecoverable error!"))))

    ;; Since grasped part has been moved, it's
    ;; pose
    ;; must be updated so that its placement is
    ;; known in later steps.
    ;; Delete intermediate result files created.
    (setf (assembly-done assem) t)
    (excl:run-shell-command "rm /home/rvl/q/obj
      *")
    (excl:run-shell-command "rm /home/rvl/q/mm
      *")
    (excl:run-shell-command "rm /home/rvl/q/
      *.out")
    (excl:run-shell-command "rm /home/rvl/q/
      /strategy")
    (excl:run-shell-command "rm /home/rvl/q/obs
      *")
    (excl:run-shell-command "rm /home/rvl/q/unc
      *"))))

;; motion-plan- Develops a motion plan
;; consisting of a list of
;; motion data structures.
(defun motion-plan (assem)
  ;; Case statement for separating out different
  ;; cases
  ;; and invoking the correct method of motion
  ;; planning.
  ;; (Cases to be added as they are implemented)
  (cond ((typep (x (assembly-connection assem))
    'threaded)
    (thread-motion-plan assem))
    (t (normal-motion-plan assem))))

;; thread-motion-plan- Develops a motion plan
;; for a threading
;; type connection.
(defun thread-motion-plan (assem)
  (let ((list nil))

    ;; First find motion plan from start to
    ;; threading position.
    (setq list (normal-motion-plan assem))

    ;; Then append a threading motion onto the
    ;; end.
    ;; This is generated symbolically.

```

```

    (setq list (append list (find-thread-motion
      assem)))))

;; update-grasped-assembly- Once a grasped part
;; has been put
;; into its assembled position, its pose
;; must
;; must be updated. Similarly, once a
;; grasped
;; subassembly has been put into assembled
;; position, the poses of it's component
;; parts must be updated.
(defun update-grasped-assembly (xform sub)
  ;; Make sure pose of part is initially identity.
  (cond ((typep sub 'part) (setf (part-pose sub)
    '(0 0 0 0 0 0))))
  ;; Call recursive routine to update the pose.
  (update-grasped-assembly-r xform sub))

;; update-grasped-assembly-r- If simply a part
;; update part pose,
;; else update assembly tree.
(defun update-grasped-assembly-r (xform sub)
  (cond ((typep sub 'part)
    (setf (part-pose sub)
      (mmult xform (make-xform (part-pose
        sub)))))
    (t (update-grasped-assembly xform
      (assembly-grasped-part sub))
      (update-grasped-assembly xform
        (assembly-fixtured-part
          sub)))))

;; find-thread-motion- Find a threading motion
;; to finish the
;; mating of the desired part.
;; NOTE: This assumes that threading will occur
;; along -Z axis
;; this of course can be extended, but we
;; need a robot
;; capable of doing compliant motion along
;; arbitrary axes.
(defun find-thread-motion (assem)
  (let ((holez) (feat1) (feat2) (connection) (item)
    (gfeats (assembly-feat1 assem))
    (ffeats (assembly-feat2 assem)))
    ;; Locate threading data struct from ptrs in
    ;; assembly.
    (dolist (item (assembly-connection assem)
      feat1)
      (cond ((equal (x item) 'thread)
        (setq feat1 (car gfeats))
        (setq feat2 (car ffeats))
        (t (setq gfeats (cdr gfeats))
          (setq ffeats (cdr ffeats)))))
      (setq holez (z (assembly-pos1 assem)))
      (cond
        ((> (feature-sign
          (eval (class-name (class-of feat1)))) 0)
          (setq thread-screw
            (make-instance 'move
              :motion '(guarded-move
                "(r (+ (r (move-initial-position
                  thread-screw))
                  15))
                "(fz ,(- *negative-z-contact-force* 3)
                  (z (move-initial-force thread-screw))))
                "(fx ,(- *forcex* (x (move-initial-force
                  thread-screw))))
                "(fy ,(- *forcey* (y (move-initial-force
                  thread-screw))))
                "(Tx 0.3) "(Tx -0.3) "(Ty 0.3) "(Ty
                  -0.3)"))

```

```

;; position-bound
'(((x (move-initial-position thread-screw))
0.1)
((y (move-initial-position thread-screw))
0.1)
(holez 0.3) (180 5) (0 5)
((+ 15
(r (move-initial-position thread-screw)))
3))
:protectional-terminating-conditions
'(((Gr (+ (r (move-initial-position
thread-screw))
14)))
:goal-terminating-conditions '((Tr -2))))))

(t (setq thread-screw
(make-instance 'move
:motion 'guarded-move
'(r ,(- (r (move-initial-position
thread-screw))
15))
'(fz ,(- (*negative-z-contact-force* 3)
(z (move-initial-force thread-screw))))
'(fx ,(- *forcecx* (x (move-initial-force
thread-screw))))
'(fy ,(- *forcecy* (y (move-initial-force
thread-screw))))
'(Tx 0.3) '(Tx -0.3) '(Ty 0.3) '(Ty
-0.3))
:position-bound
'(((x (move-initial-position thread-screw))
0.1)
((y (move-initial-position thread-screw))
0.1)
(holez 0.3) (180 5) (0 5)
((- 15 (r (move-initial-position
thread-screw)))
3))
:protectional-terminating-conditions
'(((Lr (- (r (move-initial-position
thread-screw))
14)))
:goal-terminating-conditions '((Tr
-2))))))

;; normal-motion-plan- Develops motion plan
;; thru
;; normal channels.
(defun normal-motion-plan (assem)
(let ((result nil) (done nil) (subplan)
(uncertanalcom))

;; Build up files for input to pathplan and
;; uncertanal
(build-input assem)
(excl:run-shell-command "rm /home/rvl/q
/slp.out")
;; Execute standard motion planner
(excl:run-shell-command "date")
(excl:run-shell-command
"rsh rvl4 /home/rvl/q/research/pathplan
/pathplan /home/rvl/q/research/run/profile")
(excl:run-shell-command "date")

(excl:run-shell-command "rm /home/rvl/q
/soa.out")
;; Generated path from goal to start, reverse
;; points
;; for rest of analysis.
(cond ((reverse-path)
(setq *first* nil)
;; Formulate string for invoking uncertainty
;; analysis and

;; segment characterization.
(setq uncertanalcom (concatenate 'string
"uncertanal "
(itoa *dof*)))

;; Phase 2 loop
(do () (done)
;; Stage 1: Analyze uncert, characterize
;; segments.
(eval '(excl:run-shell-command
uncertanalcom))
;; Stage 2: Generate motion plan.
(multiple-value-setq (subplan done)
(build-plan assem))
;; Append subplan to end of plan generated
;; so far.
(setq result '(.@result .@subplan))
;; Update uncertainty before looping back
;; to Stage 1.
(update-uncertainty assem))
;; Return complete motion plan.
result))))

(defun temp (assem)
(multiple-value-setq (subplan done)
(build-plan assem))
(update-uncertainty assem))

;; samepos- Returns t if pose vectors l1 and l2
;; are almost
;; equal.
(defun samepos (l1 l2)
(apply 'myand (mapcar 'near l1 l2)))

;; myand- System function and will not work
;; with apply
;; function, so I've written my own.
(defun myand (&rest list) (eval '(and ,@list)))

;; near- Returns t if x and y are very nearly
;; equal.
(defun near (x y)
(< (abs (- x y)) 0.01))

;; execute- This routine writes out motion plan.
(defun execute (item)
(let ((fp (open '/home/rvl/q/motions
:direction :output :if-exists :append
:if-does-not-exist :create)))
(format fp "motion = ~s ~%" (move-motion
item))
(format fp "equation = ~s ~%" (move-equation
item))
(format fp "force-bound = ~s ~%"
(move-force-bound item))
(format fp "position-bound = ~s ~%"
(move-position-bound item))
(format fp "goal-terminating-conditions = ~s
~%"
(move-goal-terminating-conditions item))
(format fp "protectional-terminating-conditions
= ~s ~%"
(move-protectional-terminating-conditions
item))
(format fp "vector = ~s ~%" (move-vector
item))
(format fp "skip-condition = ~s ~%"
(move-skip-conditions item))
(format fp "initial-funcs = ~s ~%"
(move-initial-funcs item))
(format fp "final-funcs = ~s ~%"
(move-final-funcs item))
(format fp "compliance-frame = '(0 0 0) ~%~%")

```

```

(close fp) t))

;; setup-assem- This funct is invoked to
;; generate input to
;; Phase 1 and Phase 2 for testing either of
;; them.
;; Used for debug purposes only.
(defun setup-assem (assem)
  (setq *first* nil)
  (build-input assem))

(defun rtd (x)
  (* x 57.2957795))

(defvar *numobst* 0)
(defvar *numobj* 0)
(defvar *obstacles* nil)
(defvar rotsamp1 1)
(defvar rotsamp2 1)
(defvar *dof* 6)
;; build-input- This routine builds up input
;; files for
;; the motion planner. This includes obst and
;; object files, uncertainties and profile
;; files.
(defun build-input (assem)
  (let ((item) (result) (fp) (string) (brep) (mn)
        (mx) (center)
        (object nil) (objfp nil) (xform nil) (flag)
        (sitem) (feat)
        (id 0) (featlist nil) (apu (assembly-pose-uncert
        assem))
        (gl) (graph) (num 0) (tform) (new-tform nil)
        (ms)
        (endpos) (ptf '(0 0 0)) (mate) (feature) (fdesc)

        (starting-position (lift-pose
        (assembly-left-pose-start assem)))
        (fx (assembly-fixtured-part assem))
        (gp (assembly-grasped-part assem)))

    ;; First build list of part-lists. There
    ;; is a part-list for
    ;; every part in the assembly, which
    ;; consists of a brep,
    ;; parts pose and its uncertainty.
    (setq *dof* 6)
    (cond ((typep fx 'part)
           (setq *obstacles* (part-csg fx)))
          (t (setq *obstacles* (assembly-csg fx))))

    ;; Uncertainty object/obstacle
    (setq *numobst* 0) (setq *numobfeat* 0)
    (setq id 0)
    (dolist (item *obstacles* result)
      (setq feat (twinobj-feature item))
      (cond ((not (member feat featlist))
             (setq id (1+ id))
             (setq featlist (append '(,feat)
                                     featlist))))
      (setq result (concatenate 'string "/home
      /rvl/q/obs"
      (itoa *numobst*)))
      (setq brep (twinobj-mmc item))
      (if (null brep) (setq brep (twinobj-nom
      item)))
      (setq flag (if (equal '+
      (twinobj-operator item))
      1 -1))
      (twin-try '(writeobst ,brep ,result ,flag
      ,id))
      (setq fp (open result :direction :output
      :if-exists :append

      :if-does-not-exist :create))
      (setq ms (twinobj-movable-surfaces item))
      (cond (ms (format fp "~s ~%" (length
      ms))
             (dolist (sitem ms)
               (format fp "~s ~s ~%"
               (x sitem) (y sitem))))
             (t (format fp "0 ~%"
             (close fp)
             (setq *numobst* (1+ *numobst*)))))

      (cond ((typep gp 'articulated-part)
             (setq gl (joint-fixed
             (x (articulated-part-csg-graph gp))))
             (cond ((null (part-center gp))
                     (setq brep (part-nom gl))
                     (setq center (twin-try '(cg ,brep)))
                     (setf (part-center gp) '(0 0 0)))
                   (t (setq center (part-center gp))
                     (setf (part-center gp) '(0 0 0)))
                     (setq flag nil)

             ;; Find mating feature coordinate frame
             ;; orientation
             (do ((feature (assembly-left-features
             assem)
                   (cdr feature))
                 (mate (assembly-connection assem) (cdr
                 mate)))
                 (equal (feature-part (car feature)) gl))
               (setq feature (car feature)) (setq mate
               (car mate))
               (setq fdesc (assoc feature
               (part-components gp)))
               (if (cdr fdesc)
                   (setq ptf '(,d (cadr fdesc)) ,e (cadr
                   fdesc))
                   ,r (cadr fdesc))))
             (setq ptf '(0 0 0))
             ;; If mating feature part of onto mating,
             ;; orientation
             ;; is upside down.
             (if (typep mate 'onto)
                 ;; Z axis upside down
                 (setq ptf '(,x ptf)
                 ,(- (y ptf) 3.141592653)
                 ,z ptf))))

      ((typep gp 'part)
       ;; Find center of mass of grasped part
       (cond ((null (part-center gp))
              (setq brep (part-nom gp))
              (setq center (twin-try '(cg ,brep)))
              (setf (part-center gp) '(0 0 0))
              (t (setq center (part-center gp))
                 (setf (part-center gp) '(0 0 0)))
              (if (part-mmc gp) (setq brep (part-mmc
              gp))
                  (setq brep (part-nom gp)))

       ;; Find orient of mating feature
       ;; coordinate frame.
       (setq feature (car (assembly-left-features
       assem)))
       (setq fdesc (assoc feature
       (part-components gp)))
       (if (cdr fdesc)
           (setq ptf '(,d (cadr fdesc)) ,e (cadr
           fdesc))
           ,r (cadr fdesc))))
       (setq ptf '(0 0 0))
       (if (typep (car (assembly-connection
       assem)) 'onto)

```

```

;; Z axis upside down
(setq ptf '(x ptf)
.(- (y ptf) 3.141592653)
(z ptf))))
(t
(cond ((null (assembly-center gp))
(setq brep (assembly-nom gp))
(setq center (twin-try '(cg ,brep)))
(setf (assembly-center gp) '(0 0 0)))
(t (setq center (assembly-center gp))
(setf (assembly-center gp) '(0 0 0))))
(if (assembly-mmc gp) (setq brep
(assembly-mmc gp))
(assembly-nom gp)))
(setq brep (assembly-nom gp)))

;; Find orient of mating feature
;; coordinate frame.
(setq feature (car (assembly-left-features
assem)))
(setq ptf (get-position gp feature))
(setq ptf (cddr ptf))
(if (typep (assembly-connection assem)
'onto)
;; Z axis upside down
(setq ptf '(x ptf)
.(- (y ptf) 3.141592653)
(z ptf))))

(cond ((typep gp 'articulated-part)
(dolist (item (articulated-part-csg-graph
gp))
(setq tform (joint-transform item))
(setq new-tform (invert-xform (z
tform)))
(setq new-tform
(fix-transform-new-tform center))
(setq new-tform (invert-xform
new-tform))
(setq tform '(.(fix-transform (x tform)
center)
(y tform) ,new-tform))
(setf (joint-transform item) tform))

;; Transform part coordinate frame to
;; center of mass,
;; then align with mating feature
;; coordinate frame.
(setq endpos (lift-pose
(x (assembly-left-pose-end assem))))
(setq endpos '(.(+ (x endpos) (x center))
.+(y endpos) (y center))
.+(z endpos) (z center))
.-(d endpos) (x ptf))
.-(e endpos) (y ptf))
.-(r endpos) (z ptf)))
(setq starting-position
'(.+(x starting-position) (x center))
.+(y starting-position) (y center))
.+(z starting-position) (z center))
.-(d starting-position) (x ptf))
.-(e starting-position) (y ptf))
.-(r starting-position) (z ptf))))
(setq graph (articulated-part-mmc-graph
gp))

(if (null graph)
(setq graph (articulated-part-nom-graph
gp)))

; Write out mmc reps
(setq num 0)
(setq brep (x (x graph)))
(twin-try '(xform .(- (x center)) .(- (y
center))

```

```

.(- (z center)) 0 0 0 ,brep))
(twin-try '(xform 0 0 0 ,(x ptf) ,(y ptf)
(z ptf)
,brep))
(twin-try '(writeobj ,brep /home/rvl/q
/mmc0 0))
(twin-try '(xform 0 0 0
.(- (x ptf)) .(- (y ptf))
.(- (z ptf)) ,brep))

(dolist (item graph)
(setq brep (z item))
(twin-try '(xform .(- (x center))
.(- (y center))
.(- (z center))
0 0 0 ,brep))
(twin-try '(xform 0 0 0
(x ptf) ,(y ptf)
(z ptf) ,brep))
(writexform (y item) num)
(setq endpos
(append endpos '(.(joint-final-position
(y item))))))
(setq starting-position
(append starting-position
'(.joint-initial-position
(y item))))))

;; Now create object file.
(setq num (1+ num))
(twin-try '(writeobj ,brep
.(concatenate 'string
"/home/rvl/q/mmc" (itoa num)) 0))
(twin-try '(xform 0 0 0 .(- (x ptf))
.(- (y ptf))
.(- (z ptf)) ,brep)))

(setq *dof* (+ *dof* num))
;; Build calling string and invoke twincon
;; routine
;; to convert brep files to planner input
;; files.
(setq string (concatenate 'string
"twincon articulated "
(itoa *numobst*)))
(dotimes (item *numobst*)
(setq string (concatenate 'string
string
"/home/rvl/q/obs"
(itoa item))))
(setq string (concatenate 'string
string
"/home/rvl/q/research/run/worldbrep"))
(setq num (1+ num))
(dotimes (item num)
(setq string
(concatenate 'string
string "/home/rvl/q/mmc" (itoa item)))
(if (< item (- num 1))
(setq string (concatenate 'string
string
"/home/rvl/q/xform"
(itoa item)))))
(t
;; Transform object so center of mass is
;; at (0 0 0)
(twin-try '(xform .(- (x center))
.(- (y center)) .(- (z center))
0 0 0 ,brep))
;; Transform object so that Z axis of
;; object
;; is in line with Z axis of mating
;; features.

```

```

(twin-try '(xform 0 0 0 ,(x ptf) ,(y ptf)
            ,(z ptf) ,brep))
(setq endpos (lift-pose
(x (assembly-left-pose-end assem))))
(setq endpos '(. (+ (x endpos) (x center))
                (+ (y endpos) (y center))
                (+ (z endpos) (z center))
                (- (d endpos) (x ptf))
                (- (e endpos) (y ptf))
                (- (r endpos) (z ptf))))
(setq starting-position
'(. (+ (x starting-position) (x center))
    (+ (y starting-position) (y center))
    (+ (z starting-position) (z center))
    (- (d starting-position) (x ptf))
    (- (e starting-position) (y ptf))
    (- (r starting-position) (z ptf))))
;; Now create object file.
;; Move coordinate frame to center of mass
;; of part.
(twin-try '(writeobj ,brep /home/rvl/q/mmc
0))
;; Xform back in case the object will be
;; used again
(twin-try '(xform 0 0 0 ,(- 0 (x ptf)) ,(-
0 (y ptf)) ,(- 0 (z ptf)) ,brep))

;; Build calling string and invoke twincon
;; routine
;; to convert brep files to planner input
;; files.
(setq string (concatenate 'string
"twincon rigid "
(itoa *numobst*)))
(dotimes (item *numobst*)
(setq string
(concatenate 'string string
(concatenate 'string
"/home/rvl/q/obs"
(itoa item)))))
(setq string
(concatenate
'string string
"/home/rvl/q/research/run/worldbrep
/home/rvl/q/mmc"))
(eval '(excl:run-shell-command ,string))
(excl:run-shell-command
"cp /home/rvl/q/obstacles /home/rvl/q
/uncobstacles")
(excl:run-shell-command
"cp /home/rvl/q/object /home/rvl/q/uncobject")

;;;;;; End of mmc reps ;;;;;;;;;;

(setq *numobst* 0) (setq *numobfeat* 0)
(setq id 0)
(setq featlist nil)
(dolist (item *obstacles* result)
(setq feat (twinobj-feature item))
(cond ((not (member feat featlist))
(setq id (1+ id))
(setq featlist (append '(. feat)
featlist))))
(cond ((twinobj-mmc item)
(setq result
(concatenate 'string "/home/rvl/q/obs"
(itoa *numobst*)))
(setq brep (twinobj-nom item))
(setq flag
(if (equal '+ (twinobj-operator item))
1 -1))

```

```

(twin-try
'(writeobj ,brep ,result ,flag ,id))
(setq fp (open result :direction :output
:if-exists :append
:if-does-not-exist :create))
(setq ms (twinobj-movable-surfaces
item))
(cond (ms (format fp "~s ~%" (length
ms)))
(dolist (sitem ms)
(format fp "~s ~s ~%"
(x sitem)
(y sitem))))
(t (format fp "0 ~%" ""))
(close fp))
(setq *numobst* (1+ *numobst*))

(cond ((typep gp 'articulated-part)
(cond ((articulated-part-mmc-graph gp)
(setq graph (articulated-part-nom-graph
gp))
(setq brep (x (x graph)))
(twin-try '(xform ,(- (x center))
,(- (y center))
,(- (z center)) 0 0 0 ,brep))
(twin-try '(writeobj ,brep /home
/rvl/q/obj0 0))
(dolist (item graph)
(setq brep (z item))
(twin-try '(xform ,(- (x center))
,(- (y center))
,(- (z center))
0 0 0 ,brep))

;; Now create object file.
(setq num (1+ num))
(twin-try
'(writeobj ,brep
(concatenate 'string
"/home/rvl/q/obj"
(itoa num)) 0)))
(setq num (1+ num))
(setq string (concatenate 'string
"twincon articulated "
(itoa *numobst*)))
(dotimes (item *numobst*)
(setq string (concatenate 'string
string
"/home/rvl/q/obs"
(itoa item)))))
(setq string (concatenate
'string string
"/home/rvl/q/research/run/worldbrep"))
(dotimes (item num)
(setq string
(concatenate 'string
string
"/home/rvl/q/obj"
(itoa item))
(if (< item (- num 1))
(setq string
(concatenate 'string
string
"/home/rvl/q/xform"
(itoa item)))))
(eval '(excl:run-shell-command ,string)))
(t (dotimes (item num)
(setq string
(concatenate 'string
"cp /home/rvl/q/mmc"
(itoa item)
"/home/rvl/q/obj"
(itoa item))

```



```

(eval '(excl:run-shell-command
string))))))
(t
(setq brep nil)
(cond ((typep gp 'part)
(cond ((part-mmcc gp)
(setq brep (part-nom gp))
(setq center (part-center gp))))))
(t
(cond ((assembly-mmcc gp)
(setq brep (assembly-nom gp))
(setq center (assembly-center gp))))))

;; Now create object file.
;; Move coordinate frame to center of
;; mass of part.
;; Align part coordinate frame with
;; mating features.
(cond (brep
(twin-try '(xform ,(- (x center))
,(- (y center))
,(- (z center))
0 0 0 ,brep))
(twin-try '(xform 0 0 0 ,(x ptf) ,(y
ptf)
,(z ptf) ,brep))
(twin-try
'writeobj ,brep /home/rvl/q/obj 0))

;; Restore original part coordinate
;; frame.
(twin-try '(xform 0 0 0 ,(- (x ptf))
,(- (y ptf))
,(- (z ptf)) ,brep))
(t (excl:run-shell-command
"cp /home/rvl/q/mmcc /home/rvl/q/obj"))

;; Build calling string and invoke
;; twincon
;; to convert brep files to planner input
;; files.
(setq string (concatenate 'string
"twincon rigid "
(itoa *numobst*)))
(dotimes (item *numobst*)
(setq string
(concatenate 'string string
"/home/rvl/q/obs"
(itoa item))))

(setq string
(concatenate
'string string
"/home/rvl/q/research/run/worldbrep
/home/rvl/q/obj"))
(eval '(excl:run-shell-command
string))))

(setq fp (open "/home/rvl/q
/uncertainties"
:direction :output
:if-exists :overwrite
:if-does-not-exist :create))
;; Record obstacle part uncertainty
(dolist (item *obstacles*)
(setq center (part-uncert
(feature-part (twinobj-feature item))))
(if (null center) (setq center '(0 0 0
0 0)))
(if (not (listp center))
(setq center '(0 0 0 0 0 0)))
(if (or (> (d center) 0) (> (e center) 0)
(> (r center) 0)) (setq rotsamp2 3))

(format fp "~s ~s ~s ~s ~s ~s" (x
center)
(y center)
(z center) (d center) (e center) (r
center))
(setq result (twinobj-center item))
(if (null result)
(setq result (self (twinobj-center item)
(twin-try
'cg
(twinobj-nom item))))))
(if (not (listp result)) (setq result '(0
0 0)))
(format fp "~s ~s ~s ~s" (x result) (y
result)
(z result)))
(dotimes (item 6)
(format fp "0 0 0 0 0 0 0 0 ~s"))

;; Record object pose and motion uncertainty
(format fp "~s ~s ~s ~s ~s ~s" (x apu)
(y apu) (z apu) (d apu) (e apu) (r apu))
(cond ((typep gp articulated-part)
(dolist (item (articulated-part-csg-graph
gp))
(if (> (joint-initial-uncertainty item)
0)
(setq rotsamp1 3))
(format fp "~s"
(joint-initial-uncertainty item))))))
(format fp "~s")
(if (or (> (d apu) 0) (> (e apu) 0) (> (r apu)
0))
(setq rotsamp1 3))
(format fp "~s ~s ~s ~s ~s ~s"
(x (assembly-control-uncert assem))
(y (assembly-control-uncert assem))
(z (assembly-control-uncert assem))
(d (assembly-control-uncert assem))
(e (assembly-control-uncert assem))
(r (assembly-control-uncert assem)))
(format fp "~s ~s ~s" rotsamp1 rotsamp2)
(close fp))

;; Read min and max extents calculated by
;; twincon
(setq fp (open "/home/rvl/q/mm"
:direction :input))
(setq mn (read fp)) (setq mx (read fp))
(close fp)

;; Now build profile file.
(setq fp (open "profile" :direction
:output
:if-exists :overwrite
:if-does-not-exist :create))
(format fp "~s ~s" *dof*) ;; Dimension
of problem
(dolist (item endpos)
(format fp "~s" item))
(format fp "~s")
(dolist (item starting-position)
(format fp "~s" item))
(format fp "~s")
(format fp "~s ~s ~s" mn mx)
(format fp "1 ~s")
(close fp))

;; itoa- Converts an integer to a string
(defun itoa (num)
(cond ((equal num 0) "0")
((equal num 1) "1")
((equal num 2) "2")

```

```

(equal num 3) "3")
(equal num 4) "4")
(equal num 5) "5")
(equal num 6) "6")
(equal num 7) "7")
(equal num 8) "8")
(equal num 9) "9")
(t (concatenate 'string (itoa (truncate num
10)))
(itoa (mod num 10))))))

;; fix-transform- Changes position of coordinate
;; matrix
;; to reflect a move by the amount specified.
(defun fix-transform (matrix move)
  (let ((pose))
    (setq pose (lift-pose matrix))
    (setq pose '(- (x pose) (x move))
      (- (y pose) (y move))
      (- (z pose) (z move)) ,@(cddr pose)))
    (make-xform pose)))

;; writexform- Dumps info needed to describe xform
;; feasible at a joint.
(defun writexform (joint num)
  (let ((ltof) (offset) (ftol) (xform)
    (trans (joint-transform joint)) (fp))
    (setq ltof (x trans))
    (setq offset (y trans))
    (setq ftol (lift-pose (z trans)))
    (setq ltof (lift-pose (mmult ltof
      offset)))
    (setq fp (open (concatenate 'string
      "/home/rvl/q/xform" (itoa num))
      :direction :output :if-exists :overwrite
      :if-does-not-exist :create))
    (format fp "~s ~s ~s ~s ~s ~s" (x
      ltof) (y ltof)
      (z ltof) (d ltof) (e ltof) (r ltof))
    (format fp "~s ~s ~s ~s ~s ~s" (x
      ftol) (y ftol)
      (z ftol) (d ftol) (e ftol) (r ftol))
    (cond ((typep joint 'revolute)
      (format fp "revolute ~s")
      (t (format fp "prismatic ~s")))
    (t (format fp "prismatic ~s")))
    (close fp)))

;; This file contains code for Phase 2 Stage 2,
;; where final
;; motion plan is developed from output of Stage
;; 1. A little
;; additional massaging of Stage 1 data from must
;; be performed
;; to get a complete state description for each
;; path segment.
;;
;; Define global variables
(defvar *first* nil)
(defvar *statelist* nil)
(defvar uncertainties nil)
(defvar constraints nil)
(defvar misc nil)

;; Build-plan- Reads in statelist from uncertainty
;; analysis &
;; builds a plan from the statelist.
;; Returns plan.
(defun build-plan (assem)
  (let ((plan nil) (done) (strategy) (start))
    ;; Read list of states that we must plan
    ;; for
    (setq done (readstates assem))

    ;; Motions are all relative to initial
    ;; point
    ;; Control uncertainty calculated relative
    ;; to first point
    ;; on statelist
    ;; Must keep track of each of these
    ;; vectors.
    (setq start (caar *statelist*))
    (if (null *first*) (setq *first* start))

    ;; For each state, find strategy to use,
    ;; and then employ
    ;; selected strategy. Append subplan found
    ;; onto end
    ;; of plan. Already at first point, so skip
    ;; first state.
    (do ((state *statelist* (cdr state)))
      ((null (cdr state)) plan)
      (let ((strategy
        (find-strategy (cadr state) *library*)))
        (setq plan
          '(@plan ,@(employ strategy start (car
            state)
            (cadr state) assem))))
        (values plan done)))

    ;; find-strategy- Looks through library of
    ;; strategies and
    ;; compares the precondition of each strategy
    ;; to
    ;; state. Returns the first strategy whose
    ;; preconditions are met by the state.
    (defun find-strategy (state library)
      (let ((selected nil))
        ;; Look at each strategy in the library
        (do ((strategy library (cdr strategy)))
          ((or selected (null strategy)) selected)
          ;; As soon as one has been selected,
          ;; exit loop and return strategy
          ;; If preconditions met, select this strategy
          (if (meet-preconditions state (caar
            strategy))
            (setq selected (car strategy))))))

    ;; meet-preconditions- Tests to see whether or not
    ;; the
    ;; preconditions are met
    (defun meet-preconditions (state preconds)
      ;; Pull out different preconditions
      (let ((uncert-cond (first preconds))
        (constraint-cond (second preconds))
        (misc-cond (third preconds)))
        (setq uncertainties (second state))
        (setq constraints (third state))
        (setq misc (fourth state))
        ;; Test preconditions
        (if (null uncert-cond) (setq uncert-cond
          t))
        (if (null constraint-cond) (setq
          constraint-cond t))
        (if (null misc-cond) (setq misc-cond t))
        ;; If all three met return t
        (and (eval uncert-cond) (eval
          constraint-cond)
          (eval misc-cond))))

    ;; employ- Employs the given strategy to formulate
    ;; a
    ;; subplan for the state.
    (defun employ (strategy start lstate cstate assem)
      ;; Pull out motion function and post condition
      ;; function
      (let ((movefunc (second strategy))

```

```

(postcondfunc (third strategy)) (subplan))
;; Develop motion subplan using motion function
(setq subplan (eval '(movefunc 'start '
    ,lstate
    ,cstate ,assem)))
;; Evaluate post condition function.
(if (postcondfunc (eval '(postcondfunc '
    ,start
    ,cstate ,assem)))
    subplan))

;; readstates- This function reads list of states
;; in from
;; uncertainty analysis. Each state is
;; list of
;; 1. ending position and orient of
;; grasped part
;; 2. uncertainties that must be overcome
;; 3. constraints on the grasped part
;; 4. Infor needed to form fine-motion
;; subplan
;; if necessary.
;; Returns t if no more states, nil if
;; some states
;; not yet tagged by uncertainty analysis.
(defun readstates (assem)
  ;; Open file containing output of Phase 2 Stage
  ;; 1.
  (let ((fp (open '/home/rvl/q/strategy :direction
    :input))
        (nstates) (point) (nuncert)
        (constraints)
        (value) (done) (state) (uncert))
    (setq *statelist* nil)
    ;; Read in number of states
    (setq nstates (read fp))
    ;; For each state
    (dotimes (i nstates)
      ;; Read in *dof* components of endpoint.
      ;; (Starting point is assumed to be endpoint
      ;; of last segment).
      (setq point nil)
      (dotimes (j *dof*)
        (setq point '(@point ,(read fp))))
      (setq nuncert (read fp))
      (setq uncert nil)
      ;; Read in nuncert components of uncertainty
      ;; This is unc_red_axis info.
      (dotimes (j nuncert)
        (setq uncert '(@uncert ,(read fp))))
      ;; Read in constraints.
      (setq constraints nil)
      (dolist (axis '(X +X -X) (Y +Y -Y) (Z +Z -Z))
        (Rotx) (Roty) (Rotz)))
      (setq value (read fp))
      (cond ((equal value 1)
        (setq constraints
          '(@constraints
            ,first axis))))
        ((equal value 2)
          (setq constraints
            '(@constraints
              ,second axis))))
        ((equal value 3)
          (setq constraints
            '(@constraints
              ,third axis))))))
    ;; Decide if at end of state list because
    ;; entire
    ;; path has been analyzed or if
    ;; collision-prone
    ;; segment found.
    (cond ((equal i (- nstates 1))

```

```

      (if (or (< nuncert 1) (equal uncert
        'Z)))
        (setq done t) (setq done nil))
        ;; If done, don't need rest of the info
        (setq start (caar *statelist*))
        (if done (setq features nil)
          ;; Finding value of misc attribute
          ;; is a little involved. Takes
          ;; some interfacing with assembly
          ;; representation system.
          (setq features
            (get-clearance-features
              fp start point assem)))
          (setq *statelist* '(@*statelist*
            ,point ,uncert
            ,constraints
            ,features)))
          (t (setq *statelist*
            '(@*statelist*
              ,point ,uncert ,constraints))))))
        ;; Return t if done, nil otherwise
        done))

;; get-clearance-features- This function finds
;; value
;; of the misc attribute knowing the
;; obstacles
;; involved in the collision. The features
;; found are put on clearlist, and will be
;; used when the uncertainty is updated.
(defun get-clearance-features (fp start end assem)
  (let ((obstacle) (obslist nil) (item) (motion)
        (point) (i)
        (shape 'peg) (featobs nil) (feat) (clearlist
          nil)
        (algs (assembly-left-features assem)) (i)
        (twinpart)
        (arfs (assembly-right-features assem))
        (acu (assembly-control-uncert assem))
        (apu (assembly-pose-uncert assem)))
    ;; First find the current uncertainty
    ;; generically
    ;; referred to as the clearance.
    (setq *clearance* '(+ (x apu)
      (* (abs (- (x start) (x end)))
        (x acu)))
      (+ (y apu)
        (* (abs (- (y start) (y end)))
          (y acu)))
      (+ (z apu)
        (* (abs (- (z start) (z end)))
          (z acu)))
      (+ (d apu)
        (* (abs (- (d start) (d end)))
          (d acu)))
      (+ (e apu)
        (* (abs (- (e start) (e end)))
          (e acu)))
      (+ (r apu)
        (* (abs (- (r start) (r end)))
          (r acu))))))
    ;; Read in RCC Flag
    (setq rcc (read fp))
    ;; Read in all of the obstacles involved in
    ;; the collision
    (setq obstacle 0)
    (do () ((equal obstacle -1))
      (setq obstacle (read fp))
      (if (not (equal -1 obstacle))
        (setq obslist (append '(@obstacle)
          obslist))))))

```

```

;; Read in other information about state
(setq motion '(, (read fp) , (read fp) , (read
fp)))
(setq point '(, (read fp) , (read fp) , (read
fp) , (read fp)
, (read fp) , (read fp)))
(close fp)

```

```

;; Create list of lists, each list in the
;; list is the
;; clearance feature on the fixtured part
;; and the matching
;; feature on the grasped part. Also
;; figure out shape.
(dolist (item obslist)
  (setq twinpart (nth item *obstacles*))
  (setq feat (twinobj-feature twinpart))

```

```

;; Sometimes feature is actually block in
;; which hole is contained. Don't care
;; about
;; block, find which hole.
(cond ((and (not (member feat arfs))
  (probably-hole item alfs))
  (setq feat (which-hole
  feat alfs arfs assem point)))
  (t (setq feat '(, feat))))

```

```

;; Collect up list of all features, but
;; do not put
;; on list twice.
(dolist (sitem feat)
  (if (not (member sitem featobs))
    (setq featobs '(, @featobs , sitem))))

```

```

;; Find clearance involved in failed
;; operation.
;; Also determine if the mating feature of
;; grasped part
;; is convex or concave. If convex type
;; =peg, else
;; type=hole. If its not clear, its
;; unimportant.
;; (This information is only used for a few
;; strategies).
;; Note- Method for finding colliding
;; feature on grasped
;; part is not quite same as
;; explained in thesis.
(dolist (item featobs)
  (cond ((member item arfs)
    (setq feat (nth (which-feature item
    arfs)
    alfs)))
    (t (setq feat nil)))
  (cond ((or (typep item 'peg) (typep item
  'nut)
  (typep item 'bolt))
    (setq shape 'hole)
    (if (null feat)
      (setq feat
  (x (which-hole item alfs arfs
  assem point))))
    (setq clearlist
  '(, @clearlist , (item , feat))))
  ((typep item 'gear)
    (setq shape 'peg)
    (setq clearlist
  '(, @clearlist , (item , feat))))
  ((typep item 'hole)
    (setq shape 'peg)
    (if (null feat)
      (setq feat
  (x (which-hole item alfs arfs
  assem point))))
    (setq clearlist
  '(, @clearlist , (item , feat))))
  (t (setq feat
  (x (which-hole item alfs arfs
  assem point))))
  (setq clearlist
  '(, @clearlist , (item , feat))))

```

```

(which-peg point item alfs)))
(setq clearlist
  '(, @clearlist , (item , feat))))
;; Now update *clearance* based on
;; clearances
;; between interacting features.
(find-clearance clearlist)
'(, clearlist , shape , motion , point , rcc)))

```

```

;; find-clearance- Finds tightest clearance
;; between
;; interacting features. Clearances are
;; calculated
;; on a class by class basis.
(defun find-clearance (clearlist)
  (dolist (item clearlist)
    (cond ((or (typep (x item) 'peg)
      (typep (x item) 'nut)
      (typep (x item) 'bolt))
      (peg-clearance (x item) (y item)))
      ((typep (x item) 'hole)
      (peg-clearance (y item) (x item)))
      ((typep (x item) 'gear)
      (gear-clearance (y item) (x item)))))

```

```

;; gear-clearance- Finds clearance between 2
;; meshing gears.
(defun gear-clearance (g1 g2)
  (let ((t1) (t2) (x1) (rotz1) (x2) (rotz2) (rad)
    (x) (y) (rotz))
    (setq t1 (spur-gear-x1 g1))
    (setq t2 (- (tooth-in-offset g2)
      (spur-gear-x2 g2)))
    (setq x1 (- t2 t1))
    (setq rad (sqrt (- (sqr (gear-radius g1))
      (sqr x1))))
    (setq rotz1 (atan x1 rad))
    (setq t1 (spur-gear-x1 g2))
    (setq t2 (- (tooth-in-offset g1)
      (spur-gear-x2 g1)))
    (setq x2 (- t2 t1))
    (setq rad (sqrt (- (sqr (gear-radius g1))
      (sqr x2))))
    (setq rotz2 (atan x2 rad))
    (if (< x1 0) (setq x1 0))
    (if (< x2 0) (setq x2 0))
    (if (< rotz1 0) (setq rotz1 0))
    (if (< rotz2 0) (setq rotz2 0))

```

```

;; Take min x clearance value from all
;; constraints
(setq x (x *clearance*))
(if (< x1 x) (setq x x1))
(if (< x2 x) (setq x x2))

```

```

;; Take min y clearance value from all
;; constraints
(setq y (y *clearance*))
(if (< x1 y) (setq y x1))
(if (< x2 y) (setq y x2))

```

```

;; Take min rotz clearance value from all
;; constraints
(setq rotz (r *clearance*))
(setq rotz 0)
(if (< rotz1 rotz) (setq rotz rotz1))
(if (< rotz2 rotz) (setq rotz rotz2))

```

```

(setq *clearance* '(, x , y , (z *clearance*)
  , (d *clearance*)
  , (e *clearance*) , rotz)))

```

```

;; tooth-in-offset- Find offset between teeth.

```

```

(defun tooth-in-offset (gear)
  (let ((rad (- (gear-radius gear) (spur-gear-y
    gear))))
    (nt (gear-num-teeth gear)) (chunk))
    (setq chunk (/ (* 6.2831853 rad) nt))
    (- chunk (spur-gear-x2 gear))))

;; peg-clearance- Find clearance between generic
;; peg and hole.
(defun peg-clearance (peg hole)
  (let ((clearance) (length) (x (* *clearance*))
    (y (* *clearance*))
    (rx (* *clearance*)) (ry (* *clearance*))
    (rz (* *clearance*))
    (thetaxy) (thetaz) (xl) (yl) (item) (hrad)
    (prad))
    ;; Find translational clearance between peg
    ;; and hole and
    ;; find length of peg to be used for
    ;; rotational clearance.
    (setq prad (calculate-peg-radius peg))
    (setq hrad (calculate-hole-radius hole))
    (setq length (calculate-peg-height peg))
    (setq clearance (- hrad prad))
    (if (< clearance 0) (setq clearance 0))

    ;; Calculate rotational clearance
    (setq clearance (* 0.1 clearance))
    (setq thetaxy (atan clearance length))
    ;; Let twin interface routine calculate max
    ;; extent
    ;; or "length" of an arbitrary peg.
    (cond ((typep peg 'arb-peg)
      (setq item (twin-try '(get-max-extents
        (feature-nom peg))))
      (setq thetaz (twin-try '(get-max-extents
        (feature-nom hole))))
      (setq xl (- (x rotz) (x item)))
      (setq yl (- (y rotz) (y item)))
      (setq thetaz
        (get-rotz-clear xl yl (x item) (y item))))

      ((typep peg 'nut) (setq thetaz 0))
      ;; Assume clearance very large otherwise
      (t (setq thetaz 10000)))

    ;; Update clearance if new clearance is
    ;; smaller
    (if (< clearance x) (setq x 0)) ;;
    (clearance))
    (if (< clearance y) (setq y 0)) ;;
    (clearance))
    (if (< thetaxy rx) (setq rx thetaxy))
    (if (< thetaxy ry) (setq ry thetaxy))
    (if (< thetaz rz) (setq rz thetaz))

    ;; Record new (or possibly old) clearances
    (setq *clearance*
      '(x,y,(z *clearance*),rx,ry,rz)))

;; get-rotz-clear- Calculate clearance w.r.t. rotz
;; axis.
(defun get-rotz-clear (x y lx ly)
  (let ((adjacent) (ox (+ x lx)) (oy (+ y ly)))
    (if (> oy ox)
      (setq adjacent ox)
      (setq adjacent oy))
    (- (acos (/ adjacent (sqrt (+ (sqr ox) (sqr
      oy)))))
      (acos (/ adjacent (sqrt (+ (sqr x) (sqr
        y)))))))

;; calculate-peg-height- Generic function for
;; calculating
;; height of a peg feature. Height is length
;; along
;; z axis.
(defgeneric calculate-peg-height (peg feature))

;; calculate-peg-height- Method used for
;; round-pegs
(defmethod calculate-peg-height ((peg round-peg))
  (+ (round-peg-cheight peg) (round-peg-bheight
    peg)))

;; calculate-peg-height- Method for comp-round-peg
(defmethod calculate-peg-height
  ((peg composite-round-peg))
  (let ((height 0))
    ;; Add in height of each component of
    ;; feature.
    (dolist (item (composite-round-peg-parts
      peg))
      (setq height (+ height (y item)))))

  ;; calculate-peg-height- Method used for
  ;; nsided-pegs
  (defmethod calculate-peg-height ((peg nsided-peg))
    (nsided-peg-cheight peg))

  ;; calculate-peg-height- Method used for
  ;; threaded-pegs
  (defmethod calculate-peg-height ((peg
    threaded-peg))
    (+ (threaded-peg-height peg)
      (threaded-peg-thread-height peg)))

  ;; calculate-peg-height- Method used for bolt
  (defmethod calculate-peg-height ((peg bolt))
    (+ (bolt-cheight peg) (bolt-bheight peg)))

  ;; calculate-peg-height- Method used for nut
  (defmethod calculate-peg-height ((peg nut))
    (nut-height peg))

  ;; calculate-peg-height- Method used for arb-peg.
  (defmethod calculate-peg-height ((peg arb-peg))
    (twin-try '(max-z-extent (feature-nom peg))))

;; calculate-peg-radius- Generic funct for
;; calculating
;; radius of a peg feature.
(defgeneric calculate-peg-radius (peg feature))

(defmethod calculate-peg-radius ((peg round-peg))
  (round-peg-radius peg))

(defmethod calculate-peg-radius
  ((peg composite-round-peg))
  (let ((radius 0)) ;; Look at each part and
    choose largest
    (dolist (item (composite-round-peg-parts
      peg))
      (if (> (z item) radius)
        (setq radius (z item)))
      (if (and (d item) (> (d item) radius))
        (setq radius (d item)))))

  (defmethod calculate-peg-radius ((peg nsided-peg))
    (nsided-peg-radius peg))

  (defmethod calculate-peg-radius ((peg
    threaded-peg))
    (+ (threaded-peg-radius peg)
      (threaded-peg-thread-depth peg)))

```

```

(defmethod calculate-peg-radius ((peg bolt))
  (+ (bolt-radius peg) (bolt-thread-depth peg)))

(defmethod calculate-peg-radius ((peg nut))
  (nut-radius peg))

(defmethod calculate-peg-radius ((peg arb-peg))
  (ourmax (twin-try '(max-x-extent ,(feature-nom
    peg)))
    (twin-try '(max-y-extent ,(feature-nom
    peg)))))

;; calculate-hole-radius- Generic function for
;; calculating radius
;; of a hole feature.
(defgeneric calculate-hole-radius (hole feature))

(defmethod calculate-hole-radius ((hole
  round-hole))
  (round-hole-radius hole))

(defmethod calculate-hole-radius ((hole
  composite-round-hole))
  (let ((radius 0))
    ;; Look at each part and choose largest
    ;; radius
    (dolist (item (composite-round-hole-parts
    hole))
      (if (> (z item) radius)
        (setq radius (z item)))
      (if (and (d item) (> (d item) radius))
        (setq radius (d item)))))

(defmethod calculate-hole-radius ((hole
  nsided-hole))
  (nsided-hole-radius hole))

(defmethod calculate-hole-radius ((hole
  threaded-hole))
  (threaded-hole-radius hole))

(defmethod calculate-hole-radius ((hole nut-hole))
  (nut-hole-radius hole))

(defmethod calculate-hole-radius ((hole arb-hole))
  (ourmax (twin-try '(max-x-extent ,(feature-nom
    hole)))
    (twin-try '(max-y-extent ,(feature-nom
    hole)))))

;; ourmax- Returns max(x,y).
(defun ourmax (x y) (if (> x y) x y))

;; which-feature- Returns n if feature is the nth
;; entry in list.
(defun which-feature (feature list)
  (let ((result 0) (answer))
    (dolist (item list)
      (if (equal item feature) (setq answer
        result))
      (setq result (+ 1 result)))
    answer))

;; tightest-clearance- Update global var
;; *clearance*
;; if necessary.
(defun tightest-clearance ()
  (let ((temp))
    (cond ((y *clearance*)
      (setq temp (x *clearance*))
      (dolist (item (cdr *clearance*))
        (if (< (x item) (x temp))

```

```

      (setq temp '(x item) ,@(cdr temp))))
    (if (< (y item) (y temp))
      (setq temp '(x temp) ,@(y item)
        ,@(z temp))))
    (if (or (null (z temp))
      (and (z item) (< (z item) (z temp))))
      (setq temp '(x temp) ,@(y temp)
        ,@(z item))))
    (setq *clearance* temp))
    (t (setq *clearance* (x *clearance*)))))

;; probably-hole- We are trying to determine if
;; clearance
;; feature on fixtured-part that we are
;; interested in
;; is a hole by examining features on the grasped
;; part.
(defun probably-hole (item list)
  (let ((flag nil))
    (dolist (item list)
      (if (> (feature-sign item) 0) (setq flag
        t)))
    flag))

;; which-hole- Try to find which hole on the
;; fixtured
;; part corresponds to the clearance
;; feature of
;; interest.
(defun which-hole (obst grasped fixtured assem
  point)
  (let ((part (feature-part obst)) (features)
    (flag t)
    (possibles nil) (com) (closest 1000) (distance))
    ;; Find features following obstacle feature
    ;; on part,
    ;; holes in a feature always follow it.
    (setq features (after-features obst
    (part-components part)))

    ;; Collect up list of holes in obstacle
    ;; feature
    (dolist (item features)
      (if (and flag (< (feature-sign item) 0))
        (setq possibles '(.@possibles ,item))
        (setq flag nil)))

    ;; If more than one possibility, decide
    ;; between them
    (cond ((caddr possibles)
      (setq features possibles) (setq possibles
        nil)
      (dolist (item features)
        (setq com (twin-try
          'cg ,(feature-nom item))))
      (setq distance
        (+ (sqr (- (x com) (x point)))
          (sqr (- (y com) (y point)))
          (sqr (- (z com) (z point)))))
        (if (< distance closest)
          (setq closest distance)))
      (dolist (item features)
        (cond ((member item fixtured)
          (setq possibles '(.@possibles ,item)))
          (t
            (setq com (twin-try
              'cg ,(feature-nom item))))
          (setq distance
            (+ (sqr (- (x com)
              (x point)))
              (sqr (- (y com)
              (y point)))
              (sqr (- (z com)
              (z point)))))

```

```

(z point)))))
(if (< (abs (- com distance)) 0.1)
  (setq possibles
    '(@possibles ,item))))))
possibles))

;; which-peg- We are trying to find which peg
;; feature is
;; clearance feature of interest.
(defun which-peg (point item list)
  (let ((com) (distance) (closest 1000)
        (tform (make-xform point)) (location) (peg))
    (dolist (item list)
      (setq com (twin-try '(cg ,(feature-nom
        item))))
      (setq location (mvmult tform '(@com 1)))
      (setq distance
        (+ (sqr (- (x location) (x point)))
           (sqr (- (y location) (y point)))
           (sqr (- (z location) (z point)))))
      (cond ((and (< distance closest)
                  (equal (feature-sign item) 1))
              (setq closest distance)
              (setq peg item))))
    peg))

(defun after-features (obst list)
  (cond ((null list) nil)
        ((equal obst (caar list))
         (just-first (cdr list)))
        (t (after-features obst (cdr list)))))

(defun just-first (list)
  (let ((newlist nil))
    (dolist (item list)
      (if newlist
          (setq newlist '(@newlist ,(x item)))
          (setq newlist '(@newlist ,item))))
    newlist))

;; update-uncertainty- This routine updates
;; the uncertainty
;; model contained in the file uncertainties
;; after
;; a fine-motion has been executed.
(defun update-uncertainty (assem)
  (let ((fp (open '/home/rvl/q/uncertainties
                  :direction :output
                  :if-exists :overwrite
                  :if-does-not-exist :create))
        (apu (assembly-pose-uncert assem))
        (acu (assembly-control-uncert assem))
        (uncert) (item) (obstacles) (result)
        (fx (assembly-fixtured-part assem)))
    ;; First build list of obstacle-lists. The
    ;; obstacles are the
    ;; PARTS making up the fixtured part or
    ;; subassembly.
    ;; Now recreate the uncertainties file.
    (dolist (item *obstacles* result)
      ;; Record obstacle part uncertainty
      (setq uncert (part-uncert (feature-part
        (twinobj-feature item))))
      (if (null uncert) (setq uncert '(0 0 0 0
        0)))
      (format fp "~s ~s ~s ~s ~s ~%"
        (x uncert) (y uncert)
        (z uncert) (d uncert) (e uncert) (r
        uncert))
      (if (or (> (d uncert) 0) (> (e uncert) 0)
          (> (r uncert) 0))
          (setq rotsamp2 3))
    )
    (close fp) (close fpout t)))

(defvar *library* nil)

;;
;; library- This file contains library of
;; available actions
;; that can be used by our motion planner.
;;
(setq *library*
  (
    ;; List of available strategies
    (
      ;; Gross motion thru free space
      (
        ;; Preconditions

```

```

(null uncertainties) ;; uncert precondition
(null constraints)) ;; const precondition
moveto) ;; Move func

( ;; Compliant motion thru free space
( ;; Preconditions
(null uncertainties) ;; uncert precondition
(not (null constraints))) ;; const
precond
complyto) ;; Move func

( ;; Guarded motion
( ;; Preconditions
(equal uncertainties '(Z))) ;; uncert
precond
guardedto) ;; Move func
update-assem-uncertainty) ;; postcond func

( ;; Biased motion along X axis
( ;; Preconditions
(equal uncertainties '(X))) ;; uncert
precond
biasedxto) ;; Move func
update-assem-uncertainty) ;; postcond func

( ;; Biased motion along Y axis
( ;; Preconditions
(equal uncertainties '(Y))) ;; uncert
precond
biasedyto) ;; Move func
update-assem-uncertainty) ;; postcond func

( ;; Peg in hole motion
( ;; Preconditions
(equal uncertainties '(X Y))) ;; uncert
precond
peginholeto) ;; Move func
update-assem-uncertainty) ;; postcond func

( ;; Rotational search
( ;; Preconditions
(equal uncertainties '(Rotz))) ;; uncert
precond
biasedrotzto) ;; Move func
update-assem-uncertainty) ;; postcond func

( ;; Tilt and slide
( ;; Preconditions
(equal uncertainties '(X Y Rotx Roty)) ;;
ync precondition
nil ;; const cond
(equal (fifth misc) 0)) ;; Misc cond
tiltandslideto) ;; Move func
update-assem-uncertainty) ;;
postcond func

( ;; RCC no Rotz
( ;; Preconditions
(equal uncertainties '(X Y Rotx Roty)) ;;
unc precondition
nil ;; const cond
(equal (fifth misc) 1)) ;; misc cond
recto) ;; move func
update-assem-uncertainty) ;;
postcond func

( ;; Convex peg and hole
( ;; Preconditions
(equal uncertainties '(X Y Rotx Roty Rotz))
;; Uncert precondition
nil ;; No constraint
(and (equal (length (first misc)) 1)
(equal (fifth misc) 0))) ;;
Miscellaneous

```

```

rotpeginholeto) ;; Move func
update-assem-uncertainty) ;; postcond
func

( ;; Multiple peg and hole
( ;; Preconditions
(equal uncertainties '(X Y Rotx Roty Rotz))
;; Uncert precondition
nil ;; No constraint
(and (> (length (first misc)) 1)
(equal (fifth misc) 0))) ;;
Miscellaneous
multiplepeginholeto) ;; Move func
update-assem-uncertainty) ;; Postcond
func

( ;; RCC with Rotz
( ;; Preconditions
(equal uncertainties '(X Y Rotx Roty Rotz))
;; Uncert precondition
nil ;; No constraint
(equal (fifth misc) 1)) ;;
Miscellaneous
rrcwithrotzto) ;; Move func
update-assem-uncertainty) ;; Postcond
func

;;
;; moveto- Function to set up a pt-to-pt motion
;; with no compliance.
;; These funcs call other functions. We will
;; only
;; show the code for a few examples of these
;; funcs
;; since they are highly similar, and there is
;; quite a
;; bit of code involved.
;;
(defun moveto (start lstate cstate assem)
  '(, (goto-position (car lstate) (car cstate))))

(defun complyto (start lstate cstate assem)
  '(, (comply-position (car lstate) (car cstate)
    (caddr cstate))))

(defun guardedto (start lstate cstate assem)
  '(, (guarded-position (car lstate) (car cstate)
    (caddr cstate))))

(defun biasedxto (start lstate cstate assem)
  (let ((point (fourth (fourth cstate))) (dir '-))
    (constraints (third cstate)))
    (if (member '-X constraints) (setq dir '+))
    (biased start point (first cstate) dir 'x
      assem)))

(defun biasedyto (start lstate cstate assem)
  (let ((point (fourth (fourth cstate))) (dir '-))
    (constraints (third cstate)))
    (if (member '-Y constraints) (setq dir '+))
    (biased start point (first cstate) dir 'y
      assem)))

(defun biasedrotzto (start lstate cstate assem)
  (let ((point (fourth (fourth cstate)))
    (rotate-search start point (first cstate)
      assem)))

(defun peginholeto (start lstate cstate assem)
  (let ((point (fourth (fourth cstate)))
    (shape (second (fourth cstate))))

```



```

'(dp 0) (list 'dp e))
(if (member 'Rotz consts)
  '(dp 0) (list 'dp r)))
((Tz -1.2) (Tz 1.2)))
:position-bound
(standard-position-bound x y z d e r))
(if (< (z current) (z last))
  (self (move-goal-terminating-conditions
    motion)
    '((Lz (+ (sz *starting-point*) ,z .15))))
  (self (move-goal-terminating-conditions
    motion)
    '((Gz (+ (sz *starting-point*) ,z -.15))))
  motion))

;; guarded- This routine generates a data
;; structure
;; describing a guarded move.
;; Type=0 Ends either when travelled dist or
;; contact
;; Type=1 Ends with contact
;; Type=2 Ends when travelled dist
;; rcc=t Add constraints '((fx) (fy) (tx) (ty))
;; like rcc
;; rcc=rotz Add constraints '((fx) (fy) (tx)
;; (ty) (tz))
(defun generate-guarded-move (last point current
  unc type rcc)
  (let ((motion) (x (- (x current) (x *first*)))
    (y (- (y current) (y *first*)))
    (z (- (z current) (z *first*)))
    (d (- (d current) (d last)))
    (e (- (e current) (e last)))
    (r (- (r current) (r last)))
    (mx (- (x point) (x *first*)))
    (my (- (y point) (y *first*)))
    (mz (- (z point) (z *first*)))
    (md (- (d point) (d last)))
    (me (- (e point) (e last)))
    (mr (- (r point) (r last))) (dist))

    (setq motion
      (make-instance 'move
        ;; Set up for type 0,1. Change if necessary.
        ;; Should make it somewhere between point
        ;; and current.
        :position-bound
        '(((+ (sx *starting-point*)
          (/ (+ mx x) 2.0))
          (+ (abs (/ (+ mx x) 2.0)) .3))
          ((+ (sy *starting-point*)
            (/ (+ my y) 2.0))
            (+ (abs (/ (+ my y) 2.0)) .3))
          ((+ (sz *starting-point*)
            (/ (+ mz z) 2.0))
            (+ (abs (/ (+ mz z) 2.0)) .3))
          ((+ (sd initial-position)
            (rtd (/ (+ md d) 2.0)))
            (rtd (+ (abs (/ (+ md d) 2.0)) .3)))
          ((+ (se initial-position)
            (rtd (/ (+ me e) 2.0)))
            (rtd (+ (abs (/ (+ me e) 2.0)) .3)))
          ((+ (sr initial-position)
            (rtd (/ (+ mr r) 2.0)))
            (rtd (+ (abs (/ (+ mr r) 2.0)) .3))))))
        (cond ((null rcc)
          (self (move-motion motion)
            'move ((ap ,x) (ap ,y) (ap ,z)
              (dp ,d) (dp ,e) (dp ,r))
              ((Tz 1.2) (Tz -1.2))))
            (equal rcc 'rotz)
            (self (move-motion motion)
              'move ((df 0) (df 0) (ap ,z)
                (dp 0) (dp 0) (df 0))
                ((Tz 1.5) (Tz -1.5))))))
            (t
              (self (move-motion motion)
                'move ((df 0) (df 0) (ap ,z)
                  (dp 0) (dp 0) (dp ,r))
                  ((Tz 1.5) (Tz -1.5))))))
            (cond (
              ;; Ends either when position reach or
              ;; contact
              (equal type 0)

              ;; Allow for typical slop in position
              ;; control.
              (if (< (z last) (z current))
                (setq pcond
                  'Gz (+ (z *starting-point*)
                    (* .95 (- (z current) (z
                      last))))))
                (setq pcond
                  'Lz (+ (z *starting-point*)
                    (* .95 (- (z current) (z
                      last))))))
                (self (move-goal-terminating-conditions
                  motion)
                  'pcond
                  (Bz (+ *force-z* 1.0))
                  (Sz (- *force-z* 1.0))))
              ;; Ends when contact, else protection
              ;; cond hit
              ((equal type 1)
                (self (move-goal-terminating-conditions
                  motion)
                  'pcond
                  (Bz (+ *forcez* 1.0))
                  (Sz (- *forcez* 1.0))))
              ;; Successfully ends when position, else
              ;; protection
              ((equal type 2)
                (if (< (z last) (z current))
                  (setq pcond
                    'Gz (+ (z *starting-point*)
                      (* .95 (- (z current) (z
                        last))))))
                  (setq pcond
                    'Lz (+ (z *starting-point*)
                      (* .95 (- (z current) (z
                        last))))))
                  (self
                    (move-protectional-terminating-conditions
                      motion)
                    'pcond
                    (self (move-goal-terminating-conditions
                      motion)
                      'pcond)
                    (self (move-position-bound motion)
                      '(((+ (sx *starting-point*) ,x) ,(+ 0.2
                        unc))
                      ((+ (sy *starting-point*) ,y) ,(+ 0.2
                        unc))
                      ((+ (sz *starting-point*) ,z) ,(+ 0.2 unc))
                      ((+ (sd initial-position) ,rtd d) 6)
                      ((+ (se initial-position) ,rtd e) 6)
                      ((+ (sr initial-position) ,rtd r) 6)))
                      (self (move-force-bound motion)
                        '(*forcex* 6) (*forcey* 6) (*forcez* 6)
                        (0 inf) (0 inf) (0 inf))))
                    motion))
                (defun standard-position-bound (x y z d e r)

```

```

'(((+ (sx *starting-point*) ,x) 0.3)
  ((+ (sy *starting-point*) ,y) 0.3)
  ((+ (sz *starting-point*) ,z) 0.3)
  ((+ (sd initial-position) ,(rtd d)) 5)
  ((+ (se initial-position) ,(rtd e)) 5)
  ((+ (sr initial-position) ,(rtd r)) 5)))

(defun convert-to-fspec (list)
  (let ((fspec nil))
    (cond ((member 'X list)
      (setq fspec '(fx ty)))
      ((or (member '-X list) (member '+X list))
      (setq fspec '(fx)))
      ((member 'Y list)
      (setq fspec '(fy tx)))
      ((or (member '-Y list) (member '+Y list))
      (setq fspec '(fy)))
      (fspec)))

;; biased- This routine builds up a fine motion
;; plan for a
;; biased strategy. This motion plan consists
;; of guarded
;; move to surface of obst, and a
;; bias-search over obst
(defun biased (start point end dir axis assem)
  (let ((guarded) (bias-search) (unc) (finish)
        (mdist)
        (bias) (xunc) (yunc) (zunc)
        (apu (assembly-pose-uncert assem))
        (acu (assembly-control-uncert assem)))

    ;; Calculate uncertainty in given
    ;; toolframe.
    (setq xunc (+ (x apu)
      (* (abs (- (x start) (x end))) (x acu))))
    (setq yunc (+ (y apu)
      (* (abs (- (y start) (y end))) (y acu))))
    (setq zunc (+ (z apu)
      (* (abs (- (z start) (z end))) (z acu))))

    ;; Get guarded-move that ends in contact.
    (setq guarded
      (generate-guarded-move start point end zunc
        1 nil))

    ;; Now get finishing guarded motion that
    ;; ends at endpoint.
    (setq finish
      (generate-guarded-move start point end zunc
        2 t))

    (cond ((equal axis 'x)
      (multiple-value-setq
        (dx dy dz)
        (convert-to-gripper (caddr point) (+ xunc
          .1) 0 0))
      (setq unc xunc))
      ((equal axis 'y)
      (multiple-value-setq
        (dx dy dz)
        (convert-to-gripper (caddr point) 0 (+
          yunc .1) 0))
      (setq unc yunc)))

    (cond ((equal dir '+)
      (setq bias (make-instance 'move
        :motion '(move ((dp ,dx) (dp ,dy)
          (dp ,dz) (dp 0) (dp 0)
          (dp 0)) nil)
        :force-bound '((*forcecx* 3)
          (*forcecy* 3)
          (*forcecz* 3)
          (*torquex* 10)
          (*torquey* 10)
          (*torquez* 10))
        :position-bound
          '(((+ (sx initial-position) ,dx)
            0.1)
            ((+ (sy initial-position) ,dy)
            0.1)
            ((+ (sz initial-position) ,dz)
            0.1)
            ((sd initial-position) 0.1)
            ((se initial-position) 0.1)
            ((sr initial-position) 0.1))))
        :goal-terminating-conditions '(((Gs ,unc))))
      (setq bias-search (make-instance 'move
        :motion
          '(move ((dp ,dx) (dp ,dy)
            (df 0) (dp 0)
            (dp 0) (dp 0))
            ((Tz 1) (Tz -1)))
          :force-bound '((*forcecx* 3)
            (*forcecy* 3)
            (*forcecz* 3)
            (*torquex* 10)
            (*torquey* 10)
            (*torquez* 10))
          :position-bound
            '(((+ (sx initial-position) ,dx) 0.1)
              ((+ (sy initial-position) ,dy) 0.1)
              ((+ (sz initial-position) ,dz) 0.1)
              ((sd initial-position) 0.1)
              ((se initial-position) 0.1)
              ((sr initial-position) 0.1))))
            :goal-terminating-conditions '(((Gs xunc))))

      (setf bias-search (make-instance 'move
        :motion
          '(move ((dp ,dx) (dp ,dy)
            (df 0) (dp 0)
            (dp 0) (dp 0))
            ((Tz 1) (Tz -1)))
          :force-bound '((*forcecx* 3)
            (*forcecy* 3)
            (*forcecz* 3)
            (*torquex* 10)
            (*torquey* 10)
            (*torquez* 10))
          :position-bound
            '(((+ (sx initial-position) ,dx)
              0.1)
              ((+ (sy initial-position) ,dy)
              0.1)
              ((+ (sz initial-position) ,dz)
              0.1)
              ((sd initial-position) 0.1)
              ((se initial-position) 0.1)
              ((sr initial-position) 0.1))))
            :goal-terminating-conditions '(((Gs xunc))))
    ))

```

```

0.1)
((sd initial-position) 0.1)
((se initial-position) 0.1)
((sr initial-position) 0.1))))))
(if (< (z end) (z start))
  (self
   (move-goal-terminating-conditions
    bias-search)
    '(Bz (+ *negative-z-contact-force* 2))))
  (self
   (move-goal-terminating-conditions
    bias-search)
    '(Sz (- *positive-z-contact-force* 2))))
  (list bias guarded bias-search finish)))

(defun convert-to-gripper (rot x y z)
  (let ((psi (x rot)) (theta (y rot)) (phi (z
  rot)) (n) (o) (a))
    (setq n '(* (cos phi) (cos theta))
          (* (sin phi) (cos theta))
          (- (sin theta))))
    (setq o '(* (- (cos phi) (sin theta) (sin
    psi))
              (* (sin phi) (cos psi))
              (+ (* (sin phi) (sin theta) (sin psi))
                 (* (cos phi) (cos psi))
                 (* (cos theta) (sin psi)))))
    (setq a '(* (+ (* (cos phi) (sin theta) (cos
    psi))
                  (* (sin phi) (cos psi))
                  (- (* (sin phi) (sin theta) (cos psi))
                     (* (cos phi) (sin psi))
                     (* (cos theta) (cos psi)))))
          (values (+ (* (x n) x) (* (x o) y) (* (x a)
    z))
                  (+ (* (y n) x) (* (y o) y) (* (y a) z))
                  (+ (* (z n) x) (* (z o) y) (* (z a)
    z))))))

;; peg-in-hole- Builds motion subplan for
;; mating
;; convex part with concave part when
;; there
;; is uncertainty in X and Y.
(defun peg-in-hole (start point end shape assem)
  (let ((guarded) (get-over-hole) (xunc) (yunc)
        (zunc)
        (skip) (bias) (xunc) (yunc) (zunc) (finish)
        (flag (if (< (z end) (z start)) t nil))
        (apu (assembly-pose-uncert assem))
        (acu (assembly-control-uncert assem)) (latunc))

    ;; Calculate uncertainty in object
    ;; coordinate frame.
    (setq xunc (+ (x apu)
                  (* (abs (- (x start) (x end))) (x acu))))
    (setq yunc (+ (y apu)
                  (* (abs (- (y start) (y end))) (y acu))))
    (setq zunc (+ (z apu)
                  (* (abs (- (z start) (z end))) (z acu))))
    (setq latunc (sqrt (/ (+ (sqr xunc) (sqr
    yunc) (sqr zunc)) 2.0))))

    ;; Get guarded-move that ends in contact or
    ;; when reached
    ;; end if hole alignment was
    ;; serendipitously achieved.
    (setq guarded
      (generate-guarded-move start point end zunc
        0 nil))

    ;; Set up skip condition in case its not
    ;; necessary to

```

```

;; make the remaining moves in the
;; strategy.
(if flag
  (setq skip '((Lz (+ (sz *starting-point*)
    (+ (- (z end) (z start)) 0.05))))))
  (setq skip '((Gz (+ (sz *starting-point*)
    (- (z end) (z start)) 0.05))))))

;; Now get finishing guarded motion that
;; ends at endpoint.
(setq finish
  (generate-guarded-move start point end zunc
    2 t))
(setq (move-skip-conditions finish) skip)
(setq get-over-hole
  (make-instance 'move
    :motion
    '(move ((dp (xtarget
    (* 1.3 latunc)
    (if (equal shape 'peg)
      1 -1)))
      (dp (ytarget
    (* 1.3 latunc)
    (if (equal shape 'peg)
      1 -1)))
      (df 0) nil nil nil)
      (if flag '((Tz 0.7))
        '((Tz -0.7))))
    :skip-conditions skip
    :final-funcs
    '((setq *starting-point*
      (append
        (list (sx *current-position*)
              (sy *current-position*)
              (cddr
                *starting-point*))))))
    :position-bound
    '(((sx initial-position) ,latunc)
      ((sy initial-position) ,latunc)
      ((+ (sz initial-position)
        (if flag -2.2))
        (+ zunc .22))
      ((sd initial-position) ,rtd .3))
      ((se initial-position) ,rtd .3))
      ((sr initial-position) ,rtd .3))))))
  (if flag
    (self
     (move-goal-terminating-conditions
      get-over-hole)
      '(Bz (- *forcez* 1.0))
      (Lz (- (sz initial-position) 0.05))))
    (self
     (move-goal-terminating-conditions
      get-over-hole)
      '(Sz (+ *forcez* 1))
      (Gz (+ (sz initial-position) 0.05))))
    (list guarded get-over-hole finish)))

;; This routine builds the motion subplan for
;; mating a convex part with a concave part when
;; there
;; is uncertainty about the Z axis.
(defun rotate-search (start point end assem)
  (let ((guarded) (rotate) (xunc) (yunc) (zunc)
        (finish)
        (bias) (xunc) (yunc) (zunc) (skip)
        (flag (if (< (z end) (z start)) t nil))
        (apu (assembly-pose-uncert assem))
        (acu (assembly-control-uncert assem)) (rzunc))

    ;; Calculate uncertainty in object
    ;; coordinate frame.
    (setq xunc

```

```

(+ (x apu) (* (abs (- (x start) (x end)))
(x acu)))
(setq yunc
(+ (y apu) (* (abs (- (y start) (y end)))
(y acu)))
(setq zunc
(+ (z apu) (* (abs (- (z start) (z end)))
(z acu)))
(setq rzunc
(+ (r apu) (* (abs (- (r start) (r end)))
(r acu)))

;; guarded-move that ends in contact or when
;; reached
;; end if hole alignment serendipidously
;; achieved.
(setq guarded
(generate-guarded-move
start point end zunc 0 nil))

;; Set up skip cond in case its not necessary
;; to
;; make the remaining moves in the strategy.
(if flag
(setq skip '((Lz (+ (sz *starting-point*)
.- (z end) (z start)) -0.1))))
(setq skip '((Gz (+ (sz *starting-point*)
.- (z end) (z start)) 0.1))))

;; get finishing guarded motion that ends at
;; endpoint.
(setq finish
(generate-guarded-move start point end zunc 2
t))
(setf (move-skip-conditions finish) skip)

(setq rotate
(make-instance 'move
;; Rotate until no contact force left
:motion
(move ((dp 0) (dp 0) (df 0)
(dp 0) (dp 0)
(if (< (- (sr interim-position)
(sr initial-position))
(* 0.9 (rtd rzunc)))
(ap (rtd rzunc))
(ap (- (rtd rzunc))))
(if flag '(Tz 0.8))
'(Tz -0.8)))
:skip-conditions skip
:position-bound
(((sx initial-position) 0.3)
((sy initial-position) 0.3)
((sz initial-position) 0.3)
((sd initial-position) (rtd .3))
((se initial-position) (rtd .3))
((sr initial-position)
(rtd (+ rzunc .3)))))

(if (< (z end) (z start))
(setf (move-goal-terminating-conditions
rotate)
'((Bz (- *forcez* 1))
(Lz (- (sz initial-position) 0.05))))
(setf (move-goal-terminating-conditions
rotate)
'((Sz (- *forcez* 1))
(Gz (+ (sz initial-position) 0.05))))

(list guarded rotate finish)))

```

EXECUTE

```

;; execute- This is the top level routine of
;; the execution unit. strategy is a
;; move data structure, such as the
;; structures instantiated by the
;; planner.
(defun execute (strategy)
(let ((skip nil) (item))
;; Execute initial functions and
;; set up data structure
(update-position)
(setf
(pmove-current-position strategy)
*current-position*)
(dolist (item (pmove-initial-funcs strategy))
(eval item))
(setq initial-position *current-position*)
(update strategy)
;; Make sure that we aren't supposed to
;; skip this step
(dolist (item (pmove-skip-condition strategy))
(if (buildcond item strategy)
(setq skip t))
(cond ((not skip)
(attempt strategy))))
;; Execute final functions.
(dolist (item (pmove-final-funcs
strategy))
(eval item)))

;; attempt- Top level driver to execute the
;; strategy
;; set forth by the planner. It attempts to
;; execute the strategy, and then checks for
;; error conditions, executes a recovery
;; motion if necessary, and reattempts the
;; execution.
(defun attempt (strategy)
;; Current info is now initial info
(current-to-past strategy)
(execute-motion (pmove-motion strategy))
;; Update current conditions
(update strategy)
(cond ((motion-successful strategy) t)
(t (if (recovery strategy)
(attempt strategy) nil))))

;; motion-successful- Returns true if all
;; indications
;; infer the motion was successful.
(defun motion-successful (strategy)
(format t "force= ~s ~% interim-force = ~s ~%"
*forces* interim-force)
(format t "position= ~s ~%" *current-position
*)
(if (and (goal-termination strategy)
(setq *verify*
(verification-motion strategy))
(position-in-bounds strategy)
(force-in-bounds strategy)) t nil))

;; verification-motion- Returns true if
;; verification
;; motion not needed or
;; verification move was made
;; and motion verified success.
(defun verification-motion (strategy)
(verify-if-needed strategy))

```

```

;; recovery- Returns true if an error was detected
;;
;; else it returns nil. If error is detected
;;
;; recovery motion is made, which *should*
;; bring the assembly to the goal position.
(defun recovery (strategy)
  (if (or (forces-out-of-bounds strategy)
        (positions-out-of-bounds strategy)
        (vary-from-path strategy)
        (path-changed strategy)
        (goal-termination-not-met strategy)
        (protectional-terminating-condition-met
         strategy))
      t
      (cond ((pmove-relief-motion strategy)
             (eval (pmove-relief-motion
                    strategy))
             (update-position))
            (t t))))

;; execute-motion- Builds a motion specification
;; and then the motion.
(defun execute-motion (posmotion)
  (let ((type) (new-motion) (motion))
    ; If motion has temporarily been replaced
    ; by
    ; recovery motion, run it, and get rid of
    ; it.
    (cond (*tempmotion*
          (setq motion *tempmotion*)
          (setq *tempmotion* nil))
          (t
           (setq motion posmotion)))
    (setq type (sx motion))
    (generous-ft-spec)
    (setq new-motion (evaluate-motion motion))
    (stable-forces)
    (eval new-motion)
    (update-position)))

(defun evaluate-motion (motion)
  (let ((spec) (item) (new-motion) (type) (axis)
        (mag) (axislist 'x y z d e r)
        (listspec (cadr motion)) (fspec nil)
        (mspec) (tspec) (lptr 0) (curpos) (start-point)
        (label-spec '(fs1 fs2 fs3 fs4 fs5 fs6)))
    (update-position)
    (do ((axis (car axislist) (car axislist))
        (start-pointl *starting-point*
                      (cdr start-pointl))
        (curposl *current-position*
                (cdr curposl))
        (item (car listspec) (car listspec)))
        ((null axis) new-motion)
      (setq axislist (cdr axislist))
      (setq curpos (car curposl))
      (setq start-point (car start-pointl))
      (setq listspec (cdr listspec))
      (setq type (car item))
      (setq mag (cadr item))
      (cond ((equal type 'ap)
            (if (equal mag 0)
                (setq spec '(@spec ,curpos))
                (setq spec '(@spec ,(+ start-point
                                       (eval mag))))))
            ((and (equal type 'dp))
             (cond ((equal mag 0)
                    (setq spec '(@spec ,curpos)))
                   (t (setq spec
                          '(@spec
                            ,(+ curpos
                                (eval mag)))))))
            ((equal type 'df)
             (eval mag))))))
    (format t "spec= ~s ~%" spec)
    (setq new-motion (comply 'fspec ,mspec))
    (format t "new-motion= ~s ~%" new-motion)
    ;; Build up force-terminating spec
    (ppm 'new-motion new-motion)
    (stable-forces)
    (dolist (item (caddr motion))
      (cond ((equal 'Tx (sx item))
            (if (> (sy item) 0)
                (setq spec
                      '(.nth lptr label-spec)
                      (> fx
                        ,(+ (float (sx *forces*)
                                   1.20)
                          (cadr item))))))
            ((equal 'Ty (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fy
                        ,(+ (float (sy *forces*)
                                   1.20)
                          (cadr item))))))
            ((equal 'Tz (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fz
                        ,(+ (float (sz *forces*)
                                   1.20)
                          (cadr item))))))
            (t
             (setq tspec '(.spec ,@tspec)))
            ((equal 'Tz (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fz
                        ,(+ (float (sz *forces*)
                                   1.20)
                          (cadr item))))))
            (t
             (setq tspec '(.spec ,@tspec)))
            (setq lptr (+ 1 lptr)))
      (format t "tspec= ~s ~%" tspec)
      (add-ft-spec tspec)
      ;; (print-ft-spec)
      new-motion))

(defun compose-force-spec (axis)
  (cond ((equal axis 'x) 'fx)
        ((equal axis 'y) 'fy)
        ((equal axis 'z) 'fz)

```

```

        (eval mag))))))
    ((equal type 'df)
     (setq spec '(@spec ,curpos))
     (if fspec
         (setq fspec '(@fspec
                       ,compose-force-spec axis))
         (setq fspec
               '(.compose-force-spec axis))))
    ((null item)
     (setq spec '(@spec ,curpos))))
    (setq *current-pose*
          (make-xyz-rpy (sx spec) (sy spec)
                        (sz spec) (sd spec)
                        (se spec) (sr spec)))
    (setq mspec (move '(*current-pose*)))
    (format t "spec= ~s ~%" spec)
    (setq new-motion (comply 'fspec ,mspec))
    (format t "new-motion= ~s ~%" new-motion)
    ;; Build up force-terminating spec
    (ppm 'new-motion new-motion)
    (stable-forces)
    (dolist (item (caddr motion))
      (cond ((equal 'Tx (sx item))
            (if (> (sy item) 0)
                (setq spec
                      '(.nth lptr label-spec)
                      (> fx
                        ,(+ (float (sx *forces*)
                                   1.20)
                          (cadr item))))))
            ((equal 'Ty (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fy
                        ,(+ (float (sy *forces*)
                                   1.20)
                          (cadr item))))))
            ((equal 'Tz (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fz
                        ,(+ (float (sz *forces*)
                                   1.20)
                          (cadr item))))))
            (t
             (setq tspec '(.spec ,@tspec)))
            ((equal 'Tz (car item))
             (if (> (cadr item) 0)
                 (setq spec
                       '(.nth lptr label-spec)
                       (> fz
                        ,(+ (float (sz *forces*)
                                   1.20)
                          (cadr item))))))
            (t
             (setq tspec '(.spec ,@tspec)))
            (setq lptr (+ 1 lptr)))
      (format t "tspec= ~s ~%" tspec)
      (add-ft-spec tspec)
      ;; (print-ft-spec)
      new-motion))

(defun compose-force-spec (axis)
  (cond ((equal axis 'x) 'fx)
        ((equal axis 'y) 'fy)
        ((equal axis 'z) 'fz)

```

```

(equal axis 'd) 'tx)
(equal axis 'e) 'ty)
(equal axis 'r) 'tz)))

(defun get-dir (axis)
  (cond ((equal axis 'x) :x)
        ((equal axis 'y) :y)
        ((equal axis 'z) :z)
        ((equal axis 'd) :phi)
        ((equal axis 'e) :theta)
        ((equal axis 'r) :psi)))

(defun pm (funct message)
  (cond (*verbose-mode*
        (format t "Inside funct: ~s - " funct)
        (if message (format t "~s ~%" message)
                    (format t "~%"))
        (t nil)))

(defun ppm (funct message)
  (cond (*verbose-mode*
        (format t "Inside funct: ~s - " funct)
        (if message (format t "~s ~%" message)
                    (format t "~%"))
        (cond (*break-mode*
              (format t "Continue? ")
              (setq temp (read))
              (if (member temp '(n no N No))
                  (break funct))
              (if (member temp '(top toplevel))
                  (break funct (reset))))
              (t (format t "~%"))))
        (t nil)))

(defun query-force (&key (robot *current-robot*))
  (let ((ft (get-ft :robot robot)) (f) (m))
    (setq f (robot::force ft))
    (setq m (robot::torque ft))
    (setq *forces* '(, (x f) , (y f) , (z f)
                     , (x m) , (y m) , (z m))))))

(defun stable-forces (&key (robot *current-robot
*))
  (let ((ft (fx 0) (fy 0) (fz 0))
        (tx 0) (ty 0) (tz 0) (f) (m))
    (query-force)
    (setq ft (get-ft :robot robot))
    (setq f (robot::force ft))
    (setq m (robot::torque ft))
    (setq fx (x f)) (setq fy (y f)) (setq fz (z f))
    (setq tx (x m)) (setq ty (y m)) (setq tz (z m))
    (setq *forces* (stabilize fx fy fz tx ty tz
                             robot))))

(defun stabilize (fx fy fz tx ty tz robot)
  (let ((f) (m))
    (get-ft :robot robot)
    (setq ft (get-ft :robot robot))
    (setq f (robot::force ft))
    (setq m (robot::torque ft))
    (cond ((< (abs (- (z f) fz)) 0.2)
           (list (/ (+ (x f) fx) 2.0)
                 (/ (+ (y f) fy) 2.0)
                 (/ (+ (z f) fz) 2.0)
                 (/ (+ (x m) tx) 2.0)
                 (/ (+ (y m) ty) 2.0)
                 (/ (+ (z m) tz) 2.0)))
          (t (stabilize (x f) (y f) (z f)
                        (x m) (y m) (z m)
                        robot))))))

;; update-position- Update current position.
(defun update-position (&key (robot *current-robot
*))
  (let ((pose (get-current-xyz-rpy :robot robot))
        (ht (current-ja-to-tr)))
    (cond ((null *current-pose*)
          (setq *current-pose* pose)
          (setq *current-position*
                '(, (x pose) , (y pose) , (z pose)
                  , (rx pose) , (ry pose)
                  , (rz pose))))
          ((same-orientation ht *current-pose*)
          (setq *current-position*
                '(, (x pose) , (y pose) , (z pose)
                  , (rx *current-pose*)
                  , (ry *current-pose*)
                  , (rz *current-pose*))
          (t (setq *current-position*
                  '(, (x pose) , (y pose)
                    , (z pose) , (rx pose)
                    , (ry pose) , (rz pose))))))

    (setf *verbose-mode* t)
    (setf *break-mode* t)

    (defun same-orientation (ht pose)
      (let ((or (make-otr)))
        ;; Compute orientation matrix for desired
        ;; pose
        (to-htr pose or)
        ;; If n n', s s', a a' ok
        (and (same-vector (n ht) (n or))
              (same-vector (s ht) (s or))
              (same-vector (a ht) (a or))))

    (defun same-vector (a b)
      (and (same-scalar (x a) (x b))
            (same-scalar (y a) (y b))
            (same-scalar (z a) (z b))))

    (defun same-scalar (a b)
      (< (abs (- a b)) 0.005))

    (defun init-compliance ()
      (set-ft-error-handler #'sue-error-handler)
      (set-compliance-mode :world)
      (set-default-cgain)
      (set-cgain :kfx .045)
      (set-speed 0.39))

    (defun sue-error-handler (robot)
      (update-position)
      (generous-ft-spec)
      (format t "Terminating on force constraint
~%" nil)

    (defun generous-ft-spec ()
      (set-ft-spec '( (abs fx (> (abs fx) 50))
                      (abs fy (> (abs fy) 50))
                      (abs fz (> (abs fz) 50))
                      (abs tx (> (abs tx) 150))
                      (abs ty (> (abs ty) 150))
                      (abs tz (> (abs tz) 150))))

    ;; goal-termination- Checks to see if all
    ;; terminating conds were met.
    (defun goal-termination (strategy)
      (let ((term (pmove-goal-terminating-conditions
                    strategy))
            (goal nil) (item))
        (cond (term)
              (dolist (item term)
                (if (buildcond item strategy)
                    (setq goal t)))
              (t t))))

```

```

;; protectional-terminating-condition-met- Returns
;; true if a protectional terminating condition
;; was met.
(defun protectional-terminating-condition-met
  (strategy)
  (let ((term
        (pmove-protectional-terminating-conditions
         strategy))
        (flag nil) (item))
    (dolist (item term)
      (if (buildcond item strategy)
          (setq flag t)))
    (cond (flag)
          (reverse-pmove-motion strategy)
          t)
    (t nil)))

;; path-changed- This routine makes sure the path
;; that the peg is traveling on hasn't
;; changed.
(defun path-changed (strategy)
  (let* ((curpos
         (pmove-current-position strategy))
         (initpos
         (pmove-initial-position strategy))
         (vx (- (sx curpos) (sx initpos)))
         (vy (- (sy curpos) (sy initpos)))
         (vz (- (sz curpos) (sz initpos)))
         (newx (eval
                (sx (pmove-vector strategy))))
         (newy (eval
                (sy (pmove-vector strategy))))
         (newz (eval
                (sz (pmove-vector strategy))))
         (vd (- (sd curpos) (sd initpos)))
         (ve (- (se curpos) (se initpos)))
         (vr (- (sr curpos) (sr initpos)))
         (newd (eval
                (sd (pmove-vector strategy))))
         (newe (eval
                (se (pmove-vector strategy))))
         (newr (eval
                (sr (pmove-vector strategy))))
        )
    (cond ((> (+ (sqr (- newx vx))
                 (sqr (- newy vy))
                 (sqr (- newz vz))) 0.2)
          t)
          ((> (+ (sqr (- newd vd))
                 (sqr (- newe ve))
                 (sqr (- newr vr))) 5)
          t)
          t)
    (format t
            "newd= ~s vd= ~s newe= ~s ve= ~s newr= ~s vr= ~s
            ~%"
            newd vd newe ve newr vr)
    (t nil)))

;; vary-from-path- This routine makes sure the peg
;; has
;; not varied from its intended path,
;; as specified by eqn.
(defun vary-from-path (strategy)
  (cond ((pmove-equation strategy)
        (let ((xreal (eval (sx (pmove-equation
                                strategy))))
              (yreal (eval (sy (pmove-equation
                                strategy))))
              (zreal (eval (sz (pmove-equation
                                strategy))))
              (dreal (eval (sd (pmove-equation
                                strategy))))
              (ereal (eval (se (pmove-equation
                                strategy))))
              (rreal (eval (sr (pmove-equation
                                strategy))))
              (curpos (pmove-current-position strategy)))
          (cond ((> (- (sx curpos) xreal) 0.35)
                  (x-too-big

```

```

strategy
  (- (sx curpos) xreal)))
t)
((> (- xreal (sx curpos)) 0.2)
 (x-too-small
  strategy
  (- (sx curpos) xreal)))
t)
((> (- (sy curpos) yreal) 0.2)
 (y-too-big
  strategy
  (- (sy curpos) yreal)))
t)
((> (- yreal (sy curpos)) 0.2)
 (y-too-small
  strategy
  (- (sy curpos) yreal)))
t)
((> (- (sz curpos) zreal) 0.2)
 (z-too-big
  strategy
  (- (sz curpos) zreal)))
t)
((> (- zreal (sz curpos)) 0.2)
 (z-too-small
  strategy
  (- (sz curpos) zreal)))
t)
((> (- (sd curpos) dreal) 0.2)
 (d-too-big
  strategy
  (- (sd curpos) dreal)))
t)
((> (- dreal (sd curpos)) 0.2)
 (d-too-small
  strategy
  (- (sd curpos) dreal)))
t)
((> (- (se curpos) ereal) 0.2)
 (e-too-big
  strategy
  (- (se curpos) ereal)))
t)
((> (- ereal (se curpos)) 0.2)
 (e-too-small
  strategy
  (- (se curpos) ereal)))
t)
((> (- (sr curpos) rreal) 0.2)
 (r-too-big
  strategy
  (- (sr curpos) rreal)))
t)
((> (- rreal (sr curpos)) 0.2)
 (r-too-small
  strategy
  (- (sr curpos) rreal)))
t)
(t nil)))
(t nil)))

```

```

;; goal-termination-not-met- Returns true if all
;; goal termination conditions were not met.
(defun goal-termination-not-met (strategy)
  (not (goal-termination strategy)))

```

```

;; buildcond- Turns a condition on a motion into
;; a lisp test.
(defun buildcond (cond strategy)
  (cond ((null cond) nil)
        t)
  (let ((axis) (type)
        (mag (eval (sy cond))))

```



```

(current-force) (current-position)
(initial-force) (initial-position))
(multiple-value-setq
 (type axis) (separate-cond (car cond)))
(setq
 current-force
 (cond
 ((equal axis 'x)
 (sx (pmove-current-force strategy)))
 ((equal axis 'y)
 (sy (pmove-current-force strategy)))
 ((equal axis 'z)
 (sz (pmove-current-force strategy)))
 ((equal axis 'd)
 (sd (pmove-current-force strategy)))
 ((equal axis 'e)
 (se (pmove-current-force strategy)))
 ((equal axis 'r)
 (sr (pmove-current-force strategy))))))
(setq
 current-position
 (cond
 ((equal axis 'x)
 (sx (pmove-current-position strategy)))
 ((equal axis 'y)
 (sy (pmove-current-position strategy)))
 ((equal axis 'z)
 (sz (pmove-current-position strategy)))
 ((equal axis 'd)
 (sd (pmove-current-position strategy)))
 ((equal axis 'e)
 (se (pmove-current-position strategy)))
 ((equal axis 'r)
 (sr (pmove-current-position strategy))))))
(setq
 initial-force
 (cond
 ((equal axis 'x)
 (sx (pmove-initial-force strategy)))
 ((equal axis 'y)
 (sy (pmove-initial-force strategy)))
 ((equal axis 'z)
 (sz (pmove-initial-force strategy)))
 ((equal axis 'd)
 (sd (pmove-initial-force strategy)))
 ((equal axis 'e)
 (se (pmove-initial-force strategy)))
 ((equal axis 'r)
 (sr (pmove-initial-force strategy))))))
(setq
 initial-position
 (cond
 ((equal axis 'x)
 (sx (pmove-initial-position strategy)))
 ((equal axis 'y)
 (sy (pmove-initial-position strategy)))
 ((equal axis 'z)
 (sz (pmove-initial-position strategy)))
 ((equal axis 'd)
 (sd (pmove-initial-position strategy)))
 ((equal axis 'e)
 (se (pmove-initial-position strategy)))
 ((equal axis 'r)
 (sr (pmove-initial-position strategy))))))
(cond ((equal type 'G)
 (if (< current-position mag) nil t)
 ((equal type 'L)
 (if (> current-position mag) nil t)
 ((equal type 'B)
 (if (< current-force mag) nil t)
 ((equal type 'S)
 (if (> current-force mag) nil t)
 ((equal type 'T)

```

```

(if (> mag 0)
 (> current-force
 (+ initial-force mag))
 (< current-force
 (+ initial-force mag))))))
(defun separate-cond (a)
 (cond ((equal a 'Lz) (values 'L 'z))
 ((equal a 'Ly) (values 'L 'y))
 ((equal a 'Lx) (values 'L 'x))
 ((equal a 'Ld) (values 'L 'd))
 ((equal a 'Le) (values 'L 'e))
 ((equal a 'Lr) (values 'L 'r))
 ((equal a 'Bz) (values 'B 'z))
 ((equal a 'By) (values 'B 'y))
 ((equal a 'Bx) (values 'B 'x))
 ((equal a 'Br) (values 'B 'r))
 ((equal a 'Be) (values 'B 'e))
 ((equal a 'Bd) (values 'B 'd))
 ((equal a 'Gz) (values 'G 'z))
 ((equal a 'Gy) (values 'G 'y))
 ((equal a 'Gx) (values 'G 'x))
 ((equal a 'Gd) (values 'G 'd))
 ((equal a 'Ge) (values 'G 'e))
 ((equal a 'Gr) (values 'G 'r))
 ((equal a 'Sz) (values 'S 'z))
 ((equal a 'Tz) (values 'T 'z))
 ((equal a 'Ty) (values 'T 'y))
 ((equal a 'Tx) (values 'T 'x))
 ((equal a 'Sy) (values 'S 'y))
 ((equal a 'Sx) (values 'S 'x))
 ((equal a 'Sd) (values 'S 'd))
 ((equal a 'Se) (values 'S 'e))
 ((equal a 'Sr) (values 'S 'r))))
;; verify-if-needed- If an adhoc strategy for
;; verification exists, execute it. Else,
;; we want to verify any motion that
;; terminates on the loss of contact,
;; which is signified by a force servo
;; AND a force terminating cond on the
;; same axis. We assume gain of contact
;; conditions and positional terminating
;; conditions do not need to be verified.
(defun verify-if-needed (strategy)
 (let ((adhoc
 (pmove-ad-hoc-verification strategy))
 (initpos
 (pmove-initial-position strategy))
 (forcex
 (sx (pmove-current-force strategy)))
 (forcey
 (sy (pmove-current-force strategy)))
 (forcez
 (sz (pmove-current-force strategy)))
 (curpos
 (pmove-current-position strategy))
 (motion
 (pmove-motion strategy) (move) (term))
 (term
 (pmove-goal-terminating-conditions
 strategy)))
 (setq move (sy motion))
 (setq term (sz motion))
 (cond
 (adhoc
 (prn verification-needed "adhoc strategy")
 (edr adhoc))
 ((and (member (sx (sx move)) '(df af))
 (assoc-s 'Tx term)
 (no-net-motion
 (cadr (assoc-s 'Tx term))) 'x
 (sx initpos) (sx curpos))

```

```

    (and (> forcex *negative-x-contact-force*)
    (< forcex *positive-x-contact-force*))
    (pm 'verification-needed "in x")
    (make-verification-move
     'x 0.12 (cadr (assoc-s 'Tx term)) strategy))
    ((and (member (sx (sy move)) '(df af))
    (assoc-s 'Ty term)
    (and (> forcey *negative-y-contact-force*)
    (< forcey *positive-y-contact-force*))
    (no-net-motion
    (cadr (assoc-s 'Ty term)) 'y
    (sy initpos) (sy curpos)))
    (pm 'verification-needed "in y")
    (make-verification-move
     'y 0.12 (cadr (assoc-s 'Ty term)) strategy))
    ((and (member (sx (sz move)) '(df af))
    (assoc-s 'Tz term)
    (no-net-motion
    (cadr (assoc-s 'Tz term))
    'z (sz initpos) (sz curpos)))
    (pm 'verification-needed "in z")
    (make-verification-move
     'z 0.06 (cadr (assoc-s 'Tz term)) strategy))
    ((and (member (sx (sz move)) '(df af))
    (assoc-s 'Bz term)
    (and (> forcez
    (+ *negative-z-contact-force* 0.3))
    (< forcez *positive-z-contact-force*))
    (no-net-motion
    1 'z (sz initpos) (sz curpos)))
    (pm 'verification-needed "in z")
    (format t "forcez= ~s ~%" forcez)
    (make-verification-move 'z 0.06 1
    strategy))
    (t t))))

;; no-net-motion- If a compliant motion
;; experiences
;; loss of force, it may make a net motion into
;; the no longer present surface. This motion
;; is one way to know if a real terminating
;; condition was met.
(defun no-net-motion (term axis init cur)
  (cond ((equal axis 'x)
    (cond ((and (< term 0)
    (> (- init cur) 0.02)) nil)
    ((and (> term 0)
    (> (- cur init) 0.02)) nil)
    (t t)))
  (t
  (format t "init= ~s " init)
  (format t "cur= ~s " cur)
  (format t "termag= ~s ~%" term)
  (cond ((and (< term 0)
  (> (- cur init) 0.018)) nil)
  ((and (> term 0)
  (> (- init cur) 0.018))
  (format t "Returning nil~%")
  nil)
  (t t))))

;; make-verification-move- If a motion was made
;; that terminates with loss of contact,
;; one method for verifying this really
;; occurred is by making a small guarded
;; move in that direction and if no contact
;; is made, the terminating condition was met.
;; This routine makes such a guarded move,
;; contact is made, it assumes the motion
;; terminated
;; falsely. If it terminates with no contact
;; before

```

```

;; it goes the entire distance, it retries the
;; guarded motion. Else it signifies success.
;; Returns
;; t if motion believed successful, nil if
;; believed in error.
(defun make-verification-move (axis length termmag
strategy)
  (let ((curpos (pmove-current-position strategy))
  (curforce (pmove-current-force strategy))
  (initpos (pmove-initial-position strategy)))
  (cond ((equal axis 'x)
    (pm 'make-verification-move "in x")
    (if (< termmag 0)
    (execute-motion
    'guarded-move
    ((ap ,(- (sx curpos) length)
    (sx *starting-point*))
    (df 0) (df 0) nil nil nil)
    (Tx 1))))
    (execute-motion
    'guarded-move
    ((ap ,(- (+ (sx curpos) length)
    (sx *starting-point*))
    (df 0) (df 0) nil nil nil)
    (Tx -1))))
    (update strategy)
    (cond
    ((not (no-net-motion
    termmag 'x (sx initpos)
    (sx (pmove-current-position
    strategy)))) t)
    (t nil)))
    ((equal axis 'y)
    (current-to-past strategy)
    (pm 'make-verification-move "in y")
    (if (> termmag 0)
    (execute-motion
    'guarded-move ((df 0)
    (ap ,(- (sy curpos) length)
    (sy *starting-point*))
    (df 0) nil nil nil)
    (Ty -1))))
    (execute-motion
    'guarded-move ((df 0)
    (sy ,((+ (sy curpos) length)
    (sy *starting-point*))
    (df 0) nil nil nil)
    (Ty 1))))
    (update strategy)
    (cond ((not (no-net-motion
    termmag 'y (sy initpos)
    (sy (pmove-current-position
    strategy)))) t)
    (t nil)))
    ((equal axis 'z)
    (current-to-past strategy)
    (pm 'make-verification-move "in z")
    (if (> termmag 0)
    (execute-motion
    'guarded-move ((df 0) (df 0)
    (dp ,(- length)
    nil nil nil) ((Tz -3))))
    (execute-motion
    'guarded-move ((df 0) (df 0)
    (ap ,(- (+ (sz curpos) length)
    (sz *starting-point*))
    nil nil nil) ((Tz 1))))
    (update strategy)
    (cond ((and (not (no-net-motion
    termmag 'z (sz initpos)
    (sz (pmove-current-position
    strategy))))
    (> (sz (pmove-current-force strategy))

```



```

(sy (pmove-current-position strategy))))
(z0 (eval
  (sz (pmove-current-position strategy))))
(dc (eval
  (sx (sd (pmove-position-bound
    strategy))))))
(g (eval
  (sy (sd (pmove-position-bound
    strategy))))))
(ec (eval
  (sx (se (pmove-position-bound
    strategy))))))
(h (eval
  (sy (se (pmove-position-bound
    strategy))))))
(rc (eval
  (sx (sr (pmove-position-bound
    strategy))))))
(i (eval
  (sy (sr (pmove-position-bound
    strategy))))))
(d0 (eval
  (sd (pmove-current-position strategy))))
(e0 (eval
  (se (pmove-current-position strategy))))
(r0 (eval
  (sr (pmove-current-position strategy))))
(cond
  ((or (> (abs (- x0 xc)) a)
    (> (abs (- y0 yc)) b)
    (> (abs (- z0 zc)) c)
    (> (abs (- d0 dc)) g)
    (> (abs (- e0 ec)) h)
    (> (abs (- r0 rc)) i))
    (pm 'position-in-bounds
      "position out of bounds")
    nil)
  (t t)))

;; check-under-bounds- Checks to see if we are out
;; of bounds, but if we continue along vector we
;; will be in bounds. If so, continue motion.
(defun check-under-bounds (strategy)
  (let ((vector (pmove-vector strategy)))
    (xc (eval (sx (sx (pmove-position-bound
      strategy))))))
    (a (eval (sy (sx (pmove-position-bound
      strategy))))))
    (yc (eval (sx (sy (pmove-position-bound
      strategy))))))
    (b (eval (sy (sy (pmove-position-bound
      strategy))))))
    (zc (eval (sx (sz (pmove-position-bound
      strategy))))))
    (c (eval (second (sz (pmove-position-bound
      strategy))))))
    (dc (eval (sx (sd (pmove-position-bound
      strategy))))))
    (g (eval (sy (sd (pmove-position-bound
      strategy))))))
    (ec (eval (sx (se (pmove-position-bound
      strategy))))))
    (h (eval (sy (se (pmove-position-bound
      strategy))))))
    (rc (eval (sx (sr (pmove-position-bound
      strategy))))))
    (i (eval (sy (sr (pmove-position-bound
      strategy))))))
    (x0) (y0) (z0) (d0) (e0) (r0) (mx)
    (my) (mz) (md) (me) (mr))
    (setq x0 (- (eval (sx
      (pmove-current-position
        strategy))) xc))
    (setq y0 (- (eval (sy
      (pmove-current-position
        strategy))) yc))
    (setq z0 (- (eval (sz
      (pmove-current-position
        strategy))) zc))
    (setq d0 (- (eval (sd
      (pmove-current-position
        strategy))) dc))
    (setq e0 (- (eval (se
      (pmove-current-position
        strategy))) ec))
    (setq r0 (- (eval (sr
      (pmove-current-position
        strategy))) rc))
    (setq mx (eval (sx vector)))
    (setq my (eval (sy vector)))
    (setq mz (eval (sz vector)))
    (setq md (eval (sd vector)))
    (setq me (eval (se vector)))
    (setq mr (eval (sr vector)))
    (cond ((and (> (abs x0) a) (> (abs mx) 0))
      (if (< (* x0 mx) 0) t nil))
      ((and (> (abs y0) b) (> (abs my) 0))
      (if (< (* y0 my) 0) t nil))
      ((and (> (abs z0) c) (> (abs mz) 0))
      (if (< (* z0 mz) 0) t nil))
      ((and (> (abs d0) g) (> (abs md) 0))
      (if (< (* d0 md) 0) t nil))
      ((and (> (abs e0) h) (> (abs me) 0))
      (if (< (* e0 me) 0) t nil))
      ((and (> (abs r0) i) (> (abs mr) 0))
      (if (< (* r0 mr) 0) t nil))
      (t nil))))

```

```

(setq y0 (- (eval (sy
  (pmove-current-position
    strategy))) yc))
(setq z0 (- (eval (sz
  (pmove-current-position
    strategy))) zc))
(setq d0 (- (eval (sd
  (pmove-current-position
    strategy))) dc))
(setq e0 (- (eval (se
  (pmove-current-position
    strategy))) ec))
(setq r0 (- (eval (sr
  (pmove-current-position
    strategy))) rc))
(setq mx (eval (sx vector)))
(setq my (eval (sy vector)))
(setq mz (eval (sz vector)))
(setq md (eval (sd vector)))
(setq me (eval (se vector)))
(setq mr (eval (sr vector)))
(cond ((and (> (abs x0) a) (> (abs mx) 0))
  (if (< (* x0 mx) 0) t nil))
  ((and (> (abs y0) b) (> (abs my) 0))
  (if (< (* y0 my) 0) t nil))
  ((and (> (abs z0) c) (> (abs mz) 0))
  (if (< (* z0 mz) 0) t nil))
  ((and (> (abs d0) g) (> (abs md) 0))
  (if (< (* d0 md) 0) t nil))
  ((and (> (abs e0) h) (> (abs me) 0))
  (if (< (* e0 me) 0) t nil))
  ((and (> (abs r0) i) (> (abs mr) 0))
  (if (< (* r0 mr) 0) t nil))
  (t nil)))

;; check-over-bounds- Checks to see if we are out
;; of
;; bounds because we went too far along vector.
;; If so reverses direction of motion.
(defun check-over-bounds (strategy)
  (let ((vector (pmove-vector strategy)))
    (xc (eval (sx (sx (pmove-position-bound
      strategy))))))
    (a (eval (second (sx (pmove-position-bound
      strategy))))))
    (yc (eval (sx (sy (pmove-position-bound
      strategy))))))
    (b (eval (second (sy (pmove-position-bound
      strategy))))))
    (zc (eval (sx (sz (pmove-position-bound
      strategy))))))
    (c (eval (second (sz (pmove-position-bound
      strategy))))))
    (dc (eval (sx (sd (pmove-position-bound
      strategy))))))
    (g (eval (second (sd (pmove-position-bound
      strategy))))))
    (ec (eval (sx (se (pmove-position-bound
      strategy))))))
    (h (eval (sy (se (pmove-position-bound
      strategy))))))
    (rc (eval (sx (sr (pmove-position-bound
      strategy))))))
    (i (eval (sy (sr (pmove-position-bound
      strategy))))))
    (x0) (y0) (z0) (d0) (e0) (r0)
    (mx) (my) (mz) (md) (me) (mr))
    (setq x0 (- (eval (sx
      (pmove-current-position
        strategy))) xc))
    (setq y0 (- (eval (sy
      (pmove-current-position
        strategy))) yc))
    (setq z0 (- (eval (sz
      (pmove-current-position
        strategy))) zc))
    (setq d0 (- (eval (sd
      (pmove-current-position
        strategy))) dc))
    (setq e0 (- (eval (se
      (pmove-current-position
        strategy))) ec))
    (setq r0 (- (eval (sr
      (pmove-current-position
        strategy))) rc))
    (setq mx (eval (sx vector)))
    (setq my (eval (sy vector)))
    (setq mz (eval (sz vector)))
    (setq md (eval (sd vector)))
    (setq me (eval (se vector)))
    (setq mr (eval (sr vector)))
    (cond ((and (> (abs x0) a) (> (abs mx) 0))
      (if (< (* x0 mx) 0) t nil))
      ((and (> (abs y0) b) (> (abs my) 0))
      (if (< (* y0 my) 0) t nil))
      ((and (> (abs z0) c) (> (abs mz) 0))
      (if (< (* z0 mz) 0) t nil))
      ((and (> (abs d0) g) (> (abs md) 0))
      (if (< (* d0 md) 0) t nil))
      ((and (> (abs e0) h) (> (abs me) 0))
      (if (< (* e0 me) 0) t nil))
      ((and (> (abs r0) i) (> (abs mr) 0))
      (if (< (* r0 mr) 0) t nil))
      (t nil))))

```

```

(setq z0 (- (eval (sz
  (pmove-current-position
    strategy))) zc))
(setq d0 (- (eval (sd
  (pmove-current-position
    strategy))) dc))
(setq e0 (- (eval (se
  (pmove-current-position
    strategy))) ec))
(setq r0 (- (eval (sr
  (pmove-current-position
    strategy))) rc))
(setq mx (eval (sx vector))) (setq my (eval
  (sy vector)))
(setq mz (eval (sz vector))) (setq md (eval
  (sd vector)))
(setq me (eval (se vector))) (setq mr (eval
  (sr vector)))
(cond ((and (> (abs x0) a) (> (abs mx) 0))
  (if (> (* x0 mx) 0) t nil))
  ((and (> (abs y0) b) (> (abs my) 0))
  (if (> (* y0 my) 0) t nil))
  ((and (> (abs z0) c) (> (abs mz) 0))
  (if (> (* z0 mz) 0) t nil))
  ((and (> (abs d0) g) (> (abs md) 0))
  (if (> (* d0 md) 0) t nil))
  ((and (> (abs e0) h) (> (abs me) 0))
  (if (> (* e0 me) 0) t nil))
  ((and (> (abs r0) i) (> (abs mr) 0))
  (if (> (* r0 mr) 0) t nil))
  (t nil)))

;; replace-motion- Replaces an old positional
;; condition with a new one.
(defun replace-motion (axis mag motion)
  (let ((type (sx motion))
        (motionspec (sy motion))
        (term (sz motion)) (newmove))
    (cond
      ((equal axis 'x)
       (setq newmove
        (type ((ap ,mag)
          @(cdr motionspec)) term)))
      ((equal axis 'y)
       (setq newmove
        (type ((sx motionspec) (ap ,mag)
          @(cddr motionspec)) term)))
      ((equal axis 'z)
       (setq newmove
        (type ((sx motionspec) (sy
          motionspec)
          (ap ,mag) @(cddr motionspec))
          term)))
      ((equal axis 'd)
       (setq newmove
        (type ((sx motionspec) (sy
          motionspec)
          (sz motionspec) (ap ,mag)
          @(cdddr motionspec)) term)))
      ((equal axis 'e)
       (setq newmove
        (type ((sx motionspec) (sy
          motionspec)
          (sz motionspec) (sd motionspec)
          (ap ,mag) (sr motionspec)) term)))
      ((equal axis 'r)
       (setq newmove
        (type ((sx motionspec) (sy
          motionspec)
          (sz motionspec) (sd motionspec)
          (se motionspec) (ap ,mag)) term))))
    (pm 'replace-motion ('considering newmove))
  )

```

```

  newmove))

;; move-in-negative-x- This strategy generally
;; does
;; a guarded motion in the negative x direction
;;
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; x-to-small to get around the problem and
;; continue in the positive x direction.
(defun move-in-negative-x (strategy xpos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy))
        (vx (eval (sx (pmove-vector
      strategy))))))
    (cond ((or (assoc-s 'Gx goal)
      (assoc-s 'Tx goal))
      (pm 'move-in-negative-x nil)
      (execute-motion
        'comply ((dp -.1) (df 0) (df 0)
          nil nil nil nil)))
      (t (x-too-small strategy (/ vx 2))))))

;; move-in-positive-x- This strategy generally
;; does
;; a guarded motion in the positive x direction,
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; x-to-big to get around the problem and
;; continue
;; in the negative x direction.
(defun move-in-positive-x (strategy xpos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy))
        (vx (eval (sx (pmove-vector strategy))))))
    (cond ((or (assoc-s 'Lx goal) (assoc-s 'Tx
      goal))
      (pm 'move-in-positive-x nil)
      (execute-motion
        'comply ((dp .1) (df 0) (df 0)
          nil nil nil nil)))
      (t (x-too-big strategy (/ vx 2))))))

;; move-in-negative-y- This strategy generally
;; does
;; a guarded motion in the negative y direction,
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; y-to-small to get around the problem and
;; continue in the negative y direction.
(defun move-in-negative-y (strategy ypos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy))
        (vy (eval (sy (pmove-vector strategy))))))
    (cond ((or (assoc-s 'Gy goal)
      (assoc-s 'Ty goal))
      (pm 'move-in-negative-y nil)
      (execute-motion
        'comply ((df 0) (dp -.1) (df 0)
          nil nil nil nil)))
      (t (y-too-small strategy (/ vy 2))))))

;; move-in-positive-y- This strategy generally
;; does
;; a guarded motion in the positive y direction,
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; y-to-big to get around the problem and
;; continue in the negative y direction.

```

```
(defun move-in-positive-y (strategy ypos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy)))
    (vy (eval (sy (pmove-vector strategy))))
    (cond ((or (assoc-s 'Ly goal)
      (assoc-s 'Ty goal))
      (pm 'move-in-positive-y nil)
      (execute-motion
        '(comply ((df 0) (dp .1) (df 0)
          nil nil nil) nil)))
      (t (y-too-big strategy (/ vy 2))))))
```

```
;; move-in-negative-z- This strategy generally
;; does
;; a guarded motion in the negative z direction,
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; z-too-small to get around the problem and
;; continue in the negative y direction.
```

```
(defun move-in-negative-z (strategy zpos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy)))
    (vz (eval (sz (pmove-vector strategy))))
    (cond ((or (assoc-s 'Gz goal)
      (assoc-s 'Tz goal))
      (pm 'move-in-negative-z nil)
      (execute-motion
        '(comply ((df 0) (df 0) (dp -.1)
          nil nil nil) nil)))
      (t (z-too-small strategy (/ vz 2))))))
```

```
;; move-in-positive-z- This strategy generally
;; does
;; a guarded motion in the positive z direction,
;; if we overshot a force constraint. In the
;; event
;; we inadvertently run into something we use
;; z-too-small to get around the problem and
;; continue in the positive z direction.
```

```
(defun move-in-positive-z (strategy zpos)
  (let ((goal (pmove-goal-terminating-conditions
    strategy)))
    (motion (pmove-motion strategy))
    (vz (eval (sz (pmove-vector strategy))))
    (cond ((or (assoc-s 'Lz goal)
      (assoc-s 'Tz goal))
      (pm 'move-in-positive-z nil)
      (execute-motion
        '(comply ((df 0) (dp .03)
          nil nil nil) ((Tz 4))))
      (and (member (sx (sz (sy motion)))
        '(af df))
        (> (third *forces*)
          (+ *negative-z-contact-force*
            1.5)))
      (execute-motion
        '(comply ((df 0) (df 0) (dp -.05)
          nil nil nil) ((Tz -0.8))))))
    (t (z-too-big strategy (/ vz 2))))))
```

```
;; z-too-small- In the event we moved to far in -z
;; we want to make a move in +z without altering
;; anything else about situation. If we have a
;; contact force in +z, we move a little bit in a
;; free direction and then try the guarded move.
(defun z-too-small (strategy length)
  (let
    ((zforce (sz (pmove-current-force strategy)))
     (xforce (sx (pmove-current-force strategy)))
     (yforce (sy (pmove-current-force strategy)))
     (curpos (pmove-current-position strategy))
     (motion (pmove-motion strategy)))
```

```
(termforce 0) (movespec))
(current-to-past strategy)
(setq movespec (sy motion))
(pm 'z-too-small nil)
(cond ((> zforce (- *positive-z-contact-force*
  0.2))
  (pm 'z-too-small "moving out of way")
  (cond
    ((member (sx (sz movespec)) '(af df))
     (execute-motion
       '(comply ((df 0) (df 0) (dp -.05)
         nil nil nil) nil)))
    ((not (member (sx (sx movespec)) '(af df)))
     (if (> *forcex* xforce)
       (execute-motion
         '(comply ((dp -.1) (df 0) (df 0)
           nil nil nil) ((Tx 1))))
       (execute-motion
         '(comply ((dp .1) (df 0) (df 0)
           nil nil nil) ((Tx -1))))))
    ((not (member (sx (sy movespec)) '(af df)))
     (if (> yforce *forcey*)
       (execute-motion
         '(comply ((df 0) (dp -.1) (df 0)
           nil nil nil) ((Ty -1))))
       (execute-motion
         '(comply ((df 0) (dp .1) (df 0)
           nil nil nil) ((Ty 1))))))
    (update strategy)
    (setq zforce (sz (pmove-current-force
      strategy)))
    (setq termforce (+ *positive-z-contact-force*
      1))
    (if (> termforce zforce)
      (execute-motion
        '(comply
          ((df 0) (df 0) (dp ,length)
            nil nil nil)
          ((Tz ,(- termforce zforce))))))
    (update strategy)
    (if (vary-from-path strategy)
      (z-too-small
        strategy
        (- length
          (sz (pmove-current-position strategy))
          (sz (pmove-initial-position
            strategy)))))))))
```

```
;; z-too-big- In the event we moved to far in +z
;; we want to make a move in -z without altering
;; anything else about situation. If we have a
;; contact force in +z, we move a little bit in a
;; free direction and then try the guarded move.
```

```
(defun z-too-big (strategy length)
  (let
    ((zforce (sz (pmove-current-force strategy)))
     (xforce (sx (pmove-current-force strategy)))
     (yforce (sy (pmove-current-force strategy)))
     (curpos (pmove-current-position strategy))
     (motion (pmove-motion strategy))
     (movespec) (termforce 0))
    (current-to-past strategy)
    (setq movespec (sy motion))
    (pm 'z-too-big nil)
    (cond
      ((< zforce (+ *negative-z-contact-force* 0.2))
       (pm 'z-too-big "moving out of way")
       (cond
         ((member (sx (sz movespec)) '(af df))
          (execute-motion
            '(comply ((df 0) (df 0) (dp .05)
              nil nil nil) nil)))
         ((not (member (sx (sx movespec)) '(af df)))
```

```

(if (> *forcex* xforce)
  (execute-motion
    'comply ((dp -1) (df 0) (df 0)
      nil nil nil) ((Tx 1))))
(execute-motion
  'comply ((dp .1) (df 0) (df 0)
    nil nil nil) ((Tx -1))))
((not (member (sx (sy movespec)) '(af df)))
  (if (> yforce *forcey*)
    (execute-motion
      'comply ((df 0) (dp -1) (df 0)
        nil nil nil) ((Ty -1))))
    (execute-motion
      'comply ((df 0) (dp .1) (df 0)
        nil nil nil) ((Ty 1))))))
(update strategy)
(setq zforce (sz (pmove-current-force
  strategy)))
(setq termforce (- *negative-z-contact-force*
  1))
(if (< termforce zforce)
  (execute-motion
    'comply ((df 0) (dp 0) (- length)
      nil nil nil)
    ((Tz (- termforce zform))))))
(update strategy)
(if (vary-from-path strategy)
  (z-too-big
    strategy
    (- length
      (- (sz (pmove-initial-position strategy))
        (sz (pmove-current-position
          strategy)))))))
;; y-too-small- In the event we moved to far in -y
;; we want to make a move in +y without altering
;; anything else about situation. If we have a
;; contact force in +y, we move a little bit in a
;; free direction and then try the guarded move.
(defun y-too-small (strategy length)
  (let
    ((xforce (sx (pmove-current-force strategy)))
      (yforce (sy (pmove-current-force strategy)))
      (zforce (sz (pmove-current-force strategy)))
      (curpos (pmove-current-position strategy))
      (motion (pmove-motion strategy))
      (movespec (termforce 0))
      (current-to-past strategy)
      (setq movespec (sy motion))
      (pm 'y-too-small nil))
    (cond
      ((> yforce (- *positive-y-contact-force* 0.2))
        (pm 'y-too-small "Moving out of way")
        (cond
          ((member (sx (sy movespec)) '(af df))
            (execute-motion
              'comply ((df 0) (dp -.05) (df 0)
                nil nil nil) nil)))
          ((not (member (sx (sx movespec)) '(af df)))
            (if (> *forcex* xforce)
              (execute-motion
                'comply ((dp -1) (df 0) (df 0)
                  nil nil nil) ((Tx 1))))
              (execute-motion
                'comply ((dp .1) (df 0) (df 0)
                  nil nil nil) ((Tx -1))))))
          ((not (member (sx (sz movespec)) '(af df)))
            (if (> zforce *forcez*)
              (execute-motion
                'comply ((df 0) (df 0) (dp -1)
                  nil nil nil) ((Tz -1))))
              (execute-motion
                'comply ((df 0) (df 0) (dp .1)
                  nil nil nil) ((Tz 1))))))
          (update strategy)
          (setq yforce (sy (pmove-current-force
            strategy)))
          (setq termforce (- *negative-y-contact-force*
            1))
          (if (< termforce yforce)
            (execute-motion
              'comply ((df 0) (dp 0) (- length)
                (df 0) nil nil nil)
              ((Ty (- termforce yforce))))))
          (update strategy)
          (if (vary-from-path strategy)
            (y-too-small
              strategy
              (- length
                (- (sz (pmove-initial-position strategy))
                  (sz (pmove-current-position
                    strategy))))))
            (y-too-big
              strategy
              length))))))

```

```

'comply ((df 0) (df 0) (dp .1)
  nil nil nil) ((Tz 1))))))
(update strategy)
(setq yforce (sy (pmove-current-force
  strategy)))
(setq termforce (+ *positive-y-contact-force*
  1))
(if (> termforce yforce)
  (execute-motion
    'comply ((df 0) (dp .length) (df 0)
      nil nil nil)
    ((Ty (- termforce yforce))))))
(update strategy)
(if (vary-from-path strategy)
  (y-too-small
    strategy
    (- length
      (- (sy (pmove-current-position strategy))
        (sy (pmove-initial-position
          strategy)))))))
;; y-too-big- In the event we moved to far in -y
;; we want to make a move in +y without altering
;; anything else about situation. If we have a
;; contact force in +y, we move a little bit in a
;; free direction and then try the guarded move.
(defun y-too-big (strategy length)
  (let
    ((xforce (sx (pmove-current-force strategy)))
      (yforce (sy (pmove-current-force strategy)))
      (zforce (sz (pmove-current-force strategy)))
      (curpos (pmove-current-position strategy))
      (motion (pmove-motion strategy))
      (movespec (termforce 0))
      (pm 'y-too-big nil)
      (setq movespec (sy motion))
      (current-to-past strategy)
      (cond
        ((< yforce (+ *negative-y-contact-force* 0.2))
          (pm 'y-too-big "Moving out of way")
          (cond
            ((member (sx (sy movespec)) '(af df))
              (execute-motion
                'comply ((df 0) (dp .05) (df 0)
                  nil nil nil) nil)))
            ((not (member (sx (sx movespec)) '(af df)))
              (if (> *forcex* xforce)
                (execute-motion
                  'comply ((dp -1) (df 0) (df 0)
                    nil nil nil) ((Tx 1))))
                (execute-motion
                  'comply ((dp .1) (df 0) (df 0)
                    nil nil nil) ((Tx -1))))))
            ((not (member (sx (sz movespec)) '(af df)))
              (if (> zforce *forcez*)
                (execute-motion
                  'comply ((df 0) (df 0) (dp -1)
                    nil nil nil) ((Tz -1))))
                (execute-motion
                  'comply ((df 0) (df 0) (dp .1)
                    nil nil nil) ((Tz 1))))))
            (update strategy)
            (setq yforce (sy (pmove-current-force
              strategy)))
            (setq termforce (- *negative-y-contact-force*
              1))
            (if (< termforce yforce)
              (execute-motion
                'comply ((df 0) (dp 0) (- length)
                  (df 0) nil nil nil)
                ((Ty (- termforce yforce))))))
            (update strategy)
            (if (vary-from-path strategy)
              (y-too-small
                strategy
                (- length
                  (- (sz (pmove-initial-position strategy))
                    (sz (pmove-current-position
                      strategy))))))
              (y-too-big
                strategy
                length))))))

```

```

(y-too-big
 strategy
 (- length
  (- (sy (pmove-initial-position strategy))
    (sy (pmove-current-position
         strategy))))))

;; x-too-small- In the event we moved to far in -x
;; we want to make a move in +x without altering
;; anything else about situation. If we have a
;; contact force in +x, we move a little bit in a
;; free direction and then try the guarded move.
(defun x-too-small (strategy length)
  (let
   ((xforce (sx (pmove-current-force strategy)))
    (yforce (sy (pmove-current-force strategy)))
    (zforce (sz (pmove-current-force strategy)))
    (curpos (pmove-current-position strategy))
    (motion (pmove-motion strategy))
    (movespec) (termforce 0))
   (current-to-past strategy)
   (setq movespec (sy motion))
   (pm 'x-too-small nil)
   (cond
    ((< xforce (+ *negative-x-contact-force* 0.2))
     (pm 'x-too-small "Moving out of way")
     (cond
      ((member (sx (sx movespec)) '(df af))
       (execute-motion
        'comply ((dp -0.2) (df 0) (df 0)
                  nil nil nil) nil)))
      ((not (member (sx (sy movespec)) '(af df)))
       (if (> yforce *forcey*)
           (execute-motion
            'comply ((df 0) (dp -0.1) (df 0)
                      nil nil nil) ((Ty -1))))
          (execute-motion
           'comply ((df 0) (dp 0.1) (df 0)
                      nil nil nil) ((Ty 1.5))))
      ((not (member (sx (sz movespec)) '(af df)))
       (if (> zforce *forcez*)
           (execute-motion
            'comply ((df 0) (dp 0) (dp -0.1)
                      nil nil nil) ((Tz -1))))
          (execute-motion
           'comply ((df 0) (dp 0) (dp -0.1)
                      nil nil nil) ((Tz 1))))
      (update strategy)
      (setf xforce (sx (pmove-current-force
                        strategy)))
      (setf termforce (- *negative-x-contact-force
                          * 1))
      (if (< termforce xforce)
          (execute-motion
           'comply ((dp ,length) (df 0) (df 0)
                     nil nil nil)
            ((Tx ,(- termforce xforce))))
          (update strategy)
           (if (vary-from-path strategy)
               (x-too-small
                strategy
                (- length
                 (- (sx (pmove-current-position strategy))
                  (sx (pmove-initial-position
                      strategy))))))
            (x-too-big
             strategy
             (- length
              (- (sx (pmove-initial-position strategy))
               (sx (pmove-current-position
                   strategy)))))))))

;; x-too-big- In the event we moved to far in +x
;; we want to make a move in -x without altering
;; anything else about situation. If we have a
;; contact force in -x, we move a little bit in a
;; free direction and then try the guarded move.
(defun x-too-big (strategy length)
  (let
   ((xforce (sx (pmove-current-force strategy)))
    (yforce (sy (pmove-current-force strategy)))
    (zforce (sz (pmove-current-force strategy)))
    (curpos (pmove-current-position strategy))
    (motion (pmove-motion strategy))
    (movespec) (termforce 0))
   (current-to-past strategy)
   (setf movespec (sy motion))
   (pm 'x-too-big nil)
   (cond
    ((> xforce (- *positive-x-contact-force* 0.2))
     (pm 'x-too-big "Moving out of way")
     (cond
      ((member (sx (sx movespec)) '(df af))
       (execute-motion
        'comply ((dp 0.2) (df 0) (df 0)
                  nil nil nil) nil)))
      ((not (member (sx (sy movespec)) '(af df)))
       (if (> yforce *forcey*)
           (execute-motion
            'comply ((df 0) (dp -1) (df 0)
                      nil nil nil) ((Ty -1))))
          (execute-motion
           'comply ((df 0) (dp 0.1) (df 0)
                      nil nil nil) ((Ty 1))))
      ((not (member (sx (sz movespec)) '(af df)))
       (if (> zforce *forcez*)
           (execute-motion
            'comply ((df 0) (df 0) (dp -1)
                      nil nil nil) ((Tz -1))))
          (execute-motion
           'comply ((df 0) (df 0) (dp 1)
                      nil nil nil) ((Tz 1))))
      (update strategy)
      (setf xforce (sx (pmove-current-force
                        strategy)))
      (setf termforce (+ *positive-x-contact-force*
                          1))
      (if (> termforce xforce)
          (execute-motion
           'comply ((dp ,(- length)) (df 0) (df 0)
                     nil nil nil)
            ((Tx ,(- termforce xforce))))
          (update strategy)
           (if (vary-from-path strategy)
               (x-too-big
                strategy
                (- length
                 (- (sx (pmove-initial-position strategy))
                  (sx (pmove-current-position
                      strategy))))))
            (x-too-small
             strategy
             (- length
              (- (sx (pmove-initial-position strategy))
               (sx (pmove-current-position
                   strategy)))))))))

;; reverse-pmove-motion- Reverses direction of
;; positional motion by finding the vector it
;; had traveled along, negating it, scaling it,
;; and adding it to the current position.
(defun reverse-pmove-motion (strategy)
  (let*
   ((bound (pmove-position-bound strategy))
    (xinit (eval (first (sx bound))))
    (xbound (eval (second (sx bound))))
    (yinit (eval (first (sy bound))))
    (ybound (eval (second (sy bound))))
    (zinit (eval (first (sz bound))))
    (zbound (eval (second (sz bound))))
    (xcurs (sx (pmove-current-position strategy)))
    (ycurs (sy (pmove-current-position strategy)))
    (zcurs (sz (pmove-current-position strategy)))
    (p 0) (tempflag nil) (axismove nil)
    (movespec nil) (newmovespec)
    (tempmove (pmove-motion strategy)))
    (setf movespec (sy tempmove))

```



```

(pm 'reverse-pmove-motion tempmove)
(cond
  ((member (sx (sx movespec)) '(df af))
   (setq axismove nil))
  ((equal (sx (sx movespec)) 'ap)
   (if (equal 0 (sy (sx movespec)))
       (setq axismove nil)
       (setq axismove (sy (sx movespec))))))
  (t (setq axismove (sy (sx movespec)))))
(cond
  (axismove
   (if (not (numberp axismove))
       (setq tempflag t)
       (if (> xcurs xinit)
           (setq p (- xinit xbound (sx *starting-point*)))
           (setq p (- (+ xinit xbound) (sx *starting-point*))))))
  (setq axismove '(ap p))
  (setq newmovespec '(.axismove)))
(t (setq newmovespec '(.sx movespec)))))
(cond
  ((member (sx (sy movespec)) '(df af))
   (setq axismove nil))
  ((equal (sx (sy movespec)) 'ap)
   (if (equal 0 (sy (sy movespec)))
       (setq axismove nil)
       (setq axismove (sy (sy movespec))))))
  (t (setq axismove (sy (sy movespec)))))
(cond
  (axismove
   (if (not (numberp axismove))
       (setq tempflag t)
       (if (> ycurs yinit)
           (setq p (- yinit ybound (sy *starting-point*)))
           (setq p (- (+ yinit ybound) (sy *starting-point*))))))
  (setq axismove '(ap p))
  (setq newmovespec '(.@newmovespec
    ,axismove)))
(t (setq newmovespec
    '(.@newmovespec ,sy movespec)))))
(cond
  ((member (sx (sz movespec)) '(df af))
   (setq axismove nil))
  ((equal (sx (sz movespec)) 'ap)
   (if (equal 0 (sy (sz movespec)))
       (setq axismove nil)
       (setq axismove (sy (sz movespec))))))
  (t (setq axismove (sy (sz movespec)))))
(cond
  (axismove
   (if (not (numberp axismove))
       (setq tempflag t)
       (if (> zcurs zinit)
           (setq p (- zinit zbound (sz *starting-point*)))
           (setq p (- (+ zinit zbound) (sz *starting-point*))))))
  (setq axismove '(ap p))
  (setq newmovespec '(.@newmovespec
    ,axismove)))
(t (setq newmovespec
    '(.@newmovespec ,sz movespec)))))
(setq newmovespec '(.@newmovespec ,@(caddr
  movespec)))
(setq tempmove
  '(.sx tempmove ,newmovespec ,sz tempmove)))
; Some motions we can't just replace
(if tempflag (setq *tempmotion* tempmove)
  (self (pmove-motion strategy) tempmove)))

```