Order Number 1342004

# Planning stable in-hand reconfiguration of objects in two dimensions

Hunter, Jerry James, M.S.

The University of Arizona, 1990



U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# PLANNING STABLE IN-HAND RECONFIGURATION OF OBJECTS IN TWO DIMENSIONS

by

Jerry James Hunter

---



A Thesis Submitted to the Faculty of the

DEPARTMENT OF SYSTEMS AND INDUSTRIAL ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

WITH A MAJOR IN SYSTEMS ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 0

## STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: Jerry J. Hunter

## APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Jeffrey C. Trinkle — 8/18/90

Dr. J. C. Trinkle — Date

Professor of Systems and Industrial Engineering

ACKNOWLEDGEMENT

I would like to express my gratitude for the unending support of my parents, Loran and Jan, my brothers, Joel and Jason, and my closest friends, Eric, Matt, and Kerry. I also would like to thank Dr. Jeff Trinkle for his help and guidance. I would also like to thank the National Science Foundation for their support under the grant number MSS-8909678.

## TABLE OF CONTENTS

# LIST OF ILLUSTRATIONS

## ABSTRACT

This thesis presents a method to reconfigure objects in an articulated robotic hand, where reconfiguration is defined as moving an object from one orientation in the hand to another orientation. Several methods using object configuration rely on a stationary work area; a stationary work area is a work area which is completely known before any operations begin and which remains unchanged through the course of the operations. The work area is the primary obstacle with which the object and hand must interact. These methods assume that the object is stably grasped by the hand, and is moved around the work area in an assembly process.

In contrast, this thesis assumes that the hand is the primary obstacle. The hand is allowed to move its fingers. The object now reacts to the hand instead of the work area. Reconfiguration of the object within the hand is the goal. An additional criterion is that stability be maintained so that all operations are predictable and reversible. Models were developed that portrayed the mechanics and the contacts for frictionless manipulation in a plane. The model assumes that we have complete knowledge of the geometry and physical properties of the object. It also assumes that position control of the fingers is perfect. The model was then used in an algorithm to plan the motion that reconfigures the object. The algorithm creates data trees. Each node of the tree describes a new configuration of the object and can be attained by manipulating the configuration of the current node's parent. The data stored in each node is the contact formation of the node's configuration, where a contact formation includes all qualitatively similar configurations. Since the data in the tree is a set of contact formations, the data tree is called a contact formation(CF) tree. Contact formation trees used in conjunction with the simulated mechanics is a unique method for solving this problem; the solution we attain is an optimal reconfiguration plan.

# CHAPTER 1 INTRODUCTION

## 1.1 Problem Description

When a part is out of position in an assembly line, several options exist. In some systems, the part is simply discarded into a reject bin. In others, a human operator is required to re-adjust the part, and in others, the system fails. If a robot in the assembly line had a means to put the part in its correct position, then operation could continue normally.

Consider the action of grabbing a pencil or pen. Do we always pick up the pencil in the grasp that is used when writing? Or, do we sometimes pick up a pencil in a bad or useless configuration, and then reconfigure it so it may be used for writing? In general, when grabbing an object, rarely is the grasp we choose the grasp in which we intend to use the object. We automatically reconfigure the grasp to suit our needs. Currently robotic manipulation devices cannot do this.

The problem addressed in this thesis is that of in-hand reconfiguration. Assume that an object has already been stably placed in a hand. Assume further that the object is not in a the desired configuration. We propose a planning method that will allow a hand to reconfigure the object to a useful configuration without putting it down.

Consider an assembly process that mates two parts A and B(Fig. 1.1).

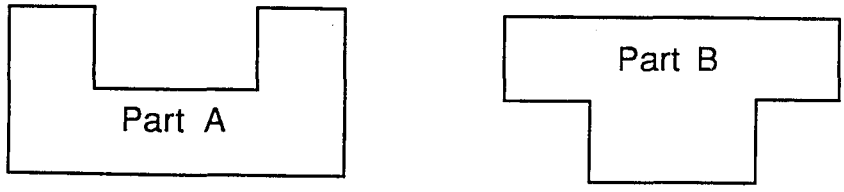



*Figure 1.1 Part A and Part B*

If the hand picks up part B in the configuration shown in Figure 1.2a, then it cannot immediately mate the parts. In Figure 1.2b, the hand has the part in a grasp that will allow the mating of A and B. The configuration in Figure 1.2a is the initial configuration and Figure 1.2b is the desired final configuration. This thesis was undertaken to develop an automatic planning algorithm that produces the joint trajectories such that the object will be manipulated from the initial configuration(Fig. 1.2a) to the final configuration(Fig. 1.2b).

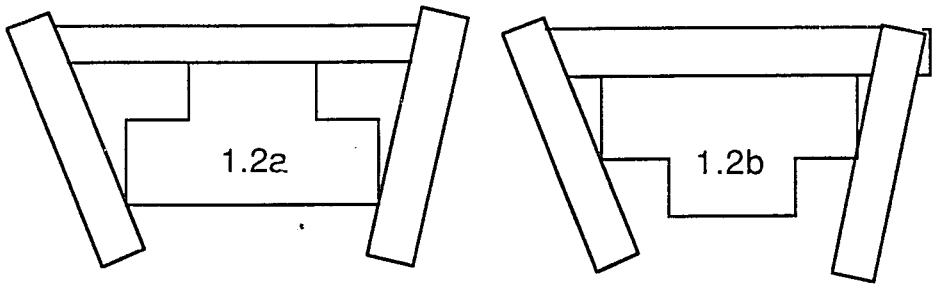



*Figure 1.2a) Bad Grasp 1.2b) Useful Grasp*

## 1.2 A Brief Outline of the Approach

The hand used in this thesis has a palm and two single-link fingers. The hand is also modeled in two dimensions. There is assumed to be no friction between the object and the hand. First the hand needs to grasp an object from a resting position and manipulate it into a stable grasp. A solution to this problem appears in Trinkle's dissertation[17]. He proposed to lift an object from its resting position on a table by determining various geometric properties of the object. A strategy was developed that eventually lifted the object from the table into an *enveloping* or *form closure* grasp. This type of grasp is the most stable because it allows free motion of the hand without affecting the object's position relative to the hand. On the other hand, a *force closure* grasp relies on gravity to keep the object in the hand; so if an object is in *force closure* and the hand is turned upside down, the object will become unstable, possibly falling from the hand altogether. Using

Trinkle's strategy to get the object into the hand is ideal for achieving an initial grasp for the problem addressed here.

Once the object is in the hand, a contact formation tree is built with the initial grasp configuration defining the root of the tree (*i.e.* the first node). A contact formation includes all qualitatively similar configurations, *e.g.* if a particular vertex of the object contacts a particular edge of a finger, all configurations where this vertex-edge contact is maintained belong to the same contact formation(see Desai[4]). The children of the root are determined by the physical and geometric properties of the object. If a contact formation occurs that is different from the root, then that contact formation is added to the tree. Since we are studying the frictionless case, stability is maintained, and all operations are assumed to be quasi-static (which implies that the system is conservative), then we can assume that all operations are reversible. Since all operations are reversible, a goal tree, which is a tree with the final grasp configuration as its root, can be used to reduce the computational operations necessary to obtain a solution.

Stability is an important issue. Without it, the object's motion may defy prediction. For example, if you were to grab a pencil, and the grasp were unstable, then the pencil would probably fall from your hand. A special case occurs when your palm is flat and facing up, the pencil would fall from your fingers, and land on your palm (rather than falling out of your hand completely), and the exact eventual resting place is at best difficult to predict. This is a problem that can be avoided if we require that stability be maintained during manipulation.

There are four parts to this thesis:

1. Model and simulate the mechanics of a user-defined, two-dimensional, frictionless, articulated hand,
2. Predict an object's motion and move an object subject to the motion of the hand and its fingers,
3. Plan a trajectory of joint angles to reconfigure an object from some initial grasp, to the final desired grasp without losing stability, and,
4. Optimize the planning strategy using data trees.

The chapters and the contents of each chapter briefly outlined are:

**Chapter 1** contains a brief introduction to the problem, a strategy to a solution, and a summary of previous work related to the topic of this thesis.

**Chapter 2** defines the nomenclature and mathematics necessary for this thesis.

**Chapter 3** gives the details of the simulation of the mechanics. The Object Motion Problem[16] is also addressed in this chapter.

**Chapter 4** shows a plan to move the object in the hand with the use of CF-trees. An initial tree and a goal tree are created and linked to create a final path to a solution.

**Chapter 5** shows the results and conclusions.

### 1.3 Previous Work

Consider the human hand as a structure of bone, tendon, and muscle, arranged in a straightforward manner. Despite its apparent simplicity, even the best robotic manipulators cannot hope to duplicate the immense versatility of the human hand. In Kerr's dissertation[9] he states:

> "Each finger mechanism as an independent unit is relatively simple. But it is precisely this lack of complexity in the individual fingers which makes the entire hand so intriguing. The human hand has roughly 22 degrees of freedom, most of which are independent. This large number of freedoms, taken with their high degree of independence is what allows for its multiplicity of function."[9]

The difficulty involved in creating and operating a robotic manipulator with 22 degrees of freedom outweighs the convenience of having such versatility. So in general, robotic researchers limit themselves to robotic hands that are much less versatile than their own hands. Since a robotic hand is less versatile, in many cases the hand must perform long complex maneuvers to accomplish what to our hands would be a relatively simple task. Using optimization shortens the number of operations that a hand must perform to complete a task.

Optimization is used in this thesis in several different forms. One is to find the most efficient strategy to reconfigure the object. On a smaller level, optimization is used to predict the motion of the object in the hand and to determine stability. The determination of stability is derived from the *minimum power principle.*

Peshkin and Sanderson[14] developed the *minimum power principle* which can be stated as:

> "The quasi-static approximation to the motion of a mechanical system is the solution to Newton's law $\mathbf{F} = \mathbf{ma}$ with the inertial term of ma ignored. Ignoring ma is only exact in trivial cases, but in many systems dissipative forces so overwhelm the inertial term that the quasi-static approximation is useful. The *minimum power principle* can be stated: A quasi-static system chooses that motion, from among all motions satisfying the constraints, which minimizes the instantaneous power."[14]

For example, consider a dry glass pushed along a table's surface. The minimum power principle agrees with our intuition that the glass will satisfy its kinematic constraints in the easiest way: "the way which minimizes the energy loss due to sliding friction." To apply this principle, the forces in the system must be normal forces, coulomb friction forces, or velocity independent forces.

The *object motion problem* derived by Trinkle and Paul[16], is a nonlinear program based on the *minimum power principle* which is solved to predict the motion of an object given the motions of the contacting bodies. To use the *object motion problem* at each time increment the object is assumed to be a quasi-static system subject to normal forces, coulomb friction forces, and forces that are independent of velocity. In the frictionless case there are two formulations of the object motion problem, the first is the velocity formulation, and the second is the force formulation. This thesis makes use of the force formulation. The object motion

problem differs from most previous work in its assumption that the hand is allowed to move, and the object will move in reaction to hand movement.

In past studies, the emphasis has been on moving the object through a work area that is completely known before any operations begin and remains unchanged through the course of the operations. This type of work area is referred to as a stationary work area. The object was assumed to be held firmly in the hand and was moved to a desired position and orientation. Solutions to the peg-in-hole have generally adopted this grasping assumption and are often referred to in grasping literature. The peg-in-hole problem is useful since a great deal of assembly requires fitting one part into another. However, most solutions to the peg-in-hole problem assume that the object is securely grasped. This thesis does not assume that the object is firmly in the hand. Instead, the object rests in the hand. The fingers are modeled as moving rigid boundaries that cause some motion in the object by contacting and pushing it. *Configuration space*[10] and *contact formations*[4] turn out to be quite useful when they are altered to include hand information.

Lozano-Pèrez and Wesley introduced the *configuration space*[10] to conveniently define the position and orientation of a system of bodies in space. The *configuration space* for a single object is defined as a six-by-one vector containing the Cartesian coordinates of a reference point on the object and its Euler angles measured with respect to a world frame. Brost[1] used *configuration space* to derive *configuration space obstacles.* He partitioned the *configuration space* into sub-sets that defined contact types. To get from one contact type to another, a path had to be plotted from one sub-set to the other. Both the work of Brost and Lozano-Pèrez and Wesley assume that the object is in a firm grasp and is moved around a fixed work space.

Desai[4] first introduced the term *contact formations.* A *contact formation* of an object describes the object's configuration in a more qualitative way than Lozano-Pèrez's *configuration.* Each *configuration* defines a specific position of a system while a *contact formation* includes all qualitatively similar configurations. In Figure 1.3, each contact is represented by a different *configuration* but both have the same *contact formation.* Desai's *contact formation space* is similar to

Brost's *configuration space obstacles*[1] in that it consisted of a partitioning of *configuration space* into sub-spaces. Each sub-space corresponded to a different contact formation. A desired configuration could be achieved by mapping a path through the CF space. But, in Desai's application, the object and hand did not move independently of each other. These ideas were used in this thesis, but the primary difference is that both the hand and the object move. In our application,we do not generate a single set in *object configuration space* to find a path through, but several overlapping sets in *joint space.* How these sets are computed and how paths are plotted through them will be discussed in detail in Chapter 4.

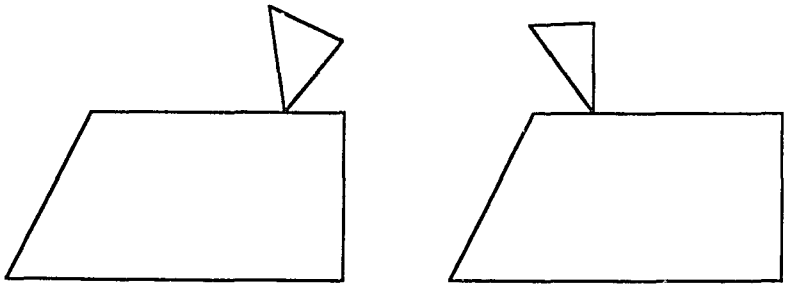


*Figure 1.3 Two Configurations in the Same Contact Formation*

Fearing[6,7] also did in-hand reconfiguration research. Fearing programmed a Stanford manipulator to "twirl" a baton. The "twirling" algorithm was open loop. His analysis considered only a two-dimensional cross section. Eventually the baton fell from the hand. However, the algorithm was a success since it demonstrated expected slip and roll. This thesis approaches the same type of problem, but rather than pre-determining a set of joint trajectories based on analysis, our algorithm determines a set of joint trajectories based on the physical and geometric properties of the object.

# CHAPTER 2 NOMENCLATURE AND KINEMATICS

## 2.1 Introduction

This thesis uses and is dependent upon kinematic transformations. These transformations simplify the data that maintain information on positions of the hand elements, velocities of the finger joints, and centers of gravity of hand and object elements. They are used to transform all hand and object elements to a common frame of reference. For a more detailed discussion of the sections in this chapter refer to Trinkle[17], Wolovich[18], or Paul[13].

## 2.2 Position Transformation

The transformation in three dimensions, relating the Cartesian coordinate frame **A** (or simply frame **A**) to frame **B**, can be represented by the homogeneous transformation matrix $U$, from Trinkle[17],

$$\mathbf{U} = \begin{pmatrix} \hat{n} & \hat{t} & \hat{o} & p \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.1}$$

The elements of the vectors $\hat{n}$, $\hat{t}$, and $\hat{o}$ are the direction cosines of the **B** axes relative to frame **A**, and $p$ is the vector defining the origin of frame **B** in reference to frame **A**(see Fig. 2.1). The bottom row is a row of scalars.

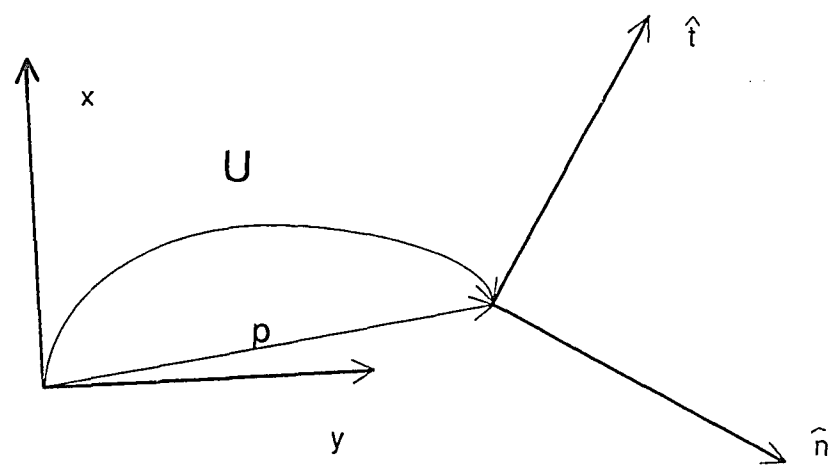



*Figure 2.1 Position Transformation*

The upper left three-by-three partition of both $U$ is an orthogonal rotation matrix , $R$,

$$\mathbf{R} = (\hat{n} \quad \hat{t} \quad \hat{o}). \tag{2.2}$$

Since we only consider planar manipulations, motion exists only in the $\hat{n}\hat{t}$ plane, so $\hat{o}$ is defined by, $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, which is the unit vector perpendicular to the plane of motion. A special case of $U$ is the two-dimensional Denavit-Hartenberg[5] transformation in which only the angle that the new frame is rotated, $\theta$, and the vector $p$ are needed. The DH transformation in two dimensions, $\mathbf{U}_{DH}$, is,

$$\mathbf{U}_{DH} = \begin{pmatrix} c\theta & -s\theta & 0 & b_i \\ s\theta & c\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.3}$$

where $b_i$ is the distance traveled along the $x$ axis after rotation (or $p = [b_i 00]^T$), and $s\theta$ and $c\theta$ are a short-hand notation for $sin\theta$ and $cos\theta$. Points in a particular frame will be represented by attaching a preceding superscript that indicates the coordinate frame. If there is no such superscript then the point is assumed to be in the world coordinate frame. The world coordinate frame is a Cartesian frame fixed to some reference point in the vicinity of one or more robots. For example, the point ${}^{B_0}V$ is a vertex represented with respect to frame $B_0$, and $V$ is the same vertex represented with respect to the world coordinate frame.

### 2.3 Force Transformation

The force transformation allows the applied forces and moments to be transformed from one frame to another. If $\mathbf{g}$ is a generalized force vector describing the applied force and moment in frame $\mathbf{B}$, then the vector is,

$${}^{\mathbf{B}}\mathbf{g} = \begin{pmatrix} {}^{\mathbf{B}}\mathbf{f} \\ {}^{\mathbf{B}}\mathbf{m} \end{pmatrix}_{6\times 1} \tag{2.4}$$

where $\mathbf{f}$ represents the applied force and $\mathbf{m}$ the applied moment. The force transformation matrix, $\mathbf{T_f(U)}$, is a $6 \times 6$ matrix and can be represented as follows,

$$\mathbf{T_f(U)} = \begin{pmatrix} \mathbf{R} & \mathbf{0} \\ \tilde{\mathbf{P}}\mathbf{R} & \mathbf{R} \end{pmatrix} \tag{2.5}$$

where $\tilde{\mathbf{P}}$ is the cross-product matrix of the vector $p$ given by,

$$\tilde{\mathbf{P}} = \begin{pmatrix} 0 & -p_z & p_y \\ p_z & 0 & -p_x \\ -p_y & p_x & 0 \end{pmatrix}. \tag{2.6}$$

In two dimensions, the generalized force vector has only three non-zero elements, $f_x$, $f_y$, and $m_z$. To transform the applied force $\mathbf{g}$ in frame $\mathbf{B}$ to the equivalent applied force in frame $\mathbf{A}$ the following equation is used,

$$^{\mathbf{A}}\mathbf{g} = \mathbf{T_f(U)}^{\mathbf{B}}\mathbf{g}. \tag{2.7}$$

The force transformation is useful in finding the wrench matrix which will be defined later in this chapter.

### 2.4 Velocity Transformation

The velocity transformation is similar to the force transformation. It is a $6 \times 6$ matrix that transforms the angular and linear velocities from one frame to another. Let the generalized velocity vector, $\mathbf{w}$ be defined as,

$$\mathbf{w} = \begin{pmatrix} \mathbf{v} \\ \omega \end{pmatrix}_{\mathbf{6\times 1}}. \tag{2.8}$$

where $\mathbf{v}$ is the linear velocity and $\omega$ is the angular velocity. In two dimensions, the velocity vector has only three non-zero elements, $v_x$, $v_y$, and $\omega_z$. To transform a velocity vector from frame $\mathbf{B}$ to frame $\mathbf{A}$ the following equation is used,

$$^{\mathbf{B}}\mathbf{w} = \mathbf{T_v(U)}^{\mathbf{A}}\mathbf{w}, \tag{2.9}$$

where

$$\mathbf{T_v(U)} = \begin{pmatrix} \mathbf{R^T} & \mathbf{R^T\tilde{P}^T} \\ \mathbf{0} & \mathbf{R^T} \end{pmatrix}. \tag{2.10}$$

The velocity transformation is useful in finding the grasp Jacobian matrix, which will also be defined later in this chapter.

### 2.5 Force and Velocity Transmission

All of the above transformations (homogeneous, force, and velocity) are general for the three-dimensional case. However, since only two-dimensional motion

is considered in our planning problem, the transformed forces and velocities are constrained to the proper dimensions. This is accomplished using the "*contact constraint matrix*" or "*the contact transmission matrix*", **E**, which will be defined later.

The physical properties of the hand and object determines the forces which can be transmitted from one to the other through contacts. Consider a frictionless case, only forces in the direction of the surface normals can be transmitted, as in Fig. 2.2.

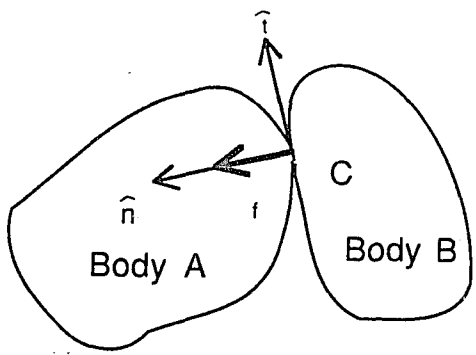



*Figure 2.2 Point Frictionless Contact*

The contact frame, **C**, is defined so that the origin of **C** is at the contact point, the $\hat{n}$ axis coincides with the contact normal, and the $\hat{t}$ axis coincides with the contact tangent. Since the contact above is only a point, no moments can be transmitted. So the transmitted force is,

$$^{\mathbf{C}}\mathbf{g} = \begin{pmatrix} -1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \mathbf{c}, \qquad where, \qquad \hat{n} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \tag{2.11}$$

and $c$ is the magnitude of the contact force and may only exert compressive loads(*i.e.* $c \geq 0$). For a general contact force and moment transmission method, we define an orthonormal contact basis, $\hat{\mathbf{i}}_\mathbf{i}$ for $i$ = 1,...,6, where $\hat{\mathbf{i}}_\mathbf{i}$ is the $i^{th}$ column

of a six-by-six identity matrix. The first three basis vectors represent unit forces in the $\hat{n}$, $\hat{t}$, and $\hat{o}$ directions, and the last three represent the unit moments about the $\hat{n}$, $\hat{t}$, and $\hat{o}$ axes.

Consider a point contact with friction, once again a moment cannot be transmitted through a point contact, so the transmitted force is,

$$^{C}\mathbf{g} = (\hat{i}_1 \quad \hat{i}_2 \quad \hat{i}_3)\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathbf{E}\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.12}$$

where the vectors $\hat{i}_1$, $\hat{i}_2$, and $\hat{i}_3$ are the columns of the *contact constraint matrix*, which is a $6 \times 3$ matrix, $\mathbf{E}$, the basis vectors that are not in the *constraint matrix* define the *contact freedom matrix*. The elements $c_1$, $c_2$, $c_3$ are the magnitudes of the forces in the directions of the vectors in $\mathbf{E}$. So, $c_1 \geq 0$ and $c_2$ and $c_3$ would be constrained by the friction model. Seven contact types and their constraint and freedom matrices are presented in Appendix A.

When modeling only two-dimensional motion, there are only three non-trivial elements of the vectors $\mathbf{g}$ and $\mathbf{w}$, these are obtained by pre-multiplying the vectors by $\mathbf{E_{2d}}$. For example,

$$\mathbf{g_{transmitted}} = \mathbf{E_{2d}}\mathbf{c} \tag{2.13}$$

where

$$\mathbf{E_{2d}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{2.14}$$

and $c$ is a vector of the magnitudes of the forces. This approach effectively zero's the out-of-plane elements. So only the forces in the $xy$ plane and the moment around the $z$ axis are considered.

## 2.6 Equations of Force and Velocity

This section describes the notation of various transformation matrices as well as how these transformations are used to relate coordinate frames, and to develop the wrench matrix and the Jacobian matrix.

The following transformations describe different aspects of the hand(fig 2.3). They are:

1) the $B$ matrices, which describe the centers of gravity of the bodies relative to the world frame, where $B_0$ is the center of gravity of the object, $B_{0,0}$ is the center of gravity of the palm, and $B_{i,j}$ is the center of gravity of finger $i$, link $j$.
2) the $A$ matrices, which describes the center of gravity of the palm to the base frame of finger $i$ with $A_{i,0}$, and from one joint $j-1$ on finger $i$ to the next joint $j$ on the same finger $i$ with $A_{i,j}$.
3) the $D$ matrices, which describe the centers of gravity on each link to that link's base joint, and,
4) the $F$ matrices, which describe the object's center of gravity with respect to the contact frames.

There are also a few frames of reference that prove to be quite useful. These are,

1) the $T$ frames, these describe the frames at the joints themselves. The frame $T_{i,0}$ is fixed to the palm at the base joint of the first finger. The $A$ matrix defines one $T$ frame with respect to the next $T$ frame by,

$$T_{i,j} = T_{i,j-1}A_{i,j} = B_{0,0}\prod_{k=0}^{j} A_{i,k} \qquad \begin{matrix} i = 0,1,\cdots,n_{fingers} \\ j = 0,1,\cdots,n_{links} \end{matrix} \tag{2.15}$$

where

$$T_{0,-1} = B_{0,0} \tag{2.16}$$

2) and the $C$ frame, or the contact frame. The $k^{th}$ contact frame on the $i,j^{th}$ link is,

$$C_{i,j,k} = T_{i,j-1}D_{i,j,k}. \tag{2.18}$$

The contact frame relative to the object frame $B_0$ is,

$$C_{i,j,k} = B_0F_{i,j,k}. \tag{2.19}$$

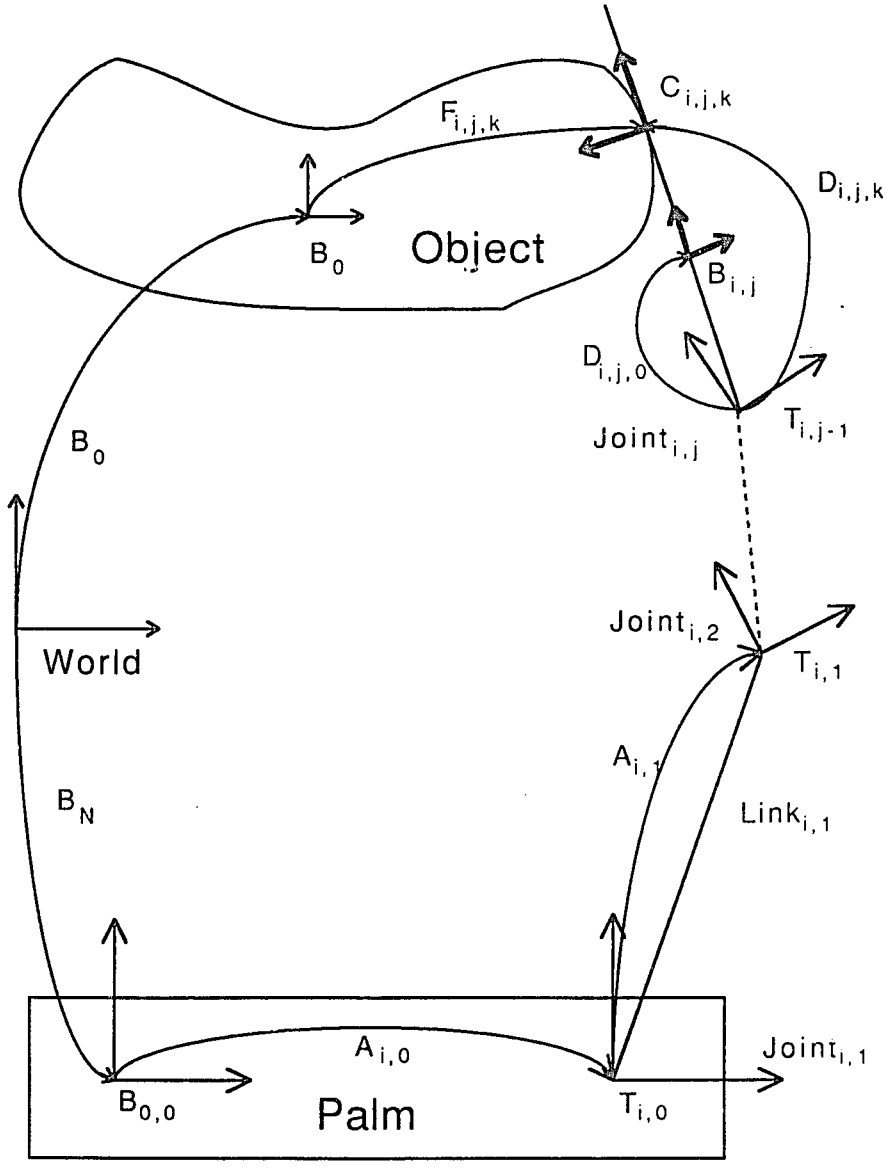



*Figure 2.3 Frames and Transformations*

## 2.7 Object Equilibrium and the Wrench Matrix

To generate the wrench matrix, the equations of equilibrium need to be formulated. Consider the external force, $g_{ext}$, acting on the object. The external force needs to be balanced by the forces transmitted by the hand to the object through the contacts. The equation that describes equilibrium is,

$$\sum_{i,j,k} \mathbf{T}_f(\mathbf{F}_{i,j,k}) {}^{C}\mathbf{g}_{i,j,k} = -{}^{B_0}\mathbf{g}_{ext} \tag{2.20}$$

where $\mathbf{T}_f(\mathbf{F}_{i,j,k})$ transforms the $ijk^{th}$ contact force, ${}^{C}\mathbf{g}_{i,j,k}$, to the frame $B_0$, where $i$, $j$, $k$ is the contact $k$ on the $i^{th}$ finger and the $j^{th}$ link.

Substituting the transmission matrix and the wrench intensity vector in for ${}^{C}\mathbf{g}_{i,j,k}$ yields,

$$\sum_{i,j,k} \mathbf{T}_f(\mathbf{F}_{i,j,k})\mathbf{E}_{i,j,k}\mathbf{c}_{i,j,k} = -{}^{B_0}\mathbf{g}_{ext}. \tag{2.21}$$

The wrench matrix, ${}^{B_0}\mathbf{W}_{i,j,k}$ of the $i,j,k^{th}$ contact is defined by $\mathbf{T}_f(\mathbf{F}_{i,j,k})\mathbf{E}_{i,j,k}$, therefore,

$$\sum_{i,j,k} {}^{B_0}\mathbf{W}_{i,j,k}\mathbf{c}_{i,j,k} = -{}^{B_0}\mathbf{g}_{ext}. \tag{2.22}$$

The indicated summations result in the global equilibrium equations,

$${}^{B_0}\mathbf{W}c = -{}^{B_0}\mathbf{g}_{ext}, \tag{2.23}$$

where

$${}^{B_0}\mathbf{W} = \left[{}^{B_0}\mathbf{W}_0 \cdots {}^{B_0}\mathbf{W}_{n_c}\right]. \tag{2.24}$$

The product of the wrench matrix and the wrench intensity vector is a compact way to represent the sum of the contact forces and moments applied to the object. The wrench matrix ${}^{\mathbf{B_0}}\mathbf{W}$ is really one matrix containing several sub wrench matrices. Each force or contact, $i$, exerted by the hand on the object, has a corresponding wrench. This wrench describes the directional cosines of the force, as well as the associated orthogonal moment arms. The magnitudes of the

forces are described by the wrench intensity vector $\mathbf{c}_{i,j,k}$. For a point contact with friction, $\mathbf{W}_{i,j,k}$, is defined as follows:

$$\mathbf{W}_{i,j,k} = \begin{pmatrix} \hat{n}_{i,j,k} & \hat{t}_{i,j,k} & \hat{o}_{i,j,k} \\ \mathbf{r}_{i,j,k} \times \hat{n}_{i,j,k} & \mathbf{r}_{i,j,k} \times \hat{t}_{i,j,k} & \mathbf{r}_{i,j,k} \times \hat{o}_{i,j,k} \end{pmatrix} \tag{2.25}$$

In $\mathbf{W}_{i,j,k}$, $\mathbf{r}_{i,j,k}$ is the vector describing the position of the $ijk^{th}$ contact point, $\hat{n}_{i,j,k}$ is the contact unit normal directed into the object, and $\hat{t}_{i,j,k}$ and $\hat{o}_{i,j,k}$ are two orthogonal unit vectors that define the friction plane. An example of a two dimensional frictionless object and its wrench matrix is given in Figure 2.4.

$$\mathbf{W} = \begin{pmatrix} n_{1x} & n_{2x} & n_{3x} \\ n_{1y} & n_{2y} & n_{3y} \\ 0 & r_{2x}n_{2y} - r_{2y}n_{2x} & r_{3x}n_{3y} - r_{3y}n_{3x} \end{pmatrix}$$

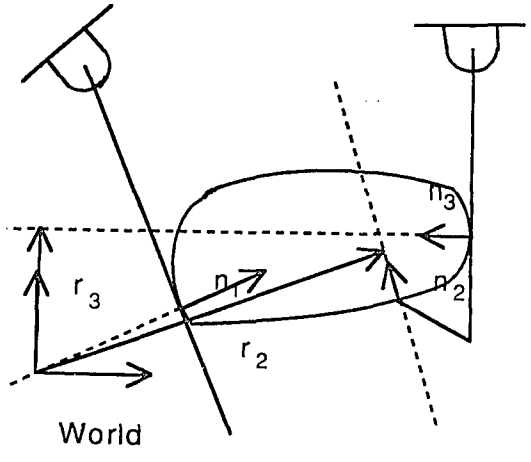



*Figure 2.4 Object and Forces*

## 2.8 Hand Contact Velocities and the Jacobian Matrix

The Jacobian matrix can be derived by taking partial derivatives of position and orientation with respect to the joint angles $\theta_{i,j}$, or the Jacobian can be derived from the velocity transformation matrix. It is much easier computationally to use

the velocity transformation since the $\mathbf{T}_v$ matrix is related to contact forces in the following fashion,

$$\mathbf{T}_f = \mathbf{T}_v^T. \tag{2.26}$$

The hand in this thesis, has only revolute joints, therefore, each link $j$ rotates around the joint frame $T_{i,j-1}$ at a joint velocity of $\dot{\theta}_{i,j}$. The equation which gives the velocity of the $k^{th}$ contact on the $i^{th}$ finger relative to $T_{i,j-1}$ in frame $C_{i,j,k}$ is,

$${}^{C}\mathbf{w}_{hand,i,j,k} = \mathbf{T}_v(\mathbf{D}_{i,j,k})^{T_{i,j-1}}\mathbf{w}_{hand,i,j} \tag{2.27}$$

where

$${}^{T_{i,j-1}}\mathbf{w}_{hand,i,j} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \dot{\theta}_{i,j} \end{pmatrix} \tag{2.28}$$

Since $\dot{\theta}_{i,j}$ is the angular velocity of the $i,j^{th}$ link, it is the same for all the contacts on the $i,j^{th}$ link. The $k$ subscript has been dropped. So, equation (2.27) can be obtained by multiplying the sixth column of the velocity transformation matrix by $\dot{\theta}_{i,j}$,

$${}^{C}\mathbf{w}_{hand,i,j,k} = [\mathbf{T}_v(\mathbf{D}_{i,j,k})]_{6^{th}col}\dot{\theta}_{i,j} \tag{2.29}$$

The velocity of the contact relative to the palm, in frame $C_{i,j,k}$ is found by adding the velocities of the contact due to each joint separately.

$${}^{C}\mathbf{w}_{hand,i,j,k} = \sum_{m=1}^{j} [\mathbf{T}_v(\mathbf{D}_{i,j,k,m})]_{6^{th}col}\dot{\theta}_{i,m} \tag{2.30}$$

In matrix form this is,

$${}^{C}\mathbf{w}_{hand,i,j,k} = {}^{C}\mathbf{J}_{i,j,k}\dot{\theta}_{i,j} \tag{2.31}$$

where $\mathbf{J_{i,j,k}}$ is the Jacobian at the $k^{th}$ contact on the $j^{th}$ link on the $i^{th}$ finger. The global Jacobian is a block diagonal matrix whose diagonal elements are the $J_{i,j,k}$ matrices. The global Jacobian is not used in this thesis, but its calculations can be seen in Trinkle[17].

# Chapter 3 Description and Simulation of Mechanics.

## 3.1 Introduction

The model we use allows the hand to be any shape or size, and have singly- or multiply-linked fingers. However, the specific hand simulated in this thesis has a single palm and two single-link fingers. The mechanics accurately describe the motion for any user-defined hand. The grasp planner in Chapter 4 precludes the use of multiply-linked fingers at this point, although, with a fair amount of modification, multiply-linked fingers could also be used.

## 3.2 Representation of Polygons and Hand Motion

Each element of the hand and the object is represented by a polygon. These polygons may be of any size or shape, as long as they are convex. In the program a polygon is represented by a pointer to a data structure. The structure need only contain two basic elements to retain all the information necessary to reproduce, or do mathematical operations on an object. These elements are the number of vertices and the location of the vertices with respect to the object's center of gravity. For convenience the vertices are taken from the polygons in a counter- clockwise order and stored in the data structure. For example, a square of side 20 would be represented as follows:

```
square→vertex_number = 4;
square→vertex[1].x = 10;
square→vertex[1].y = 10;
  ⋮       ⋮          ⋮
square→vertex[4].x = 10;
square→vertex[4].y = -10;
```

where "square" is the variable name of the data structure. In this example, the object's center of gravity is at the point (0,0). Data element "square" is a polygon_pointer(p_ptr) data type. First the polygons must be transformed from their base coordinate frame to the world coordinate frame, and then they must be linked in such a way that they simulate the motion of a hand. Once the palm polygon is

recorded in a data structure, it is transformed to a chosen world frame using the matrix $\mathbf{B}_{0,0}$, where,

$$\mathbf{B}_{0,0} = \begin{pmatrix} c\theta & -s\theta & x \\ s\theta & c\theta & y \\ 0 & 0 & 1 \end{pmatrix}, \tag{3.1}$$

and $\theta$, $x$, and $y$, are, respectively, the orientation, and position of the palm in the plane. Once the palm is in place, the fingers are defined relative to the palm. This will allow redefinition of the entire hand simply by redefining the palm. Using the notation described in Chapter 2, each link, $i$, will be transformed to its joint on the palm by the translation matrix $\mathbf{A}_{i,0}$, where,

$$\mathbf{A}_{i,0} = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

and $x$ and $y$ are the Cartesian coordinates with respect to the palm frame, of the joint to which the links will be attached. Then the link will be transformed by the rotation matrix $\mathbf{A}_{i,1}$, where,

$$\mathbf{A}_{i,1} = \begin{pmatrix} c\theta_i & -s\theta_i & 0 \\ s\theta_i & c\theta_i & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

and $\theta_i$ is the angle of finger $i$ with respect to the palm.

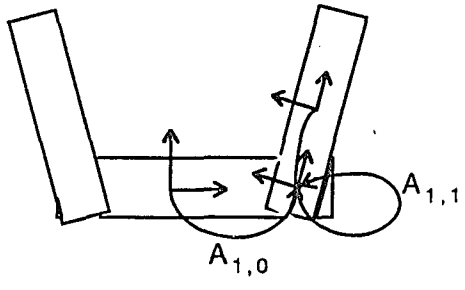



*Figure 3.1 Hand and Transformations*

Finally, the finger must be transformed from its internal reference frame to the joint connection. This is accomplished using the matrix $\mathbf{D}_{i,j}$ where $i$ is the finger and $j$ is the link(Fig 3.1). Every time either a finger or the palm is moved, the values of the $\theta_i$'s and the palm's positional values with respect to the world frame, $(x, y)$, must be updated, and the transformations re-multiplied. Once the transformations have been made, the vertices of each polygon are now in world frame coordinates. These world frame polygons are saved in a data structure conveniently named "hand."

The hand data structure is represented as follows:

hand → finger[1] → link[1] → vertex_number = n;
→ vertex[1].x = x;
⋮ ⋮
→ vertex[4].y = y;
hand → finger[2] → link[1] → vertex_number = n;
→ vertex[1].x = x;
⋮ ⋮
→ vertex[4].y = y;
hand → palm → vertex_number = n;
→ vertex[1].x = x;
⋮ ⋮
→ vertex[4].y = y;

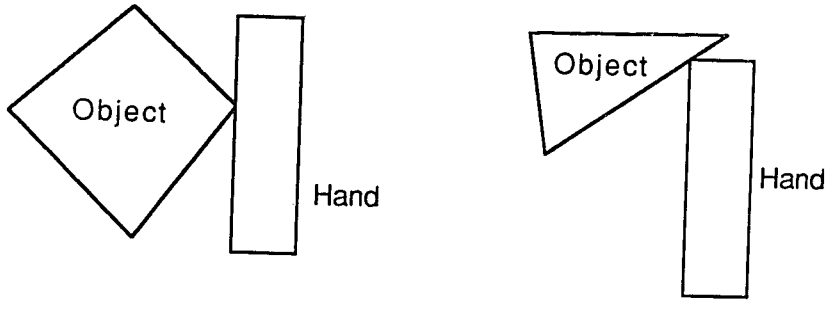



*Figure 3.2 a) "Case-One" b) "Case-Two" Contact*

### 3.3 Simulation of Contacts

Two types of legal contact may occur. The first type of contact, a "case-one" contact, occurs when an object vertex contacts an edge on any one of the hand polygons(Fig. 3.2a). The second type of contact, a "case-two" contact, occurs when a vertex of a hand polygon makes contact with an edge of the object(Fig. 3.2b). These two types of contact may be combined into several sub-types and will be discussed in Chapter 4.

A contact is assumed to occur when two polygons are within $\varepsilon$ distance of one another. To find the contacts between the object and hand polygons, the object and hand must be in the world reference frame. Once this is done, each hand polygon is tested for contact with the object in the following fashion. The distance from the vertex $j$ of the object to the line connecting vertex $i$ to vertex $i+1$ of a hand polygon is found(Fig 3.3).

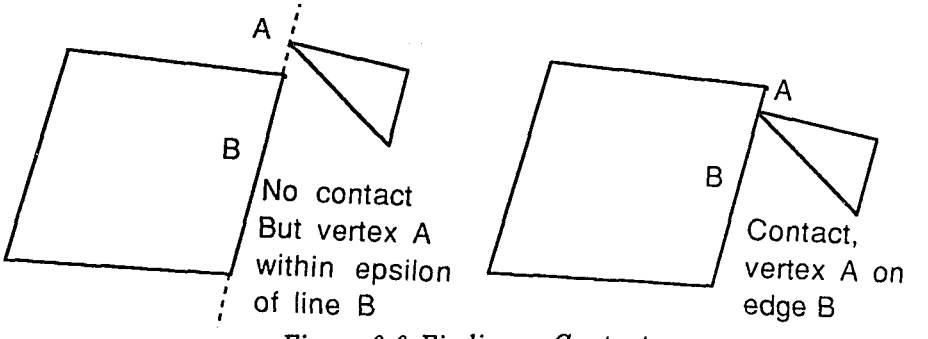



*Figure 3.3 Finding a Contact*

If this distance is smaller than the chosen epsilon, then the point $j$ of the object may contact the edge $i$ of the hand polygon being tested. The object vertex may contact the line, but not the hand edge, so object vertex $j$ must also be tested to determine if it is between the hand polygon vertex $i$ and vertex $i+1$. If the distance and the "between" test are both satisfied, then the object has a contact point at $j$. The test also needs to be done for hand vertices to object edges. The

algorithm is as follows:

**Given** the object polygon and a hand polygon;

**Create** the matrix $O$ of size $\#object\_vertices \times \#hand\_polygon\_vertices$
where the element $o_{i,j}$ gives the distance between
object vertex $i$ and the hand edge $j$.

**Create** the matrix $H$ of size $\#hand\_polygon\_vertices \times \#object\_vertices$
where the element $h_{i,j}$ gives the distance between
hand vertex $i$ and the object edge $j$.

**for** $i = 1$ to $\#hand\_polygon\_vertices$
**for** $j = 1$ to $\#object\_vertices$
{
**if** ( $|o_{j,i}| \leq \varepsilon$) **then**
**if** (object vertex $j$ between hand polygon vertices $i$ and $i+1$) **then**
add "case-one" contact to contact information vector
**if** ( $|h_{i,j}| \leq \varepsilon$) **then**
**if** (hand polygon vertex $j$ between object vertices $i$ and $i+1$) **then**
add "case-two" contact to contact information vector
}

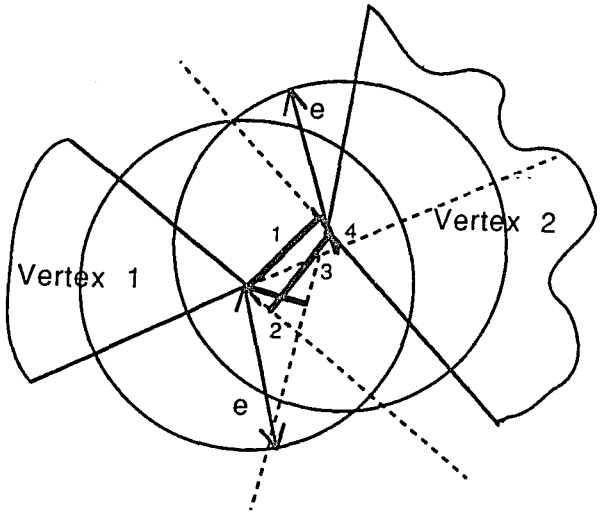



*Figure 3.4 Two Vertices in $\varepsilon$ Distance*

This algorithm is sufficient in most cases. However, there exists a case where it fails and needs assistance. This case occurs when the distance between a vertex of a hand polygon and a vertex of the object is less then the chosen $\varepsilon$ (Fig. 3.4). Instead of indicating a single contact, the algorithm produces four.

Each vertex is defined by the intersection of two lines. When a vertex is within $\varepsilon$ distance of another vertex, it is within $\varepsilon$ distance of the two lines that define that other vertex; so the vertex is close enough to each line that can have an edge contact with both. The other vertex will also show two edge contacts giving a total of four contacts instead of one. A solution to this problem is to create an additional algorithm that determines which of the four contacts is the correct contact.

Since each line defining the object polygon is a directed line going counter-clockwise around the object, a point inside the object gives a negative result when substituted into each of the polygon's line equations. At a vertex, a point to the left of both lines that define that vertex is a point inside the object, and a point to the right of both these lines is outside the object, where the right of a line generates a positive value and the left of a line generates a negative value.

Given an object vertex that is in contact with a hand vertex, a point-to-point contact region can be divided into four sub-regions. Call the lines that define the object vertex line $i$ and line $i+1$. Region 1 is the area to the right of line $i$ and to the left of line $i+1$. Region 2 is the area to the right of both line $i$ and line $i+1$. Region 3 is the area to the left of line $i$ and to the right of line $i+1$ and finally region 4 is the area to the left of both line $i$ and line $i+1$(Fig 3.5).

Assuming that a point-to-point contact has occurred, the following heuristic is used to select the correct contact.

1) If the hand vertex is in region 1, then the contact is with line $i$.
2) If the hand vertex is in region 3, then the contact is with line $i+1$.
3) If the hand vertex is in region 2, then determine the distance from the hand vertex to both line $i$ and line $i+1$. Then if the distance to line $i$ is smaller, then the contact is with line $i+1$, else if the distance to line $i+1$ is smaller then the contact is with line $i$.

4) If the hand vertex is in region 4, then determine the distance from the hand vertex to both line $i$ and line $i+1$. If the distance to line $i+1$ is smaller, then the contact is with line $i+1$, else if the distance to line $i$ is smaller then the contact is with line $i$.

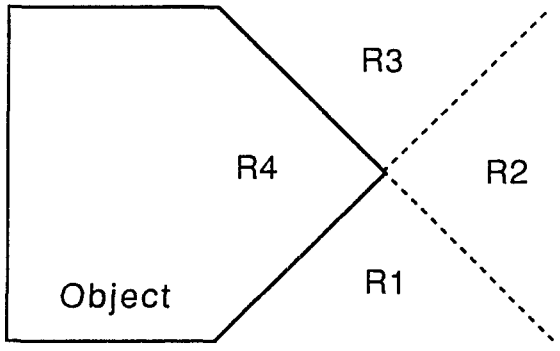



*Figure 3.5 Point-To-Point Contact Region*

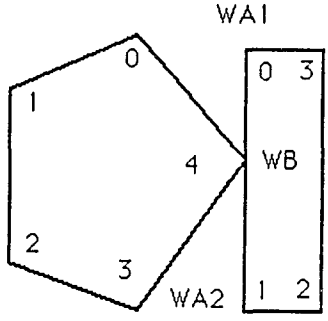
Usually these guidelines find the correct contact, but there are extreme instances where it fails. A solution is to lower the value of $\varepsilon$ so that a point-to-point contact will practically never happen. However, if $\varepsilon$ is too small more difficult problems occur with object motion. These problems are discussed later in this Chapter.

### 3.4 The Contact Information Vector

The contact information vector stores all the information relevant to determining the *contact formations*. The CI vector is a vector of structures. There are three times as many vector elements as there are object vertices. The vector is:

ci→v[1]
⋮
ci→v[2n]
ci→e[1]
⋮
ci→e[n],

where n is the number of object vertices. In Figure 3.6 where object vertex 4 contacts finger 1, the information would be:




v[4].contact_with = 1;
v[4].contact = 1;
v[4].contact_# = 1;
v[4].hvertex = 0;
v[4].overtex = 4;
v[4].A1 = {x = -1 , y = 1 }
v[4].A2 = {x = -1 , y = -1 }
v[4].B = {x = 0 , y = 3 }.
v[4].WA1 = {x = 0 , y = 2 }
v[4].WA2 = {x = 0 , y = 0 };
v[4].WB = {x = 0 , y = 1 }.

*Figure 3.6 Contact Information*

Each element of the vector is a data structure containing the following:

1) contact - indicates whether or not a contact with this element has occurred.
2) contact_with - indicates with which of the hand polygons there is a contact.
3) hvertex - either the number of the hand vertex if a case-two contact, or the number of the hand edge for case-one contacts.
4) contact_number - this is the number of the contact that has to do with the contact normals and will be explained later.

5) overtex - either the number of the object vertex if a case-one contact, or the number of the object edge for case-two contacts.
6) A1 and A2 - contain the x and y coordinates for the two points that define the edge. Both are in the polygon's base frame coordinates.
7) B - the x and y coordinates of the contacting vertex in that polygons base frame coordinates.
8) WA1, WA2, and WB - the world frame coordinates for the points defined in 6 and 7.

If there is a case-one contact, then this means that there is a contact between a vertex of the object and an edge on the hand, so contact information would be stored in $v[i]$ where $i$ is the number of the object vertex contacted. If there is a case-two contact, then the contact is between a hand polygon vertex and an object edge, so the information would be stored in $e[i]$ where $i$ is the number of the edge on the object contacted. The reason that there are $2n$ of the "$v$" elements, is that there exists the possibility that a given vertex will contact two hand polygons, in which case the information needs to be saved separately.

### 3.5 Simulation of Dexterous Manipulation

Assuming the object is in a stable configuration, and the contacts have been found, the next step is to determine how the object reacts to finger motion. First we determine which contacts will be maintained during motion. The algorithm that determines which of the contacts will be maintained will be discussed later. For now assume that there are at least three contacts between the hand and the object; there are exactly three that will be maintained for a single increment in quasi-static manipulation. The equations that relate the object orientation to the position of the hand are called the contact geometric equations from Trinkle and Paul[16]. The contact geometric equations attempt to keep a vertex of one polygon in contact with an edge of another. So if one polygon is moved, the equations give $x$, $y$, and $\theta$ of the other polygon so that a contact between the vertex and edge is maintained. They are derived for both case-one and case-two contacts in the following two sections.

Case 1: In this case object vertex ${}^{o}V_j$ (where the $o$ superscript refers to an object vertex) will maintain contact with hand polygon edge $k$. The hand polygon edge $k$ is defined by the two vertices ${}^{h}V_k$ and ${}^{h}V_{k+1}$ (the $h$ superscript refers to a hand polygon vertex) in the world reference frame where:

$$ {}^{h}V_k = (x_k, y_k), \qquad {}^{h}V_{k+1} = (x_{k+1}, y_{k+1}) $$

The line through these two points may be written as:

$$ -x\delta_y + y\delta_x = y_k\delta_x - x_k\delta_y \tag{3.4} $$

where

$$ \delta_x \doteq x_k - x_{k+1} $$

$$ \delta_y \doteq y_k - y_{k+1} $$

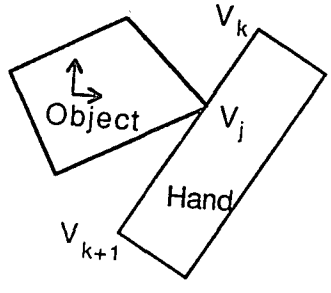



*Figure 3.7 "Case-One" Contact*

By substituting into equation 3.4, we have:

$$ -x(y_k - y_{k+1}) + y(x_k - x_{k+1}) - x_k y_{k+1} + x_{k+1} y_k = 0 \tag{3.5} $$

or

$$ Ax + By + C = 0 = f(x, y) \tag{3.6} $$

where

$$A = y_{k+1} - y_k \tag{3.6a}$$

$$B = x_k - x_{k+1} \tag{3.6b}$$

$$C = x_{k+1}y_k - x_k y_{k+1} \tag{3.6c}$$

Since ${}^hV_k$ and ${}^hV_{k+1}$ are known because the position of the hand is known, $A$, $B$, and $C$ are constant. We put $x$ and $y$ in Eq(3.6) in terms of $x_{obj}$, $y_{obj}$, and $\theta_{obj}$, where the objects center of gravity in the world reference frame is at $(x_{obj}, y_{obj})$, multiplying by $B_0$ we get,

$${}^oV_j = B_0\,{}^{B_0}{}^oV_j = \begin{pmatrix} c\theta_{obj} & -s\theta_{obj} & x_{obj} \\ s\theta_{obj} & c\theta_{obj} & y_{obj} \\ 0 & 0 & 1 \end{pmatrix}^{B_0} \begin{pmatrix} x_j \\ y_j \\ 1 \end{pmatrix}. \tag{3.7}$$

This yields

$$x_j = {}^{B_0}x_j c\theta_{obj} - {}^{B_0}y_j s\theta_{obj} + x_{obj} = x_j(x_{obj}, y_{obj}, \theta_{obj}), \tag{3.7a}$$

$$y_j = {}^{B_0}x_j s\theta_{obj} + {}^{B_0}y_j c\theta_{obj} + y_{obj} = y_j(x_{obj}, y_{obj}, \theta_{obj}), \tag{3.7b}$$

where ${}^{B_0}x_j$ and ${}^{B_0}y_j$ are known, and $x_{obj}$, $y_{obj}$, and $\theta_{obj}$ describe the unknown position and orientation of the object. Since ${}^oV_j$ must lie on the edge defined by Eq(3.6), Eq(3.7a) and Eq(3.7b) may be substituted into Eq(3.6) yielding,

$$Ax_j(x_{obj}, y_{obj}, \theta_{obj}) + By_j(x_{obj}, y_{obj}, \theta_{obj}) + C = 0 \tag{3.8}$$

For each instance of a case-one contact that will be maintained, Eq(3.8) must be solved.

**Case 2:** In this case, a hand polygon vertex ${}^hV_j$ contacts an object edge $k$. The edge $k$ is defined by the two points ${}^oV_k$ and ${}^oV_{k+1}$ in the world reference frame where:

$${}^oV_k = (x_k, y_k), \qquad {}^oV_{k+1} = (x_{k+1}, y_{k+1}).$$

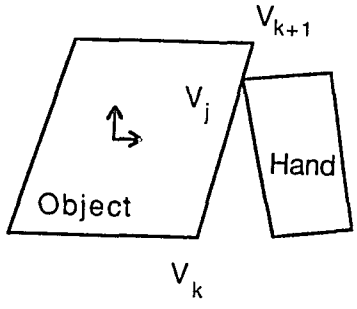



*Figure 3.8 "Case-Two" Contact*

Once again Eq(3.5) will be used, the difference is that ${}^{h}V_j$ is known through hand kinematics, ${}^{o}V_k$ and ${}^{o}V_{k+1}$ are now the unknowns yielding,

$$-x(y_k - y_{k+1}) + y(x_k - x_{k+1}) - x_k y_{k+1} + x_{k+1} y_k = 0 \tag{3.5}$$

The vertices ${}^{o}V_k$ and ${}^{o}V_{k+1}$ can be found with respect to $x_{obj}$, $y_{obj}$, and $\theta_{obj}$, in the same fashion as case 1.

$$({}^{o}V_k \quad {}^{o}V_{k+1}) = B_0 {}^{B_0}(V_k \quad V_{k+1}) = \begin{pmatrix} c\theta_{obj} & -s\theta_{obj} & x_{obj} \\ s\theta_{obj} & c\theta_{obj} & y_{obj} \\ 0 & 0 & 1 \end{pmatrix}^{B_0} \begin{pmatrix} x_k & x_{k+1} \\ y_k & y_{k+1} \\ 1 & 1 \end{pmatrix} \tag{3.9}$$

This yields,

$$x_k = {}^{B_0}x_k c\theta_{obj} - {}^{B_0}y_k s\theta_{obj} + x_{obj} = x_k(x_{obj}, y_{obj}, \theta_{obj}) \tag{3.9a}$$

$$y_k = {}^{B_0}x_k s\theta_{obj} + {}^{B_0}y_k c\theta_{obj} + y_{obj} = y_k(x_{obj}, y_{obj}, \theta_{obj}) \tag{3.9b}$$

$$x_{k+1} = {}^{B_0}x_{k+1} c\theta_{obj} - {}^{B_0}y_{k+1} s\theta_{obj} + x_{obj} = x_{k+1}(x_{obj}, y_{obj}, \theta_{obj}) \tag{3.9c}$$

$$y_{k+1} = {}^{B_0}x_{k+1} s\theta_{obj} + {}^{B_0}y_{k+1} c\theta_{obj} + y_{obj} = y_{k+1}(x_{obj}, y_{obj}, \theta_{obj}) \tag{3.9d}$$

Now $x_k$, $y_k$, $x_{k+1}$, $y_{k+1}$ can be substituted into equation(3.5). Since ${}^{h}V_j$ is part of the hand, both $x_j$ and $y_j$ are known. The substitution gives an equation in the three unknowns, $x_{obj}$, $y_{obj}$, and $\theta_{obj}$.

Each contact results in one equation in three unknowns. Simultaneously solving the equations gives the position and orientation of the object in reaction to the motion of the hand. Thus at least three contacts are needed to find a solution. Once the contacts to be maintained have been determined, the equations are solved with an iterative routine.

### 3.6 Stability Criteria

Stable object motion is of primary interest. Object motion is stable when the object can move while satisfying the equations of static equilibrium at every instant of time. The two types of stability considered are:

1. form closure, and,
2. force closure, Mason and Salisbury[12].

Form closure, or an enveloping grasp, provides the most secure grasp. To get form closure in the frictionless case, there must be at least four contacts. Since we have only three hand polygons(which are all convex), there must be two contacts on one polygon forming an edge contact and one contact on each of the remaining hand polygons. The conditions for a form closure grasp are:

$$\pi \geq \phi_1 - \phi_2 \geq 0 \tag{3.10}$$

where $\phi_1$ and $\phi_2$ are the normal angles of the two contacts that are not on the same edge. Also, the intersection of these two force normals must lie between the parallel normal forces that define the edge contact(Fig. 3.9).

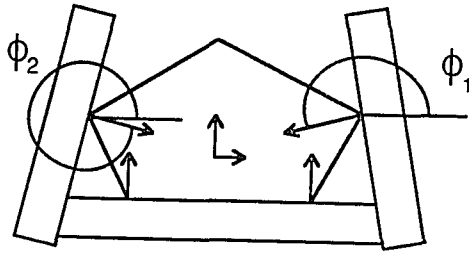



*Figure 3.9 Form Closure*

When an object is in form closure, gravity plays no part in its stability, therefore, the hand could be turned upside down and still maintain stability. So form closure is the preferred grasp condition for the simulation.

The second type of stable grasp, is a force closure grasp. When the palm is facing upward, the conditions for force closure are:

1) for four or more contacts,

$$\phi_1 - \phi_2 < 0 \tag{3.11}$$

where $\phi_i$ is the normal of the contact with finger $i$. Also the lifting phase plane(LPP)-triangle must contain the origin, where the vertices of the LPP-triangle are defined by $\{cos\phi_i, d_i\}$ for $i$= 1 to (# of contacts), and $d_i$ is the moment arm to the object's center of gravity, Trinkle *et al.*[15], and

2) for three contacts,

$$\mathbf{c} = -\mathbf{W}^{-1}\mathbf{g}_{ext} > \mathbf{0}. \tag{3.12}$$

where the inequality applies element-by-element. In force closure, gravity helps maintain stability; as the hand moves, it is possible that the grasp would become unstable. To maintain stability two types of finger motion are used.

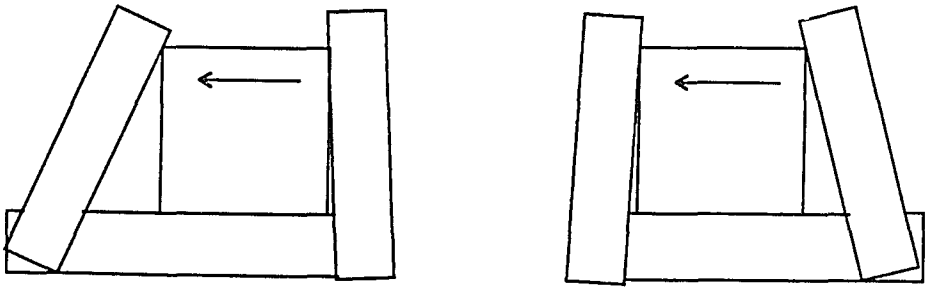


*Figure 3.10 Compliant Motion*

The two types of finger motion are:

1) compliant motion, which is the motion necessary to maintain a form closure grasp, Mason and Salisbury[12], and

2) independent motion, which is the motion used when the object is in force closure.

Compliant motion can be described as finger one rotating $\theta_1$ which moves the finger toward the object. The object is pushed by the finger in the direction of $\theta_1$. Finger two complies with the motion allowing the object to maintain an edge contact. Compliance is simulated by placing the finger(Appendix B) on the object after the object has moved. So the hand/object system with five degrees of freedom is reduced to a single degree of freedom. This is necessary to maintain form closure.

Independent motion simply allows either finger to move in any direction at any rate completely independent of one another.

# CHAPTER 4 PLANNING

## 4.1 Introduction

Since the problem is to manipulate an object from an initial configuration to a final configuration, a representation of configuration is needed. One such representation is Lozano-Pèrez and Wesley's[10] idea of object configuration. In three-space this configuration consists of a six-by-one vector. The first three elements of the vector are the objects center of gravity in world coordinates, and the next three elements are the Euler angles which define the orientation of the object's fixed-frame with respect to the world frame. The space of all possible configurations is called the *configuration space.* The configuration vector of an object in a plane is a three-by-one vector.

A *contact formation* is a concise way to represent sub-spaces in the configuration space and will be discussed in Section 4.2. Later sections of this chapter introduce contact formation(CF) trees and the strategies used to manipulate the object while maintaining *form* or *force closure.*

## 4.2 Representation of Contact Formations

According to Desai[4], there are three topological elements of an object or hand polygon. These elements are surfaces, edges, and vertices. Since this thesis only deals with objects in a plane, only two of the above elements are used, edges and vertices(Fig. 4.1). When two or more objects are in contact, there may be several elemental contacts. The contact formation is defined as the set of elemental contacts. This set is represented in the contact formation structure.

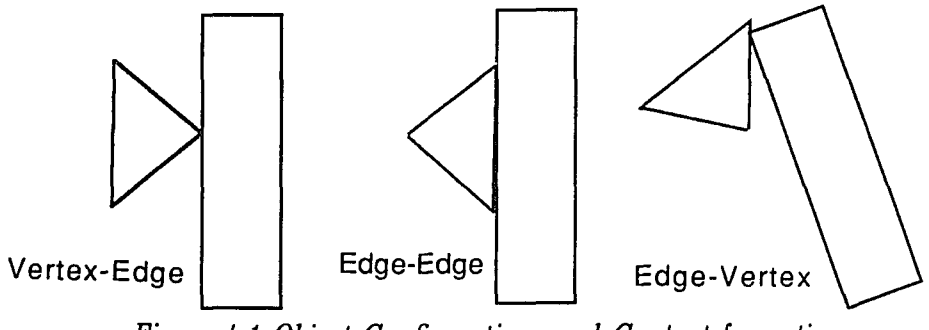



*Figure 4.1 Object Configurations and Contact formations*

The contact formation structure has one element for each hand polygon. Each element has four variables to be recorded. The four variables of a contact formation are:

1) the number of contacts;
2) the type of each contact;
3) the index of the object vertex or edge that is part of the contact, and;
4) the index of the hand polygon that is part of the contact.

There are three types of elemental contacts that are used in a contact formation. The first is a case-one contact, this is when an object vertex contacts a hand polygon's edge. The elemental contact is a "*ve*" or object-vertex-to-hand-edge contact. The second type of contact is an edge contact. The edge contact occurs when there are two case-one contacts on a single hand polygon. This means, simply, that an edge of an object lies on the edge of a hand polygon, this is an "*ee*" or object-edge-to-hand-edge contact. The third and final type of contact considered here, is a case-two contact. It occurs when a vertex of a hand polygon contacts an edge of the object. The elemental contact is a "*ev*" or object-edge-to-hand-vertex contact.

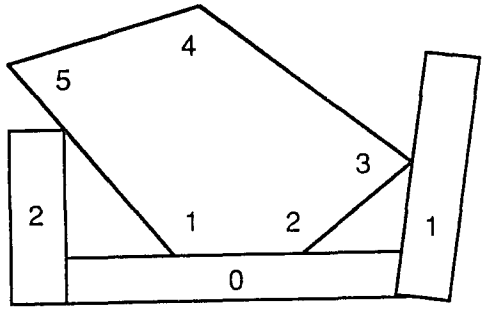
The third element of a *contact formation* is either the object vertex or edge depending on which type of contact has occurred. If the contact is a "*ve*"-type contact, then the object vertex number is $i$ of the vertex ${}^{o}V_i$. If the contact is type-"*ee*" or type-"*ev*", then the number is $j$ for the object edge defined by the two vertices ${}^{o}V_j$ and ${}^{o}V_{j+1}$.

The fourth element of a contact formation is the number of the hand polygon contacted. The numbers of the hand polygons are defined as:

$$\begin{aligned} palm &= 0; \\ finger1, link1 &= 1; \\ finger1, link2 &= 2; \\ \vdots \quad &= \vdots \\ finger1, linkn &= n; \end{aligned}$$

$$finger2, link1 = n + 1;$$
$$finger2, link2 = n + 2;$$
$$\vdots \quad = \vdots$$
$$finger2, linkm = n + m;$$

So an example of a contact formation structure is:




| | | |
|---|---|---|
| "ee" | 1 | 0 |
| "ve" | 3 | 1 |
| "ev" | 5 | 2 |

*Figure 4.2 Contact Formation Structure*

### 4.3 Contact Formation Set

The *configuration-space* approach to manipulation planning was first introduced by Lozano-Pèrez and Wesley[10]. Their approach dealt with finding a collision free path for an object grasped securely in a hand through a field of obstacles. The obstacles were transformed into *configuration obstacles* which represented all the configurations of the object while in contact with the obstacles.

Using *configuration-space obstacles*, Brost[1] found a way to represent a solid, three dimensional, multi-faceted *region* in a three-space whose axes are $x_{obj}$, $y_{obj}$, and $\theta_{obj}$. In the region the object maintained the same elemental contact with the obstacle. As in contact formations, these regions do not contain unique configurations. A region could have an infinite number of different configurations while still maintaining the same elemental contact. Each region has *boundaries* or facets. These define the extremes of the object motion while still maintaining

the same elemental contact. Plotting a path through Brost's three dimensional configuration-space corresponds to the motion an object will go through, as well as its relation to the obstacle, and the contacts between them.

These ideas lead to the *contact formation space* of Desai[4]. The contact formation space is really the configuration space partitioned into contact formations. The difference between Desai's work and the work here, is that instead of using the object's configuration vector, $x_{obj}$, $y_{obj}$, and $\theta_{obj}$, to define the axes of the space, we use the joint angles. *Joint space* is defined by $\theta_1$ and $\theta_2$. An additional angle, the angle of the palm, $\theta_p$, could also be used, but is not a variable in this simulation.

Assume an object is being moved by the fingers and that the object is in some contact formation(CF) X. By moving the fingers, one of two things will happen; the objects will move and maintain the same elemental contacts, or the object will move and a new set of elemental contacts will arise. Desai called these transition motion and reconfiguration motion respectively. The second case occurs when either a previous elemental contact is broken, or a new elemental contact occurs; both result in a new contact formation.

Each point in joint space represents a configuration of the hand. The object is recorded in joint space by its contact formation. For example, some set of joint angles, $\theta_1$ and $\theta_2$, give a contact formation X. Every time the fingers are moved, the contact formation is recorded at that new point in joint space.

Eventually, the finger motion will give rise to a new contact formation. Assuming a starting point, every point tried thereafter must be attained by continuously moving the fingers until that new point is reached, *i.e.*, there can be no discontinuities in finger movement.

Once in a contact formation, we need to know what finger motion will cause a new contact formation. To determine where new CF's occur, the fingers are moved at a number of set rates until a new CF is found, or an instability occurs. Essentially this maps out the boundary that defines the area(not including the boundary) of the original CF. The boundaries of this area are other CF's, or infeasible regions. Infeasible regions are encountered when a particular set of joint

angles cause the object to become unstable. This process will be discussed in more detail in Section 4.4.

Once a CF area has been mapped, the boundary is crossed into a new CF. The area of the new CF is mapped out in the same fashion as explained above. The process is continued until the desired CF is found on a boundary. Then one need only plot a path from the initial CF to the final CF. The CF's in between are the intermediate CF's necessary to reconfigure the object. The path created gives the joint trajectories to be fed to the finger controllers.

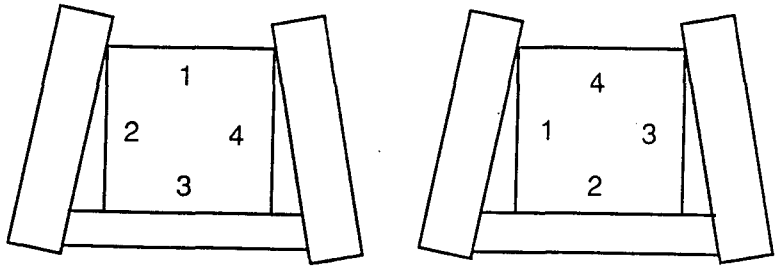



*Figure 4.3 Two CF's with Same Joint Angles*

Certain objects will allow the fingers to be in the same configuration in joint space for two or more contact formations(Fig. 4.3). So some overlapping of the areas might occur. We assume that this can only happen from an intermediate CF(Fig 4.4).

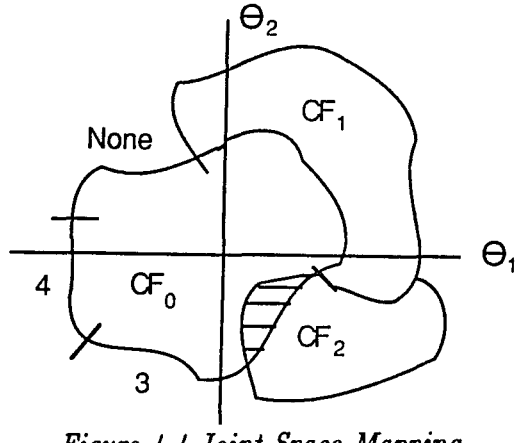



*Figure 4.4 Joint Space Mapping*

## 4.4 Contact Formation Trees

Rather than mapping out the entire contact formation space, a faster approach is to create a tree. Each time a new contact formation is encountered, a node is added to the tree. This is the same as crossing the boundaries in the contact formation space.

Assume that the object is being manipulated by the hand. The configuration of the object and hand will be recorded when its quasi-static motion properties change qualitatively. This, for example, occurs when an existing contact has been broken, or when a new contact is encountered.

A new contact formation is not the only time that a new node is added to the tree. If the current configuration is in form closure and it moves out of this grasp condition, then a new node is added. A new node will not be added if it has the same configuration and grasp condition as the parent node. Analogously, if the object is in force closure and it is moved to a form closure grasp, then a new node will be created.

When in form closure, recall that form closure may have the edge contact on either the fingers or the palm, the fingers will move to maintain this grasp condition. This means that one finger will push, and the other will have compliant motion; so $\theta_1$ can be viewed a function of $\theta_2$ or *vice versa*. Form closure will be maintained as long as the intersection between the two case-one normal forces are in between the two normal forces that define the edge contact. When solving for the object's position, three contacts will be maintained. If the edge contact is on the palm, then the three contacts to be maintained are the two contacts on the palm, and the contact on the pushing finger. If the edge contact is on a finger, the two contacts that define the edge contact and the contact on the palm will be maintained. As long as the object remains in form closure, the same contacts will be maintained. The object is pushed until either a new contact formation is found (in which case a new node is created) or until the object leaves form closure (which also results in a new node). While doing form closure manipulations, the object will not become unstable.

When in force closure it is much more difficult to stably manipulate the object. Force closure manipulations depend on contiguous contact formations areas. We must be able to map a straight line from one point on a boundary to another. This line represents a set of constant angular velocities of the joints.

Once in a new CF, we must choose the angular velocities $\dot{\theta}_1$ and $\dot{\theta}_2$ for the joints. This is done by choosing ratios of $\dot{\theta}_1/\dot{\theta}_2$. These ratios are obtained by traveling around a unit circle. The number of ratios(spokes) need to be determined. Then by evenly spacing the unit length spokes, a set of $x$ and $y$ coordinates can be found at the end of each spoke. Each pair gives the ratios $x/y$ which can be substituted for $\dot{\theta}_1/\dot{\theta}_2$(Fig 4.5). Once the ratio is found, it needs to be determined if the velocities:

1) are kinematically feasible, and
2) maintain stability.

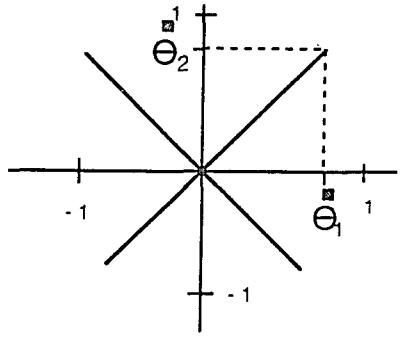



*Figure 4.5 Spokes*

The procedure for determining these two criteria is to solve a linear program. The linear program is the *force formulation* for the frictionless case of the object motion problem (see Trinkle and Paul[16]). Use of the force formulation is two fold in purpose. First, it tells if the angular velocities are feasible and if the object maintains stability; second, if there are more than three contacts and the object is not in form closure, it tells which of the contacts will break when the motion is

performed. The form of the LP is as follows:

$$\begin{aligned} &\text{Max } \textstyle\sum_{i=1}^{n_c} \dot{\theta}_i^T \mathbf{j}_{in} c_{in} \\ &\text{s.t. } \mathbf{g}_{ext} + \textstyle\sum_{i=1}^{n_c} \mathbf{W}_{in}\ c_{in} = 0 \\ &\qquad\qquad\qquad\qquad c_{in} \geq 0; \quad i \in I_{n_c} \end{aligned}$$

where $j_{in}$ is the normal component of the Jacobian of the $i^{th}$ contact so that $\dot{\theta}_i^T \mathbf{j}_{in}$ is the velocity transferred to the object through contact $i$. There are as many variables in the LP as there are contacts, but there will always be exactly three equations. The variables $c_{in}$ are the magnitude of the forces that are applied, and the matrix $W_{in}$ is the wrench matrix described in Chapter 2. If the LP is infeasible, it means that the grasp has become unstable, and the angular velocities leading to the instability are discarded. If the result is unbounded, then the object is in form closure and the proposed $\dot{\theta}_1$ and $\dot{\theta}_2$ are kinematically inadmissible. If a basic feasible solution exists, then stability is maintained and the choice of joint velocities is kinematically admissible. Any variable that is returned as non-basic from the LP indicates that the magnitude of the force applied at the contact is zero and has therefore been broken. Any non-zero basic variable corresponds to a contact that will be maintained. These will be sent to the object's position solver, in Chapter 3.5, as the contacts to be maintained.

The fingers continue to move at the angular velocities until a new contact formation is found, the grasp becomes unstable, or the object enters into form closure. A new node is created when the object enters into form closure or a new contact is found. Once one of these conditions has occurred, the object is returned to its original position, and the next pair of angular velocities is used. The method for finding angular velocities can be described as traversing the circumference of a unit circle and at $n$ equidistant stops taking the x-component as the value for $\dot{\theta}_1$ and the y-component as the value for $\dot{\theta}_2$(Fig. 4.6). So $n$ spokes of unit length are found and each end point gives a different pair of angular velocities. This essentially maps out the boundary for the current contact formation. The values for $\dot{\theta}_1$ and $\dot{\theta}_2$ need not be recorded in joint space.

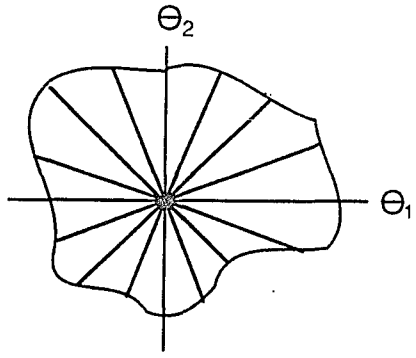



*Figure 4.6 Mapping Joint Space*

### 4.5 Using Two Contact Formation Trees

Two trees were used to try and speed up planning time. The first tree, or the initial tree, has as its root, the initial configuration. The second tree, or goal tree, has as its root, the final configuration. Since the operations require that the frictionless object always be stable, the manipulations are reversible and the goal tree can be grown backward toward the initial grasp contact formation. Both trees are linked lists where the front pointers go forward as in a normal linked list, but the back pointers point to the parent node(Fig. 4.7), this is called a tri-structure tree. The algorithm for creating a tri-structured tree is as follows:

```
Create root node;
parent = root;
vertex = root;
while(expand tree)
  while(child node can be created)
    create child node
    child node back pointer = parent;
    vertex next pointer = child;
    vertex = vertex next pointer;
  end
  parent = parent next pointer;
end
```

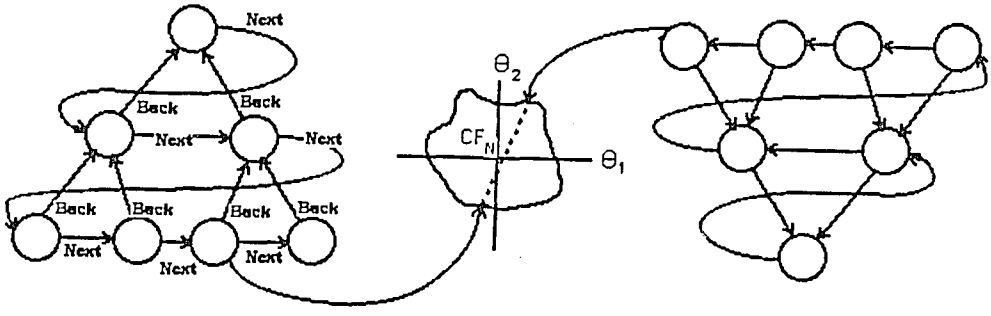



*Figure 4.7 CF Trees*

In each node of the tree, we maintain all the information necessary to reconstruct the configuration of the hand and the position and orientation of the object. The contact formation is placed in the node. This is used to determine if the goal configuration has been reached. The angles of the fingers are saved, as are $x_{obj}$, $y_{obj}$, and $\theta_{obj}$. The angular velocities of the fingers $\dot{\theta}_1$ and $\dot{\theta}_2$ that were used to get to the node are also saved in the node.

The trees each grow one row at a time. After a row is created, the nodes are checked to see if the same contact formation exists in both trees. When the same node exists in both trees, the trees are connected at that contact formation. The CF's that are the same in each tree are assumed to exist in the same CF area in joint space. Each tree's CF enters this area from a different point on the boundary. If we assume that the area is defined as a simple closed curve and its interior, then getting from one spot on the boundary to another is best done by using the line between them as the path. Once the line is known, a continuous path from the initial configuration to the final configuration has been constructed(Fig. 4.7).

# CHAPTER 5 - RESULTS AND CONCLUSIONS

## 5.1 Results

In trying not to sacrifice generality, the program became quite large. The object that the user chooses to be manipulated is placed in a data file called "*obj1.dat*". The first line of this data file have the number of object vertices. The second and following lines has a coordinate pair each describing an object vertex. The pairs are arranged such that the perimeter of the object is traversed counter-clockwise. The same is done for each element of the hand. Each finger link is described by "*link.dat*", and the palm is described by "*palm.dat*".

The program then prompts the user to specify which configuration to start with, and which configuration to end with. Using *MicroSoft QuickC Graphics*, a number of graphic routines were customized to display the manipulator's initial and final configurations, as well as show the search for an optimal path.

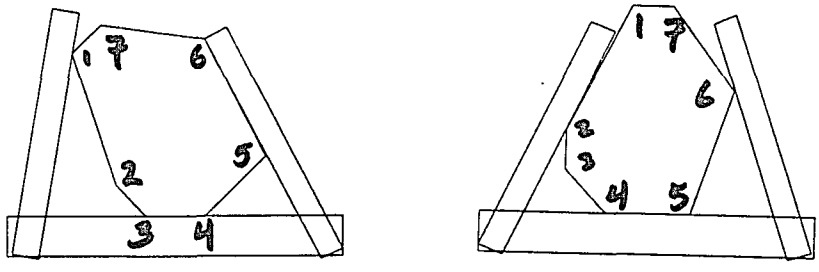



*Figure 5.1 Initial and Final Configuration for Seven Sided Object*

Once in place, the initial configuration was "exercised" until all the children were found. Then the final configuration was "exercised" in the same fashion. Once an entire row had been built on both trees, the trees were checked to determine if there existed a node in each tree with the same contact formation. If there was

such a pair of nodes, then the algorithm had converged. If there was not such a pair of nodes, then the next row for each tree was constructed. This process continued until the same contact formation appeared in both trees.

Figure 5.1 shows a seven-sided object being manipulated from side two to side three. It took seventy units of time to complete the operation. For each unit of time the force exerted on each vertex of the object and the hand were calculated. The non-zero graphs are shown. These forces were calculated using the wrench matrix for the grasp. The equations for this calculation are:

1) For 3 contacts,

$$\mathbf{c} = -\mathbf{W}^{-1}\mathbf{g_{ext}}$$

2) For 4 contacts,

$$\mathbf{c} = -\mathbf{W}^{\dagger}\mathbf{g}_{ext} + \mathbf{N}k$$

where $\mathbf{W}^{\dagger}$ is the pseudo-inverse, $\mathbf{N}$ is a normalized column vector in the null space of $\mathbf{W}$, and $k$ is the smallest positive scalar chosen such that $\mathbf{c} \geq \mathbf{0}$.

3) For 5 contacts,

$$\mathbf{c} = -\mathbf{W}^{\dagger}\mathbf{g}_{ext} + \mathbf{N}\begin{pmatrix} k_1 \\ k_2 \end{pmatrix}$$

where $\mathbf{W}^{\dagger}$ is the pseudo-inverse, $\mathbf{N}$ is a matrix whose columns are basis vectors in the null space of $\mathbf{W}$, and $k_1$ and $k_2$ are the smallest positive constants chosen such that $\mathbf{c} \geq \mathbf{0}$. A linear program is used to determine the smallest pair of $k_1$ and $k_2$. The LP is as follows:

$$\begin{aligned} &\textbf{Max } k_1 + k_2 \\ &\textbf{s.t. } -\mathbf{W}^{\dagger}\mathbf{g}_{ext} + \mathbf{N}\mathbf{k} \geq 0 \\ &\qquad\qquad k_n \geq 0; \quad n = 1, 2 \end{aligned}$$

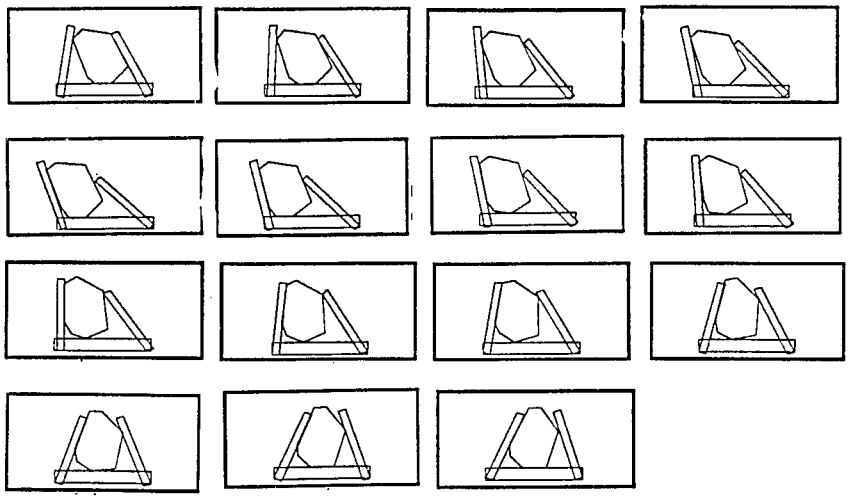


*Figure 5.2 Frames of Motion for Seven-Sided Object*

In the graphs that follow, there appear to be discrepancies between the frames of motion and the graphs of force histories. In fact there are no discrepancies. One of these apparent discrepancies shows an object vertex one in frame one contacting finger two, but the graph of vertex one's force history shows that the force is zero until time five. This is a result of using the linear program described earlier in this chapter. Using the linear program in the five case contact will occasionally cause either $k_1$ or $k_2$ to be zero. When $k_1$ or $k_2$ are zero, it is the same as saying that a contact has occurred, but the force applied is infinitely small, and therefore zero. So discontinuities in the force history graphs are a result of the LP.

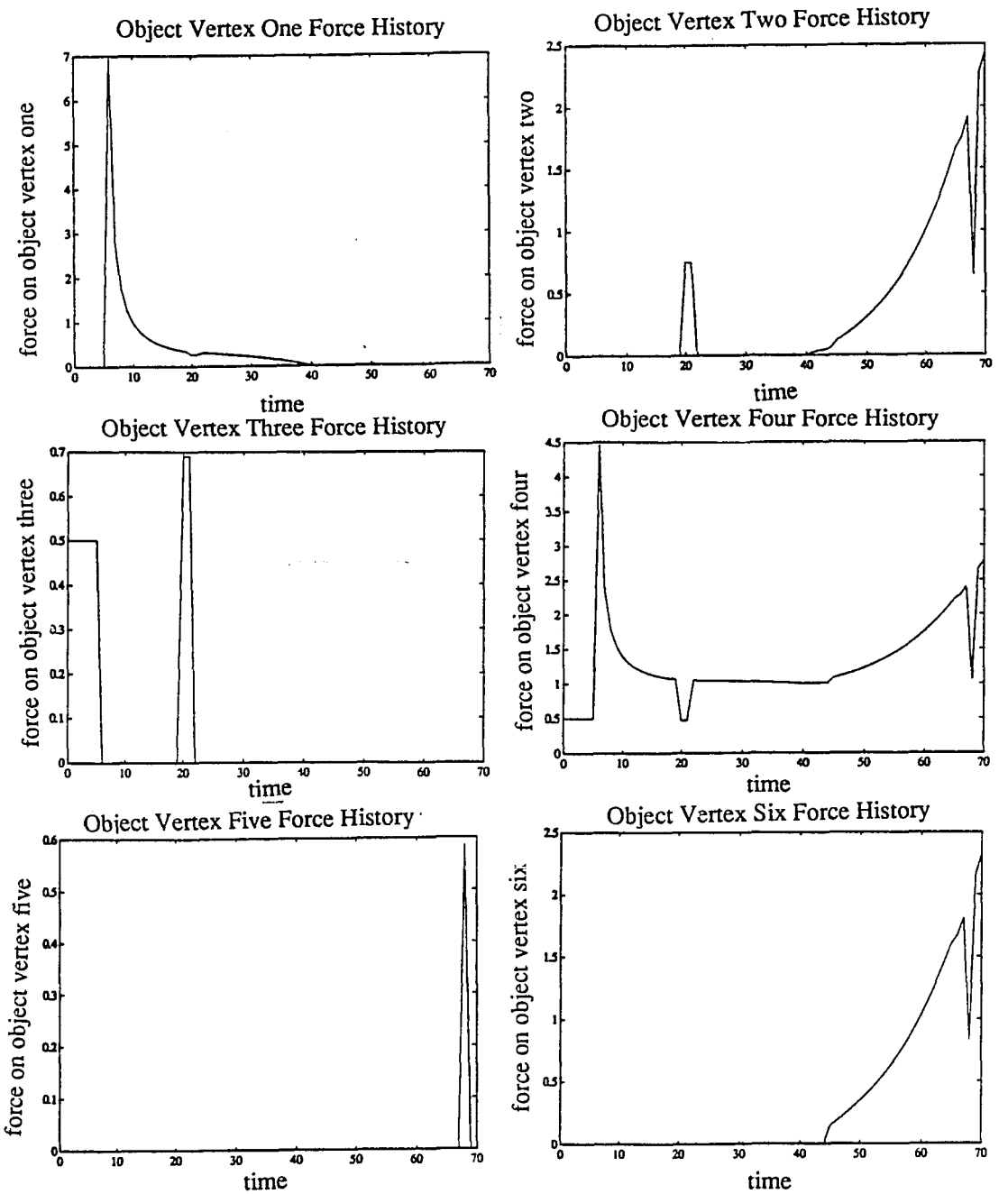



*Figure 5.3 Graphs of Object Force Histories*

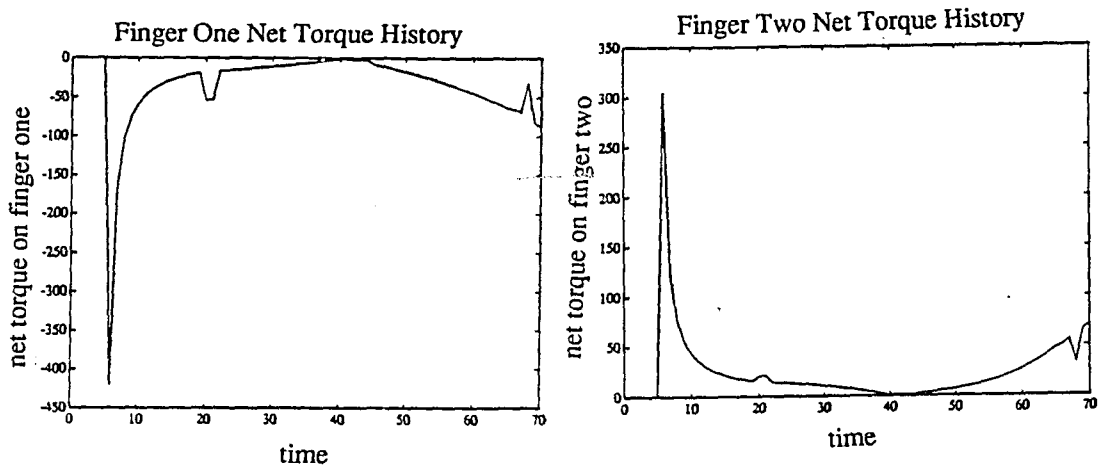



*Figure 5.4 Net Torque*

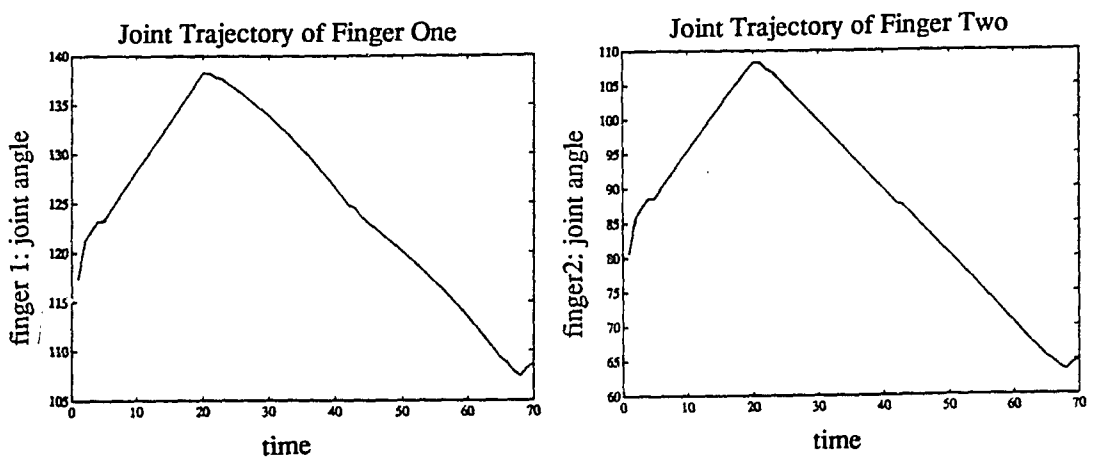



*Figure 5.5 Joint Trajectories*

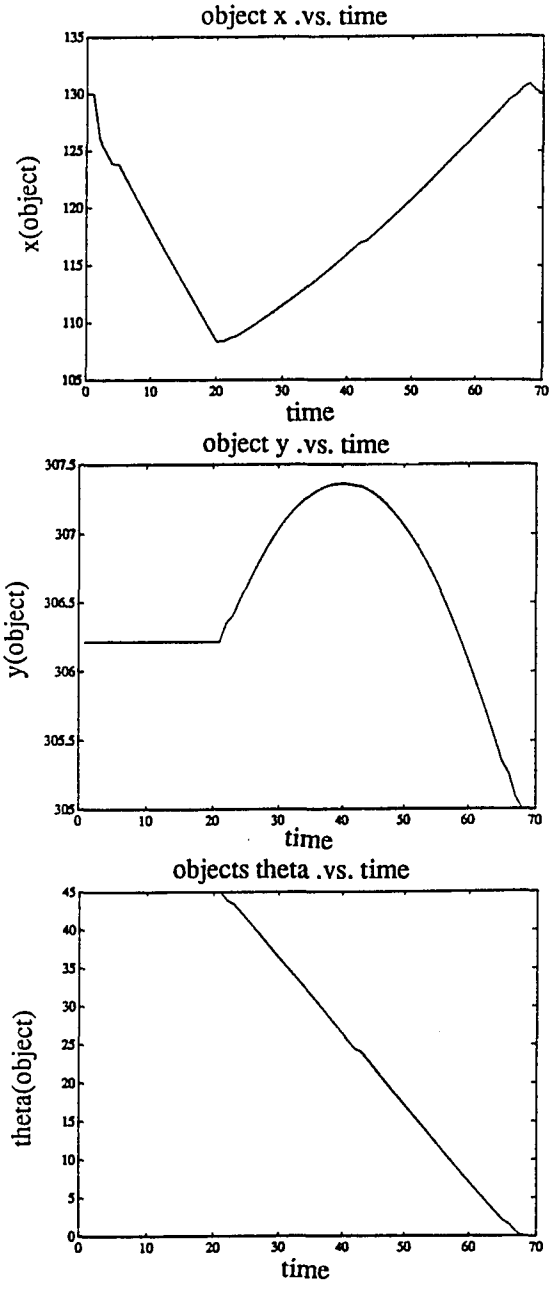



*Figure 5.6 Object Motion Over Time*

Clearly this model implies that in-hand reconfiguration is feasible, of this there is no doubt. A question of equal importance: when does the model fail?

## 5.2 Problems

A solution for reconfiguring an object did not always exist. The geometry of some objects will not allow stability at all times. When we attempted to reconfigure these objects, the program eventually got into a cycle and the trees would grow until the PC ran out of memory. There were also objects that would cause singularities in the numerical routines. More specifically, these objects caused matrix manipulations to become ill-conditioned. With much more work, finger and object placement by numerical solutions could be replaced by geometric solutions. We attempted placing fingers on the object using geometric solutions. Originally finger placement was accomplished using an iterative routine; changing to a geometric solution increased the speed of the program and reduced the number of objects that caused ill-conditioning, thereby allowing manipulation planning for a greater variety of objects.

Another problem was the $\epsilon$ chosen to be the contact distance. If the distance was too large, many point-to-point contacts occurred eventually causing the program to fail. Conversely, if the distance was to small, contacts that existed were not found. This problem was caused by the error generated in using an iterative method to move the object (Chapter 3.5). The solved position of the object was not accurate enough to use a value of $\epsilon$ less than $10^{-4}$.

## 5.3 Future Work

There are several ways to improve the program in future work. Adding friction is an obvious first step, this would complete the two-dimensional model of the hand. There is also the prospect of allowing instabilities, or allowing the object to fall when it becomes unstable. So rather than softly moving the object to its next edge contact, perhaps the object would fall to its next edge contact. The use of a goal tree would be precluded since the path could not be generated backwards. The time to find a path would be greatly decreased, possibly obtaining a solution fast enough to be of use in real time, but the sacrifice may be accuracy. Yet another possibility would be an attempt at using Brost's *configuration space obstacles* in

conjunction with the *joint space* approach used here for a more complete treatment on contact configurations.

# APPENDIX A - CONTACT TYPES

Shown here are seven types of contact that may occur. Each contact type has a figure picturing the contact, a *contact constraint matrix*, and a *contact freedom matrix*. The *contact constraint matrix* indicates the directions of motion in which the contact is constrainted. The *contact freedom matrix* indicates the directions of motion in which the contact is not constrained.

| Contact | Contact Constraint Matrix | Contact Freedom Matrix |
|---|---|---|
| No Contact | $[0]$ | $[I]_{6\times6}$ |
| Point Contact<br>No Friction | $\begin{pmatrix}0\\0\\1\\0\\0\\0\end{pmatrix}$ | $\begin{pmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{pmatrix}$ |
| Edge Contact<br>No Friction | $\begin{pmatrix}0&0\\0&0\\1&0\\0&1\\0&0\\0&0\end{pmatrix}$ | $\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$ |
| Surface Contact<br>No Friction | $\begin{pmatrix}0&0&0\\0&0&0\\1&0&0\\0&1&0\\0&0&1\\0&0&0\end{pmatrix}$ | $\begin{pmatrix}1&0&0\\0&1&0\\0&0&0\\0&0&0\\0&0&0\\0&0&1\end{pmatrix}$ |

| Contact | Contact Constraint Matrix | Contact Freedom Matrix |
|---|---|---|
| Point Contact Friction | $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ | $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ |
| Edge Contact Friction | $\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$ |
| Surface Contact Friction | $[\mathbf{I}]_{6\times 6}$ | $[\mathbf{0}]$ |

# APPENDIX B - FINGER PLACEMENT

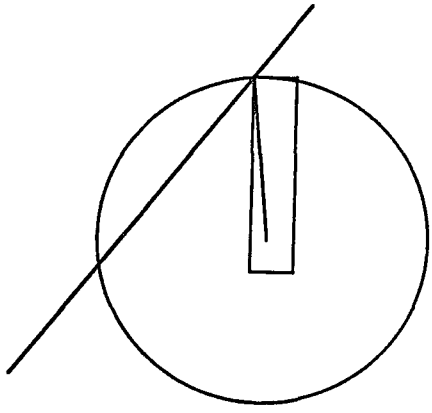


*Figure B.1 Placing finger*

If $|m| < 0$ where $m$ is the slope of the line.

$$y = mx + b \tag{B.1}$$

$$r^2 = x^2 + y^2 \tag{B.2}$$

Substituting (B.1) into (B.2) gives the following quadratic equation.

$$(m^2 + 1)x^2 + 2mbx + (b^2 - r^2) = 0 \tag{B.3}$$

solving for $x$ gives:

$$x = \frac{-mb \pm \sqrt{m^2r^2 - b^2 - r^2}}{(m^2 + 1)} \tag{B.4}$$
$$y = \pm\sqrt{r^2 - x^2}$$

if $|m| \geq 0$ then

$$x = \frac{1}{m}y - \frac{b}{m} \Rightarrow x = cy + d \tag{B.5}$$

$$r^2 = x^2 + y^2 \tag{B.2}$$

similar computation gives:

$$y = \frac{-cd \pm \sqrt{c^2r^2 - d^2 - r^2}}{(c^2+1)}$$
$$x = \pm\sqrt{r^2 - y^2} \qquad (B.6)$$

In both cases, there are two solutions for both $x$ and $y$, but only one solution lies on the edge of the object. A simple test determines which are the correct values for ($x$,$y$). Once the point is known, the angle $\phi$ can be solved for and the interior finger angle $\alpha$ is added to get the joint angle $\theta$.

## APPENDIX C - PROGRAM

The program and all it routines are supplied in this appendix. Each routine has a commented header that describes what the functions does.

```
#include "minclude.h"
#define SPOKES 12
#define LDIFF 0.005
#define EPSILON 0.05

main()
{
   char
            *palm_name,
            ***link_names,
            *object_name,
            *w;

   point
            palm_base,
            d_trans,
            base1,
            base2,
            oldobase,
            obase;

   float    temp;

   double   sz,
            size,
            obj_info[4],
            diff1,
            diff2,
            gext,
            otheta,
            oldotheta,
            thetaf[2][2],
            oldthetaf[2][2],
            thetafdot[2][2],
            thetap,
            theta,
            b_i,
            d_i,
```

```
        tol,
        tdiff,
        solve_finger_angle(),
        **dmatrix(),
        **a1,
        **m_create();

int
        i,
        ind,
        j,
        k,
        edge,
        k1,
        n,
        n4,
        n5,
        ns,np,ms,mp,m1,m2,m3,nm1m2,go,
        c[3],
        stype,
        tree,
        cont,
        done,
        good,
        same,
        lastnode,
        legal,
        samecf,
        samecfnum,
        edge_search,
        x_pixels,
        y_pixels,
        edge_num,
        *links,
        **fverteces,
        overteces,
        pverteces,
        num_hand_polygons,
        num_fingers,
        num_finger_links[2],
        N,
        LPP,
        oldN,
        L[3],
```

```
            placef1,
            placef2,
            fc[3][4],
            nec,
            nfc,
            nfc1,
            row1[2],
            heapsize[2],
            match,
            match1[2],
            esc,
            icase,
            *basic,
            *svar,
            *non_basic,
            *ivector(),
            inc1,inc2,dec1,dec2,
            test_for_samecf(),
            same_cf(),
            equivelant();

p_ptr
            tobject,
            object,
            initialize_polygon(),
            object_list,
            hand_list[7];

h_ptr
            h1,
            h2,
            initialize_hand();

b_ptr       b0;

void
            draw_finger(),
            draw_polygon(),
            free_polygon(),
            solve_position(),
            getL(),
            begin_ps(),
            end_ps(),
            call_ps(),
```

```
            move_finger();

char
            text[80],
            filename[10],
            answer,
            graph,
            *strsave();

FILE
            *fileptr,
            *file1,
            *file2,
            *file3;

rt_ptr
            **A,
            **B,
            **D,
            **T_frame,
            finger_placement[2],
            **initialize_rt_array2(),
            ***initialize_rt_array3();

ci_ptr
            ci,
            oldci,
            initialize_ci();

ni
            fn,
            oldfn,
            initialize_fn(),
            force_normals();

cf          oldcf[3],cf[3];

cfnode      n1[2],v1[2],current1[2],
          root1[2],path,p1[2],addnode();
```

```
/************** INITIALIZATION ******************/

    num_hand_polygons = 3;
    num_fingers = 2;
    num_finger_links[0] = 1;
    num_finger_links[1] = 1;
    printf("Turn graphics on?(y/n)");
    graph = getch();
    printf("%c\n",graph);
    printf("Input Epsilon");
    scanf("%f",&temp);
    tol = (double) temp;


    link_names = (char ***)calloc(num_fingers, sizeof(char **));
    for (i = 0; i < num_fingers; i++)
    link_names[i] = (char **)
                calloc(num_finger_links[i], sizeof(char *));

    for (i=0;i<2;i++)
    finger_placement[i] = (rt_ptr)
            malloc(sizeof(struct relative_transformations));
    base1.x = 35;
    base1.y = 0;
    coord_transform(0.0,base1,finger_placement[0]->rt);

    base2.x = -35;
    base2.y = 0;
    coord_transform(0.0,base2,finger_placement[1]->rt);

    links = (int *)calloc(num_fingers, sizeof(int));

    fverteces = (int **)calloc(num_fingers, sizeof(int*));
    for (i = 0; i < num_fingers; i++)
      fverteces[i] = (int *)calloc(num_finger_links[i],sizeof(int));

    b0 = (b_ptr)malloc(sizeof(struct b0_vert));
    fn = (ni)malloc(sizeof(struct normal_info));
    oldfn = (ni)malloc(sizeof(struct normal_info));

    w = strsave("test.try");
    link_names[0][0] = strsave("link.dat");
    link_names[1][0] = strsave("link.dat");
    object_name = strsave("obj1.dat");
```

```
palm_name = strsave("palm1.dat");

get_vertex_count(&pverteces, fverteces, &overteces, palm_name, link_names,
                 object_name, num_fingers, num_finger_links);

palm_base.x = 200;
palm_base.y = 200;
thetap = 0;
thetaf[0][0] = 15;
thetaf[1][0] = 135;
thetaf[0][1] = 0;
thetaf[1][1] = 0;
thetafdot[0][0] = 0;
thetafdot[1][0] = 0;
thetafdot[0][1] = 0;
thetafdot[1][1] = 0;

for (i = 0; i < num_fingers; i++)
  links[i] = num_finger_links[i];

h1 = initialize_hand(h1, num_fingers, links, fverteces, pverteces, finger_placement,
          palm_base);

object = initialize_polygon(object, overteces, 0);

get_polygon_coordinates(pverteces, fverteces, overteces, palm_name, link_names,
           object_name, num_fingers, num_finger_links, h1, object);
h1->palm->name = 0;
h1->finger[0]->link[0]->name = 1;
h1->finger[1]->link[0]->name = 2;
object->name = 3;
h1->finger[0]->theta_i[0] = 90;
h1->finger[0]->b_i[0] = 35;
h1->finger[1]->theta_i[0] = 105;
h1->finger[1]->b_i[0] = -35;

d_trans.x = 25;
d_trans.y = 0;
coord_transform(0.0,d_trans,h1->finger[0]->D[0]->rt);
d_trans.x = 25;
d_trans.y = 0;
coord_transform(0.0,d_trans,h1->finger[1]->D[0]->rt);
```

```
initialize_screen(&x_pixels, &y_pixels);
sz = ((double) y_pixels / (double) x_pixels);
A = initialize_rt_array2(A, num_fingers, num_finger_links[0]);
B = initialize_rt_array2(B, num_fingers, num_finger_links[0]);
D = initialize_rt_array2(D, num_fingers, num_finger_links[0]);
T_frame = initialize_rt_array2(T_frame, num_fingers, num_finger_links[0]);

ci = (ci_ptr) malloc(sizeof(struct contact_information));
ci->v = (struct contact_verticies *)
        calloc(object->vert_num*2,sizeof(struct contact_verticies));
ci->e = (struct contact_verticies *)
        calloc(object->vert_num*2,sizeof(struct contact_verticies));
ci = initialize_ci(object, ci);

h2 = initialize_hand(h2, num_fingers, links, fverteces, pverteces, finger_placement,
        palm_base);
tobject = initialize_polygon(tobject, overteces, 0);

printf("%s %d %s\n","There are",object->vert_num,
                "edges, which will start on the palm? ");
scanf("%d",&j);

printf("%s %d %s\n","There are",object->vert_num,
                "edges, which will end on the palm? ");
scanf("%d",&k1);

begin_ps (filename, fileptr);

put_text_for_windows();

size = 1.0;
for(k=0; k<4; k++)
{
 if(k==0)
   {palm_base.x=500; palm_base.y=450; edge_num=j;}

 else if(k==1)
   {palm_base.x=500; palm_base.y=200; edge_num=k1;}

 else if(k==2)
   {palm_base.x=130; palm_base.y=280; edge_num=k1;}

 else if(k==3)
   {palm_base.x=130; palm_base.y=280; edge_num=j;}
```

```
place_object(object, obj_info, palm_base, edge_num);
obase.x = obj_info[0];
obase.y = obj_info[1];
otheta  = obj_info[2];

move_object(object, tobject, obase.x, obase.y, otheta);

thetaf[0][0] =solve_finger_angle(h1,palm_base,tobject,1);
thetaf[1][0] =solve_finger_angle(h1,palm_base,tobject,2);

move_hand(h1, h2, thetaf, thetap, palm_base,
                        A,B,D, T_frame);

    if(k==2 || k==3)
     {
       get_hand_list(h2,num_fingers,num_finger_links,hand_list);
       object_list = tobject;

       solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
                    A,B,D,T_frame, hand_list, 1, &done);

       solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
                    A,B,D,T_frame, hand_list, 2, &done);

       contact_normals(ci,object,hand_list,object_list,h1,tol,
                        &legal,L,fn, obase);

       get_cf(ci, cf);
       form_closure(thetaf, fn, fc, &nfc);
       size = 1.5;
       stype = 0;


      if(k==3)
       {ind = 0;
        row1[ind] = 1;   heapsize[0] = 1;  n1[ind] = NULL;
    tree = 1;
       call_ps (hand_list, object_list, filename, fileptr);
       current1[ind] = addnode(cf, obase, otheta, thetaf, n1[ind],
                    row1[ind],heapsize[0]++, thetafdot, c,tree, stype);
        v1[ind] = current1[ind];
          root1[ind] = current1[ind];
```

```
            }
           if(k==2)
            {ind = 1;
             row1[ind] = 1;  heapsize[1] = 1;  n1[ind] = NULL;
             tree = 2;
             call_ps (hand_list, object_list, filename, fileptr);
             current1[ind] = addnode(cf, obase, otheta, thetaf, n1[ind], row1[ind],
                          heapsize[1]++, thetafdot, c, tree, stype);
            v1[ind] = current1[ind];
            root1[ind] = current1[ind];
            }


         }
     if(k<2)
      {
        draw_hand(h2, 3, y_pixels, sz, size);
        draw_polygon(tobject, 2, y_pixels, sz, size);
      }
   }
   make_windows();
   sprintf(text,"Press any key to continue...\n");
   _outtext(text);
   getch();

   size = 1.5;
   match = 0;
   ind = 0;
   file1 = fopen("tree.dat","w");


/***************** BEGINNING OF THE LOOP ********************/

  while(!match)
   {

      current1[0] = root1[0];
      while(current1[0]->row != row1[0])
        current1[0] = current1[0]->next;
      esc = 0;
      while(current1[0]->row == row1[0] && !esc)
       {
        n1[1] = root1[1];
        do{
```

```
        current1[1] = n1[1];
        match = nodes_equal(current1[0],current1[1]);
        if(n1[1]->next != NULL)
          n1[1] = n1[1]->next;
       }while(current1[1]->next != NULL && !match);
      if(current1[0]->next != NULL && !match)
        current1[0] = current1[0]->next;
      else
        esc = 1;
     }


if(!match)
 {

 for(ind=0;ind<2;ind++)
  {
  tree = ind+1;
  while(v1[ind]->row == row1[ind])
   {

    thetaf[0][0] = v1[ind]->thetaf[0][0];
    thetaf[1][0] = v1[ind]->thetaf[1][0];
    obase = v1[ind]->obase;
    otheta = v1[ind]->otheta;

    move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
          palm_base,A,B,D,T_frame, num_fingers,num_finger_links, hand_list,
          object_list, tobject, ci, tol, &legal, L, fn);

    form_closure(thetaf, fn, fc, &nfc1);
    nec = number_of_edge_contacts(v1[ind]->cf);


  if(nfc1)
     for(k1=0;k1<nfc1;k1++)
      for(n=0;n<2;n++)
       {

        thetaf[0][0] = v1[ind]->thetaf[0][0];
        thetaf[1][0] = v1[ind]->thetaf[1][0];
        obase = v1[ind]->obase;
        otheta = v1[ind]->otheta;
```

```
move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
   palm_base,A,B,D,T_frame,  num_fingers,  num_finger_links,hand_list,
     object_list, tobject, ci, tol, &legal, L, fn);

get_cf(ci, cf);

get_formc_parameters(&placef1,&placef2, c, fc[k1], n, cf, &inc1,&in
              c2,&dec1,&dec2,fn);
if(graph==’y’)
draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);

samecf = 1; legal = 1; tdiff = 1.5; done = 0;

while(!done)
{
 for(i=0;i<2;i++)
  for(j=0;j<2;j++)
   oldthetaf[i][j] = thetaf[i][j];
 oldotheta = otheta;
 oldobase = obase;
 copy_cf(cf,oldcf);
 copy_fn(fn,oldfn);
 legal = 1;

 if(inc1) thetaf[0][0]+= tdiff;
 if(inc2) thetaf[1][0]+= tdiff;
 if(dec1) thetaf[0][0]-= tdiff;
 if(dec2) thetaf[1][0]-= tdiff;

 move_all(h1, h2, thetaf,thetap, palm_base,A,B,D,T_frame, num_fingers,
        num_finger_links, hand_list, ci, object, b0, c, &obase, &otheta,
        tobject, object_list, &done);

 if(!done)
  {
   if(placef1)
    solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
          A,B,D,T_frame, hand_list, 1, &done);

   if(placef2)
    solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
          A,B,D,T_frame, hand_list, 2, &done);

   if(thetaf[0][0]<=20 || thetaf[1][0] >=160)
```

```
    done = 1;
 }

if(!done)
 {
  contact_normals(ci,object,hand_list,object_list,h1,tol,
                  &legal,L,fn, obase);

  get_cf(ci, cf);
  samecf = test_for_samecf(oldcf,cf);


  if(legal && !samecf)
   { samecfnum = same_cf_numbers(oldcf,cf);
     if(samecfnum)
      legal = 0;
   }
 }
if(done)
 legal = 0;

if(legal)
{
 if((cf[1].type == 0 || cf[0].type == 0)&&
    (equivelant(fn->c[1],fn->c[3],0.5) ||
     equivelant(fn->c[1],fn->c[4],0.5)))
      {legal = 0; done = 1;}

        if((cf[2].type == 0 || cf[0].type == 0)&&
             equivelant(fn->c[2],fn->c[3],0.5))
      {legal = 0; done = 1;}
}

  good = good_cf(cf);

if((!samecf || !in_form_closure(thetaf, fn, fc[k1]))
      && ci->N>=3 && legal && good)
{
     match1[ind]=0;
     p1[ind] = v1[ind];
     if(!match1[ind] && p1[ind]->back!=NULL)
      do{
        match1[ind] = new_nodes_equal(p1[ind]->back,cf);
        if(match1[ind])
```

```
            match1[ind]  =  equivelant(p1[ind]->back->obase,
                                       obase, 1.0);
           if(p1[ind]->back!=NULL)
            p1[ind] = p1[ind]->back;
          }while(p1[ind]!=root1[ind] && !match1[ind]);

        if(!match1[ind] && v1[ind]->node_number>3)
         match1[ind] = new_nodes_equal(root1[ind],cf);

        if(inc1)
          stype = 1;
        else if(dec1)
          stype = -1;
        else if(inc2)
          stype = 2;
        else if(dec2)
          stype = -2;

        if(!match1[ind])
         {
          sprintf(text,"%s %d %s %d\n", "node #",
                  v1[ind]->node_number, "heapsize",
                  heapsize[ind]);
         _outtext(text);

         call_ps (hand_list, object_list, filename, fileptr);
          fprintf(file1,"%d %d %d\n", tree, heapsize[ind],
                        v1[ind]->node_number);

         n1[ind] = addnode(cf, obase, otheta, thetaf, v1[ind], v1[ind]->row+1,
              heapsize[ind]++, thetafdot, c, tree, stype);
         current1[ind]->next = n1[ind];
         current1[ind] = n1[ind];
        }
}
if(legal)
 tdiff = 1.5;
else
{
 tdiff*= 0.5;
 for(i=0;i<2;i++)
  for(j=0;j<2;j++)
   thetaf[i][j] = oldthetaf[i][j];
 otheta = oldotheta;
```

```
        obase = oldobase;
        copy_cf(oldcf,cf);
        copy_fn(oldfn,fn);
        move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
             palm_base,A,B,D,T_frame,num_fingers,num_finger_links,hand_list,
             object_list, tobject, ci, tol, &legal, L, fn);
        if(tdiff<LDIFF)
         legal = 0;
        samecf = 1;
       }
       if(samecf && in_form_closure(thetaf, fn, fc[k1]) && legal && !done)
        done = 0;
       else
        done = 1;

      if(graph=='y')
       draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);

     } /*end while*/
    } /* end nfc */

if(!nfc1 || nec)
{
 n4 = 0;  n5 = 0; k1 = 0; n = 0;
 cont = 1;
 k1 = number_of_edge_contacts(v1[ind]->cf);
 while(cont)
 {
  if(n5<k1)
    edge_search = 1;
  else
    edge_search = 0;

  for(i=0;i<2;i++)
   for(j=0;j<2;j++)
     thetaf[i][j] = v1[ind]->thetaf[i][j];
  obase = v1[ind]->obase;
  otheta = v1[ind]->otheta;

  move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
     palm_base,A,B,D,T_frame, num_fingers, num_finger_links, hand_list,
     object_list, tobject, ci, tol, &legal, L, fn);

  get_cf(ci, cf);
```

```
if(edge_search)
 {
  get_edge_parameters(&placef1,&placef2, c, n5, n++, cf,
                      &inc1,&inc2,&dec1,&dec2);
  if(inc1||inc2)
    {thetafdot[0][0] = 1;
     thetafdot[1][0] = 1;}
  if(dec1||dec2)
    {thetafdot[0][0] = -1;
     thetafdot[1][0] = -1;}

 }
else
 {
  generate_angular_velocities(SPOKES,n4,thetafdot);
  placef1 = 0;  placef2 = 0;
 }

if(graph=='y')
draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);

done=0;  samecf = 1;  legal = 1; tdiff = 1.5;

while(!done)
 {
  for(i=0;i<2;i++)
   for(j=0;j<2;j++)
    oldthetaf[i][j] = thetaf[i][j];
  oldotheta = otheta;
  oldobase = obase;
  copy_cf(cf,oldcf);
  copy_fn(fn,oldfn);

  if(!edge_search)
   {
    ns = ci->N; ms = 3;
    np = ns+1; mp = ms+2;
    m1 = 0; m2 = 0; m3 = 3;
    nm1m2 = ns + m1 + m2;

    gext = -1;
    a1 = dmatrix(1,mp,1,np);
    get_LP(a1,fn,obase, thetafdot, gext, palm_base, h2);
```

```
  non_basic = ivector(1,ns);
  basic = ivector(1,ms);
  svar = ivector(1,ns);
  j=1;
  for(i=1;i<=6;i++)
   if(fn->n_ind[i]==1)
    {svar[j]=i; j++;}

  simplx(a1,ms,ns,m1,m2,m3,&icase,non_basic,basic);

  if (icase == -1)
   { done = 1;}

  go = 1;
  for(i=1;i<=3;i++)
    if(basic[i]>ns)
       { go=0; break;}

  for(i=1;i<=3;i++)
    { if(go)
        c[i-1] = svar[basic[i]];
      if(a1[i+1][1]<=0)
       done = 1;
    }

  free_dmatrix(a1,1,mp,1,np);
  free_ivector(basic,1,ms);
  free_ivector(svar,1,ns);
  free_ivector(non_basic,1,ns);
 }

if(!done)
 {
  if(!edge_search)
   {
    thetaf[0][0] += thetafdot[0][0];
    thetaf[1][0] += thetafdot[1][0];
   }
  else
   {
    if(inc1) thetaf[0][0]+= tdiff;
    if(inc2) thetaf[1][0]+= tdiff;
    if(dec1) thetaf[0][0]-= tdiff;
```

```
    if(dec2) thetaf[1][0]-= tdiff;
   }


   move_all(h1, h2, thetaf, thetap, palm_base, A,B,D,T_frame, num_fingers,
          num_finger_links, hand_list, ci, object, b0, c, &obase, &otheta,
          tobject, object_list, &done);

if(!done)
 {
  if(placef1)
   solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
      A,B,D,T_frame, hand_list, 1, &done);

  if(placef2)
   solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
      A,B,D,T_frame, hand_list, 2, &done);

  if(thetaf[0][0]<=20 || thetaf[1][0] >=160)
    done = 1;
 }

  if(!done)
   {
    contact_normals(ci, object, hand_list, object_list,
          h1,tol, &legal,L,fn, obase);

    get_cf(ci, cf);
    samecf = test_for_samecf(oldcf,cf);

    form_closure(thetaf, fn, fc, &nfc1);


    if(legal && !samecf)
     { samecfnum = same_cf_numbers(oldcf,cf);
       if(samecfnum)
         legal = 0;
     }
   }
  if(done)
    legal = 0;

  if(legal)
    {
```

```
    if((cf[1].type == 0 || cf[0].type == 0)&&
       (equivelant(fn->c[1],fn->c[3],0.5) ||
        equivelant(fn->c[1],fn->c[4],0.5)))
         {legal = 0; done = 1;}
    if((cf[2].type == 0 || cf[0].type == 0) &&
         equivelant(fn->c[2],fn->c[3],0.5))
         {legal = 0; done = 1;}
  }

good = good_cf(cf);

if((!samecf || nfc1) && legal && ci->N>=3 && L[0]>0 && good)
 {
   p1[ind] = root1[ind];  match1[ind]=0;
   while(p1[ind]->row <=
                 row1[ind] && p1[ind]->next != NULL)
      p1[ind]=p1[ind]->next;

      do{
        if(p1[ind]->back == v1[ind])
          match1[ind] = new_nodes_equal(p1[ind],cf);
        if(p1[ind]->next!=NULL)
          p1[ind]=p1[ind]->next;
        }while(p1[ind]->next!=NULL && !match1[ind]);

      if(!match1[ind])
       if(p1[ind]->back == v1[ind])
        match1[ind] = new_nodes_equal(p1[ind],cf);

      p1[ind] = v1[ind];
      if(!match1[ind] && p1[ind]->back!=NULL)
       do{
          match1[ind] = new_nodes_equal(p1[ind]->back,cf);
          if(match1[ind])
           match1[ind]    =    equivelant(p1[ind]->back->obase,
                                   obase,1.0);
          if(p1[ind]->back!=NULL)
           p1[ind] = p1[ind]->back;
         }while(p1[ind]!=root1[ind] && !match1[ind]);

      if(!match1[ind] && v1[ind]->node_number>3)
       match1[ind] = new_nodes_equal(root1[ind],cf);

   if(!match1[ind])
```

```
    {
    sprintf(text,"%s %d %s %d\n","node #", v1[ind]->node_number,
        "heapsize",heapsize[ind]);
    _outtext(text);

    if(!edge_search)
      stype =0;
    else
     {
      if(inc1)
        stype = 1;
      else if(dec1)
        stype = -1;
      else if(inc2)
        stype = 2;
      else if(dec2)
        stype = -2;
     }

    call_ps (hand_list, object_list, filename,                    fileptr);
     fprintf(file1,"%d  %d  %d\n",  tree,  heapsize[ind],
              v1[ind]->node_number);

    n1[ind] = addnode(cf, obase, otheta, thetaf, v1[ind], v1[ind]->row+1,
          heapsize[ind]++, thetafdot, c, tree, stype);
    current1[ind]->next = n1[ind];
    current1[ind] = n1[ind];
   }
   done=1;

 }
if(graph==’y’)
  draw_all(h2,  tobject,  ci,  fn,  obase,  object_list,
              hand_list, y_pixels, sz, size);

if((ci->N<3 || L[0] ==0)&&legal)
  done = 1;

if(legal)
 {
  tdiff = 1.5;
  if(nfc1)
    done = 1;
 }
```

```
        else
         {
          for(i=0;i<2;i++)
           for(j=0;j<2;j++)
            thetaf[i][j] = oldthetaf[i][j];
          otheta = oldotheta;
          obase = oldobase;
          copy_cf(oldcf,cf);
          copy_fn(oldfn,fn);
          legal = 1;

        move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
              palm_base, A,B,D,T_frame, num_fingers, num_finger_links,
              hand_list, object_list, tobject, ci, tol, &legal, L, fn);

          for(i=0;i<2;i++)
           for(j=0;j<2;j++)
            thetafdot[i][j] *= 0.5;
          tdiff*= 0.5;
          if(tdiff<(LDIFF))
            done = 1;
         }


       } /* if done */

     } /* end while !done */

    if(edge_search && n==2)
      {n5++; n=0;}
    else if(!edge_search)
     {n4++;
      if(n4==SPOKES)
       cont = 0;}
   } /* end while cont */
  } /* end if nfc */

  if(v1[ind]->next != NULL)
    v1[ind] = v1[ind]->next;
  else
    {sprintf(text,"end of tree"); _outtext(text);}
 } /* while row */
 row1[ind]++;
}
```

```
    }  /* end if !match */
   } /* end while !match */

sprintf(text,"Found a path.\n"); _outtext(text);
end_ps (filename, fileptr);
getch();

    if(match)
     {
      path = current1[0];
      current1[0]->next = current1[1];

      while(current1[0]->row != 1)
       {
         current1[0] = current1[0]->back;
         current1[0]->next = path;
         path = current1[0];
       }
      path = current1[1];
      v1[0] = current1[1];

      while(current1[1]->row != 1)
       {
         current1[1] = current1[1]->back;
         path->next = current1[1];
         path = current1[1];
       }
      path = v1[0];
      current1[1] = path;
      while(current1[1] != root1[1])
       {
         current1[1] = current1[1]->next;
         current1[1]->back = path;
         path = current1[1];
       }
    } /* end if match */

/********* follow the path from root1[0] ***************/
   begin_ps (filename, fileptr);

   file1 = fopen("verthist.dat","w");
   file2 = fopen("object.dat","w");
   file3 = fopen("wrenches.dat","w");
   answer = 'y';
```

```
do{
path = root1[0];
for(i=0;i<2;i++)
 for(j=0;j<2;j++)
   thetaf[i][j] = path->thetaf[i][j];
obase = path->obase;
otheta = path->otheta;
move_all_get_contacts(object,obase,otheta,h1,h2,thetaf,thetap,
        palm_base,A,B,D,T_frame,num_fingers,num_finger_links,
        hand_list,object_list,tobject,ci,tol,&legal,L,fn);

get_cf(ci, cf);

draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3, palm_base,
          h2, hand_list, object_list, filename, fileptr);


print_path(path);
do{
     lastnode = path->tree;
     path = path->next;
     if(abs(path->stype) == 1)
      {placef1 = 0; placef2 = 1;}
     if(abs(path->stype) == 2)
      {placef1 = 1; placef2 = 0;}
     if(path->stype == 0)
      {placef1 = 0; placef2 = 0;}

  if(lastnode==path->tree)
  {
     if(path->tree == 1)
     {
       sprintf(text,"%s\n","in tree one.");
       _outtext(text);

     if(path->stype != 0)
     {

     do{
        solve_move_finger(thetaf, h1, h2, palm_base,tobject,
              thetap, A,B,D,T_frame, hand_list, 1, &done);

        solve_move_finger(thetaf, h1, h2, palm_base,tobject,
```

```
                thetap, A,B,D,T_frame, hand_list, 2, &done);


  if(path->stype==1) thetaf[0][0]+=1.0;
  else if(path->stype==-1) thetaf[0][0]-=1.0;
  else if(path->stype==2) thetaf[1][0]+=1.0;
  else if(path->stype==-2) thetaf[1][0]-=1.0;

  move_all(h1, h2, thetaf, thetap,palm_base,A,B,D,T_frame, num_fingers,
     num_finger_links, hand_list, ci, object, b0, path->c, &obase, &otheta,
     tobject, object_list, &done);

  if(placef1)
    solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
      A,B,D,T_frame, hand_list, 1, &done);

  if(placef1)
    solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
      A,B,D,T_frame, hand_list, 2, &done);


  contact_normals(ci, object, hand_list, object_list, h1, tol, &legal, L, fn,
                obase);

  draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
  solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
       palm_base, h2, hand_list, object_list, filename, fileptr);

  sprintf(text,"%f %f\n", thetaf[0][0], path->thetaf[0][0]);
  _outtext(text);
  }while(fabs(thetaf[0][0] - path->thetaf[0][0])>0.5);
}
else if(path->stype==0)
{
do{
  thetaf[0][0] += path->thetafdot[0][0];
  thetaf[1][0] += path->thetafdot[1][0];

 move_all(h1, h2, thetaf, thetap, palm_base,A,B,D,T_frame, num_fingers,
  num_finger_links, hand_list, ci,object, b0, path->c, &obase, &otheta,
  tobject, object_list, &done);

  }while(fabs(thetaf[0][0] - path->thetaf[0][0])>0.5);
}
```

```
        for(i=0;i<2;i++)
         for(j=0;j<2;j++)
          thetaf[i][j] = path->thetaf[i][j];
        obase = path->obase;
        otheta = path->otheta;

      move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap, palm_base,
          A,B,D,T_frame, num_fingers, num_finger_links, hand_list, object_list,
          tobject, ci, tol, &legal, L, fn);

        get_cf(ci, cf);

        draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
        solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
            palm_base, h2, hand_list, object_list, filename, fileptr);


        print_path(path);
        }
/***************** animate tree two ************************/
       else if(path->tree == 2)
       {
       print_path(path);
        path = path->back;
        if(path->stype != 0)
        {
        do{
          if(path->stype==-1) thetaf[0][0]+=1.0;
          else if(path->stype==1) thetaf[0][0]-=1.0;
          else if(path->stype==-2) thetaf[1][0]+=1.0;
          else if(path->stype==2) thetaf[1][0]-=1.0;

         move_all(h1, h2, thetaf, thetap, palm_base, A,B,D,T_frame,
           num_fingers, num_finger_links, hand_list, ci, object, b0, path->c, &obase,
           &otheta, tobject, object_list, &done);


         if(placef1)
           solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
             A,B,D,T_frame, hand_list, 1, &done);

         if(placef2)
           solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
             A,B,D,T_frame, hand_list, 2, &done);
```

```
  contact_normals(ci,object,hand_list,object_list,h1,tol,
                  &legal,L,fn, obase);

  draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);

  solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
    palm_base, h2, hand_list, object_list, filename, fileptr);


 }while(fabs(thetaf[0][0]-path->next->thetaf[0][0])>0.5);
}
else if(path->stype==0)
{
  move_all_get_contacts(object, obase, otheta, h1, h2,
          thetaf, thetap, palm_base, A,B,D,T_frame,
          num_fingers, num_finger_links, hand_list,
          object_list, tobject, ci, tol, &legal, L, fn);

do{
  thetaf[0][0] -= path->thetafdot[0][0];
  thetaf[1][0] -= path->thetafdot[1][0];

  print_path(path);
  getch();
  print_path(path->next);

  c[0] = 1; c[1] = 2; c[2] = 3;

 move_all(h1, h2, thetaf, thetap, palm_base, A,B,D,T_frame, num_fingers,
    num_finger_links, hand_list, ci, object, b0, c, &obase, &otheta, tobject,
    object_list, &done);

  }while(fabs(thetaf[0][0]-path->next->thetaf[0][0])>0.5);
}
path = path->next;
for(i=0;i<2;i++)
 for(j=0;j<2;j++)
  thetaf[i][j] = path->thetaf[i][j];
obase = path->obase;
otheta = path->otheta;
move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap, palm_base,
  A,B,D,T_frame, num_fingers, num_finger_links, hand_list, object_list,
```

```
    tobject, ci, tol, &legal, L, fn);

  get_cf(ci, cf);

  draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
  solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
    palm_base, h2, hand_list, object_list, filename, fileptr);


  print_path(path);

 }
}
else
{
  connect_tree_parameters(path, thetaf, c, &edge, thetafdot, &placef1,
    &placef2, &inc1, &inc2, &dec1, &dec2);
  sprintf(text,"Connecting trees.\n");_outtext(text);

  if(edge)
   {
   do{
      if(inc1) thetaf[0][0]+= 1.0;
      if(inc2) thetaf[1][0]+= 1.0;
      if(dec1) thetaf[0][0]-= 1.0;
      if(dec2) thetaf[1][0]-= 1.0;

    move_all(h1, h2, thetaf, thetap, palm_base, A,B,D,T_frame, num_fingers,
     num_finger_links, hand_list, ci, object, b0, c, &obase, &otheta, tobject,
     object_list, &done);


     if(placef1)
       solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
         A,B,D,T_frame, hand_list, 1, &done);

     if(placef2)
       solve_move_finger(thetaf, h1, h2, palm_base, tobject, thetap,
         A,B,D,T_frame, hand_list, 2, &done);


     contact_normals(ci,object,hand_list,object_list,h1,tol,
                     &legal,L,fn, obase);
```

```
    draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);

solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
    palm_base, h2, hand_list, object_list, filename, fileptr);


    }while(fabs(thetaf[0][0] - path->thetaf[0][0])>0.5);
  }
else
  {
   move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap,
    palm_base, A,B,D,T_frame, num_fingers, num_finger_links, hand_list,
    object_list, tobject, ci, tol, &legal, L, fn);

   contact_normals(ci,object,hand_list,object_list,h1,tol,
                   &legal,L,fn, obase);

   draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
   solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
     palm_base, h2, hand_list, object_list, filename, fileptr);


   diff1 = fabs(thetaf[0][0] - path->thetaf[0][0]);
   diff2 = fabs(thetaf[1][0] - path->thetaf[1][0]);
   if(diff1 > diff2)
    {
     thetafdot[1][0] = 1;
     thetafdot[0][0] = diff1/diff2;
    }
   else
    {
     thetafdot[0][0] = 1;
     thetafdot[1][0] = diff2/diff1;
    }
   if(thetaf[0][0] > path->thetaf[0][0])
     thetafdot[0][0] = -thetafdot[0][0];
   if(thetaf[1][0] > path->thetaf[1][0]);
     thetafdot[1][0] = -thetafdot[1][0];

   do{
      thetaf[0][0] += thetafdot[0][0];
      thetaf[1][0] += thetafdot[1][0];

move_all(h1, h2, thetaf, thetap, palm_base, A,B,D,T_frame, num_fingers,
```

```
        num_finger_links, hand_list, ci, object, b0, c,&obase, &otheta, tobject,
        object_list, &done);

      contact_normals(ci,object,hand_list,object_list,h1,tol,
                        &legal,L,fn, obase);

      draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
      solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
                    palm_base, h2, hand_list, object_list, filename, fileptr);


        }while(fabs(thetaf[0][0] - path->thetaf[0][0])>0.5);
       }

      for(i=0;i<2;i++)
       for(j=0;j<2;j++)
        thetaf[i][j] = path->thetaf[i][j];
      obase = path->obase;
      otheta = path->otheta;
      move_all_get_contacts(object, obase, otheta, h1, h2, thetaf, thetap, palm_base,
        A,B,D,T_frame, num_fingers, num_finger_links, hand_list, object_list,
        tobject, ci, tol, &legal, L, fn);

      get_cf(ci, cf);

      draw_all(h2, tobject, ci, fn, obase, object_list, hand_list, y_pixels, sz, size);
      solve_force(object, ci->N, ci, fn, obase, thetaf, otheta, file1, file2, file3,
        palm_base, h2, hand_list, object_list, filename, fileptr);

      print_path(path);

     }
    }while(path != root1[1]);
    sprintf(text,"Would you like to see results again?(y/n)");
    _outtext(text);
    answer = getch();
    sprintf(text,"%c\n",answer);_outtext(text);
    }while(answer==='y');
    fclose(file1);
    fclose(file2);
    fclose(file3);
    end_ps (filename, fileptr);

  /* done */
```

```
    fclose(file1);

    tmode(_DEFAULTMODE);
}
```

The contents of compare.c is:

```
#include "minclude.h"
\*********************************************************\
\**********  Equivelant determines if two *****************\
\**********  cartesian points are equal  ******************\
\*********************************************************\
int
equivelant(pt_1, pt_2, tolerance)
   point          pt_1,
                  pt_2;
   double          tolerance;
{
   double d,dist();
   d = dist(pt_1, pt_2);
   if (d <= tolerance)
       return (1);
   return (0);
}
\*********************************************************\
\*******  between points determines if point ***************\
\*******  3 is between point 1, and point 2  ***************\
\*********************************************************\
int
between_pts(point pt_1,point pt_2,point pt_3,double tolerance)
/*  is pt_1 between pt_2 and pt_3 */
{
   double          minf_x,
                  minf_y,
                  maxf_x,
                   maxf_y,
                   minf(),
                   maxf(),
                  fabs();

   minf_x = minf(pt_2.x, pt_3.x);
   maxf_x = maxf(pt_2.x, pt_3.x);
   minf_y = minf(pt_2.y, pt_3.y);
   maxf_y = maxf(pt_2.y, pt_3.y);
   if(fabs(minf_x - pt_1.x) <= tolerance)
      pt_1.x = minf_x;
   if(fabs(maxf_x - pt_1.x) <= tolerance)
      pt_1.x = maxf_x;
   if(fabs(minf_y - pt_1.y) <= tolerance)
```

```
        pt_1.y = minf_y;
    if(fabs(maxf_y - pt_1.y) < = tolerance)
        pt_1.y = maxf_y;
    if (((pt_1.x > = minf_x) && (pt_1.x < = maxf_x) &&
        (pt_1.y > = minf_y) && (pt_1.y < = maxf_y)))
        return (1);
    return (0);
}
```

These routines, in initial.c, initialize all the variables used by the simulation.

```
#include "minclude.h"

/********* INITIALIZATION ROUTINES ************/
/**************************************************/
/* Initialize screen determines which graphics */
/* card the PC is running on, then initializes */
/* the program accordingly.                    */
/**************************************************/
void
initialize_screen(int *x_pixels, int *y_pixels)
{
   int    cd;
   int    video_mode[11],
          i = 0;
   struct videoconfig vc;

   video_mode[7] = 6;
   video_mode[8] = 5;
   video_mode[9] = 4;

   video_mode[0] = 18;
   video_mode[1] = 17;
   video_mode[2] = 16;
   video_mode[3] = 15;
   video_mode[4] = 14;
   video_mode[5] = 19;
   video_mode[6] = 13;
   video_mode[10] = -1;

   while (_setvideomode(video_mode[i]) == 0)
     i++;
   if (video_mode[i] == _DEFAULTMODE)
   {
      fprintf(stderr, "%s\n", "This machine does not have                graphics
        capability.");
      exit(1);
   }
   _getvideoconfig(&vc);
   *x_pixels = vc.numxpixels;
   *y_pixels = vc.numypixels;
}
/**********************************************/
```

```
/*  Initialize polygon creates the pointer   */
/*  to the structure for storing polygons.   */
/*********************************************/
p_ptr
initialize_polygon(p_ptr p,int vert_num)
{
   int     i;

   p = (p_ptr) malloc(sizeof(struct polygon));
   p->vert_num = vert_num;
   for (i = 0; i <= vert_num; i ++)
   {
     p->vertex[i].x = 0.0;
     p->vertex[i].y = 0.0;
   }
return(p);
}

void
free_polygon(p_ptr p)
{
  free(p->vertex);
  free(p);
}
/*********************************************/
/*  Initialize finger creates the pointer    */
/*  to the structure for storing fingers.    */
/*********************************************/
struct finger *
initialize_finger(struct finger *f,int links,int *verteces,int finger_num)
{
       int i;
       void m_initialize();
       double    **m_create();

       f = (struct finger *)malloc(sizeof(struct finger));
       f->num_links = links;
       for (i = 0; i < links; i++)
       {
           f->D[i] = (rt_ptr)malloc(sizeof(struct
                   relative_transformations));
           m_initialize(f->D[i]->rt, 3, 3);
           f->link[i] = initialize_polygon(f->link[i],
                   verteces[i]);
```

```
        }
        return(f);

}

void
free_finger(struct finger *f)
{
    int j;
    void    m_free();

    for (j = 0; j < f->num_links; j++)
    {
        free_polygon(f->link[j]);
        m_free(f->D[j]->rt, 3);
    }
    free(f->theta_i);
    free(f->b_i);
    free(f->link);
    free(f);

}
void
m_initialize(double t[3][3],int r,int c)
{
    int i,
        j;

    for (i = 0; i < r; i ++)
        for (j = 0; j < c; j++)
            t[i][j] = 0.0;
}

rt_ptr
**initialize_rt_array2(rt_ptr **X,int r,int c)
{
    int    i,
           j,
           k;

    double **m_create();
    void   m_initialize();

    X = (rt_ptr **)calloc(r+1, sizeof(rt_ptr *));
```

```
    for (i = 0; i <= r; i++)
        X[i] = (rt_ptr *)calloc(c+1, sizeof(rt_ptr));
    for (i = 0; i <= r; i++)
        for (j = 0; j <= c; j++)
        {
            X[i][j] = (rt_ptr)malloc(sizeof(struct
                    relative_transformations));
            m_initialize(X[i][j]->rt, 3, 3);
        }
    return(X);
}

void
free_rt_array2(rt_ptr **X,int r,int c)
{
    int     i,
            j,
            k;

    void    m_free();

    for (i = 0; i <= r; i++)
    {
        for (j = 0; j <= c; j++)
        {
            m_free(X[i][j]->rt, 3);
            free(X[i][j]);
        }
        free(X[i]);
    }
}

h_ptr
initialize_hand(h_ptr hand,int num_fingers,int *links,int **fverteces,int
pverteces,rt_ptr *finger_placement,
struct cartesian_point base)
{
    int i;
    double  **m_create();
    void    m_copy();

    hand = (struct hand *)malloc(sizeof(struct hand));
    hand->finger_num = num_fingers;
        hand->palm = initialize_polygon(hand->palm,
```

```
            pverteces);

        for (i = 0; i < num_fingers; i++)
        {
            hand->finger_location[i] =
           (rt_ptr)malloc(sizeof(struct
                relative_transformations));
            m_copy(finger_placement[i]->rt,
              hand->finger_location[i]->rt, 3, 3);
            hand->finger[i] = initialize_finger(hand->finger[i],
                        links[i], fverteces[i], i) ;
        }
        return(hand);
}

void
free_hand(h_ptr hand)
{
    int j;

    for (j = 0; j < hand->finger_num; j++)
          free_finger(hand->finger[j]);
    free_polygon(hand->palm);
    free(hand->finger_location);
    free(hand);
}

ci_ptr
initialize_ci(p_ptr obj,ci_ptr ci)
{  int  i;

  for(i=0; i<obj->vert_num*2; i++)
   {
    ci->v[i].contact = 0;
    ci->v[i].contact_with = -1;
    ci->v[i].hvertex = -1;
    ci->v[i].overtex = -1;
    ci->v[i].B.x = 0.0;
    ci->v[i].B.y = 0.0;
    ci->v[i].WB.x = 0.0;
    ci->v[i].WB.y = 0.0;
    ci->v[i].A1.x = 0.0;
    ci->v[i].A1.y = 0.0;
    ci->v[i].WA1.x = 0.0;
```

```
    ci->v[i].WA1.y = 0.0;
    ci->v[i].A2.x = 0.0;
    ci->v[i].A2.y = 0.0;
    ci->v[i].WA2.x = 0.0;
    ci->v[i].WA2.y = 0.0;
    ci->e[i].contact = 0;
    ci->e[i].contact_with = -1;
    ci->e[i].hvertex = -1;
    ci->e[i].overtex = -1;
    ci->e[i].B.x = 0.0;
    ci->e[i].B.y = 0.0;
    ci->e[i].WB.x = 0.0;
    ci->e[i].WB.y = 0.0;
    ci->e[i].A1.x = 0.0;
    ci->e[i].A1.y = 0.0;
    ci->e[i].WA1.x = 0.0;
    ci->e[i].WA1.y = 0.0;
    ci->e[i].A2.x = 0.0;
    ci->e[i].A2.y = 0.0;
    ci->e[i].WA2.x = 0.0;
    ci->e[i].WA2.y = 0.0;

   }
   ci->N = 0;
   ci->vert_num = obj->vert_num;
   return(ci);
}

ni initialize_fn(ni fn)
{ int   i,j;
  for(i=0; i<7; i++)
   {
     fn->n_ind[i] = 0;
     for(j=0; j<7; j++)
       {
         fn->q_ind[i][j] = 0;
       }
   }
   return(fn);
}
# define PI 3.141559265359
/*************************************************************/
/*  Place object sets the object on the palm of the hand  */
/*  the edge that will set on the palm is sent in j.      */
```

```
/*********************************************************/
void place_object(p_ptr object, double x[3],point palm_base,
                          int j)
{
  int     k,s;
  double  lineA[3],lineB[3],lineCG[3],
          distA,s1,s2,dist_line(),a[2][3],xa;
  point   zero,m;

  j--;
  line(object->vertex[j],object->vertex[j+1],lineA);
  zero.x=0; zero.y=0;
  distA = dist_line(zero,lineA);
  x[0] = palm_base.x;
  x[1] = palm_base.y+5+fabs(distA);

  lineCG[0]=1;  lineCG[1]=0;  lineCG[2]=0;
  m.x = 10*lineA[0];
  m.y = 10*lineA[1];

  line(zero,m,lineB);

  if(lineB[0]!=0)
   { s1 = -lineB[1]/lineB[0];
     x[2] = atan(s1) * 180/PI;}
  else if(-lineB[1] < 0)
   x[2] = -90;
  else
   x[2] = 90;

  for(k=0;k<3;k++)
   {
    a[0][k] = lineB[k];
    a[1][k] = lineA[k];
   }
  gauss(2,1,a,1.0e-6,&s);
  xa = -a[0][2];

  if(xa>0 && x[2]>0)
    x[2] = x[2] - 180;
  if(xa<0 && x[2]<0)
    x[2] = x[2] + 180;
}
/*********************************************************/
```

```
#include "minclude.h"
double
minf(double x,double y)
{
   if (x < y)
       return (x);
   return (y);
}

double
maxf(double x,double y)
{
   if (x > y)
       return (x);
   return (y);
}

int
sign(double x)
{
   if (x < 0)
       return (-1);
   return (1);
}

double
dist(point pt_1, point pt_2)
{
   double          d;

   d = sqrt(pow((pt_1.x - pt_2.x), 2.0) +
            pow((pt_1.y - pt_2.y), 2.0));
   return (d);
}

void
line(point pt_1, point pt_2, double coef[3])
{
   double          delta[2],
                   dist;
   int             i;

   delta[0] = pt_1.x - pt_2.x;
   delta[1] = pt_1.y - pt_2.y;
```

```
    dist = sqrt(pow(delta[0], 2.0) + pow(delta[1], 2.0));
    coef[0] = -delta[1] / dist;
    coef[1] = delta[0] / dist;
    coef[2] = (pt_1.x * delta[1] - pt_1.y * delta[0]) / dist;
    return;
}

double
dist_line(point pt, double coef[3])
{
    double          d;

    d = pt.x * coef[0] + pt.y * coef[1] + coef[2];
    return (d);
}

void
m_mult(double a[3][3],double b[3][3])
{
    int             i,
                    j,
                    k;
    double          c[3][3];
    void            m_initialize(),
                    m_free();

    m_initialize(c,3,3);
    for(i = 0; i < 3; i++)
        for(j = 0; j < 3; j++)
            for(k = 0; k < 3; k++)
                c[i][j] += a[i][k] * b[k][j];
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            a[i][j] = c[i][j];
}
/*************************************************************
* gm_mult - This routine multiplies two matrices of any size*
* together and returns a double pointer to the result.      *
* Usage:                                                    *
*       a = gm_mult(a, r1, c1, b, c2)                       *
*       External Functions/Procedures: m_create, m_free,    *
*            m_initialize                                   *
* Parameters:                                               *
*       Input:                                              *
```

```
*          a  = matrix a                                  *
*          b  = matrix b                                  *
*          r1 = row of matrix 1                             *
*          c1 = col of matrix 1                           *
*            c2 = col of matrix 2                         *
*                                                                                *
*       Output:                                         *
*            axb is returned in a                         *
* Parameter Declaration:                                    *
*       Declare:  a, b as double **                       *
*                 r1, c1, c2 as int                     *
*****************************************************************/
double
**gm_mult(double **a,int r1,int c1,double **b,int c2)
{
   int          i,
                j,
                k;
   double         **c,
                **m_create(),
                temp1,
                temp2;
   void          m_initialize(),
                m_free();

   c = m_create(c, r1, c2);
   m_initialize(c, r1, c2);
   for(i = 0; i < r1; i++)
      for(j = 0; j < c2; j++)
         for (k = 0; k < c1; k++)
         {
            temp1 = a[i][k];
            temp2 = b[k][j];
            c[i][j] += (temp1 * temp2);
         }
   m_free(a, r1);
   a = m_create(a, r1, c2);
   for (i = 0; i < r1; i++)
      for (j = 0; j < c2; j++)
         a[i][j] = c[i][j];
  m_free(c, r1);
  return(a);
}
```

```
/*************************************************************
 * invg - This procedure computes the inverse matrix A       *
 *        using gaussian elimination.                        *
 * Usage:                                                    *
 *        invg(N,A,Delt,B);                                  *
 *        External Functions/Procedures: procedure gauss     *
 * Parameters:                                               *
 *        Input:                                             *
 *             N = order of the matrix                       *
 *             A = N by N array of matrix values             *
 *             Delt = machine zero (tolerance)               *
 *        Output:                                            *
 *             B = N by N array of inverse matrix            *
 * Parameter Declaration:                                    *
 *        Declare A as an N by N array of doubles.           *
 *************************************************************/

void
invg(N, A, Delt, B)
   int          N;
   double       **A,
               **B,
                Delt;
{
   int          i,
                j,
                M,
                two_N;
   double       **C;
   void         ggauss();

   /* Compute working matrix C from matrix A and the appended identity matrix
      number of columns of matrix C is 2*N */
   C = (double **) malloc((unsigned) N * sizeof(double *));
   two_N = 2 * N;
   for (i = 0; i < N; ++i)
   {
      C[i] = (double *)
           malloc((unsigned) two_N * sizeof(double));
      for (j = 0; j < N; ++j)
      {
         C[i][j] = A[i][j];
         C[i][N + j] = ((i == j) ? 1.0 : 0.0);
      }
```

```
    }
    /* compute matrix inverse by Gaussian Elimintaion */
    M = N;
    ggauss(N, M, C, Delt);
    for (i = 0; i < N; ++i)
    {
        for (j = 0; j < N; ++j)
            B[i][j] = C[i][j + N];
        free((char *) C[i]);
    }
    free((char *) C);
}

void
ggauss(N, M, A, Delt)
    int             N,
                    M;
    double          **A,
                    Delt;
{
    double          pivot,
                    tmp;
    int             index,
                    i,
                    j,
                    k,
                    MAXINDEX;

    MAXINDEX = N - 1;
    if (N > 1)
    {
        for (k = 0; k < MAXINDEX; ++k)
        {
            pivot = (double) fabs((double) A[k][k]);
            index = k;
            /* search for index of maximum pivot value */
            for (i = (k + 1); i < N; ++i)
                if ((tmp = (double) fabs((double) A[i][k])) > pivot)
                {
                    pivot = tmp;
                    index = i;
                }
            if (k != index)
                /* interchange rows k and index */
```

```
                for (j = k; j < (M + N); ++j)
                {
                    tmp = A[k][j];
                    A[k][j] = A[index][j];
                    A[index][j] = tmp;
                }

            /* check if pivot too small */
            if (pivot < Delt)
            {
                singularity();
                return;
            }
            /* forward elimination step */
            for (i = (k + 1); i < N; ++i)
                for (j = (k + 1); j < (M + N); ++j)
                    A[i][j] -= A[i][k] * A[k][j] / A[k][k];
        }
        if (((double) fabs((double) A[MAXINDEX][MAXINDEX])) < Delt)
        {
            singularity();
            return;
        }
        /* back substitution */
        for (k = 0; k < M; ++k)
        {
            for (i = MAXINDEX; i >= 0; --i)
            {
                for (j = i + 1; j < N; ++j)
                    A[i][k + N] -= A[j][k + N] * A[i][j];
                A[i][k + N] /= A[i][i];
            }
        }
    } else
    if,(((double) fabs((double) A[0][0])) < Delt)
    {
        singularity();
        return;
    } else
    for (j = 0; j < M; ++j)
        A[0][N + j] /= A[0][0];
}

singularity()
```

```
{   char text[80];
    sprintf(text, "\nThe Matrix is Singular.  Gaussian");
    _outtext(text);
    sprintf(text, "Elimination cannot be performed.\n");
    _outtext(text);
}

double **pinv(double **a,int r, int c,double **pa)
{
 double **at,**at1,**tt,**t,**gm_mult(),**aa,**m_create();
 int i,j,k;

 at = m_create(at,c,r);
 at1 = m_create(at1,c,r);
 t = m_create(t,r,r);
 tt = m_create(tt,r,r);
 aa = m_create(aa,r,c);

  for(i=0;i<r;i++)
   for(j=0;j<c;j++)
    {
     aa[i][j] = a[i][j];
     at[j][i] = a[i][j];
     at1[j][i] = a[i][j];
    }
/*  t = gm_mult(aa,3,4,at1,3);*/
  for(i = 0; i < r; i++)
   for(j = 0; j < r; j++)
    for (k = 0; k < c; k++)
      t[i][j] += (aa[i][k] * at1[k][j]);

  invg(3,t,1.0e-6,tt);

/*  pa = gm_mult(at,4,3,tt,3);*/
  for(i = 0; i < c; i++)
   for(j = 0; j < r; j++)
    for (k = 0; k < r; k++)
      pa[i][j] += (at[i][k] * tt[k][j]);

 m_free(at,c);
 m_free(at1,c);
 m_free(t,r);
 m_free(tt,r);
 m_free(aa,r);
```

```
  return(pa);
}
#undef N

void
gauss(int N,int M,double a[2][3],double Delt,int *s)
{
   double          pivot,
                   tmp;
   int             index,
                   i,
                   j,
                   k,
                   MAXINDEX;

   MAXINDEX = N - 1;
   if (N > 1)
   {
      for (k = 0; k < MAXINDEX; ++k)
      {
         pivot = (double) fabs((double) a[k][k]);
         index = k;
         /* search for index of maximum pivot value */
         for (i = (k + 1); i < N; ++i)
            if ((tmp = (double) fabs((double) a[i][k])) > pivot)
            {
               pivot = tmp;
               index = i;
            }
         if (k != index)
            /* interchange rows k and index */
            for (j = k; j < (M + N); ++j)
            {
               tmp = a[k][j];
               a[k][j] = a[index][j];
               a[index][j] = tmp;
            }

         /* check if pivot too small */
         if (pivot < Delt)
         {
            /*singularity();*/
            *s = 1;
            return;
```

```
            }
            /* forward elimination step */
            for (i = (k + 1); i < N; ++i)
                for (j = (k + 1); j < (M + N); ++j)
                    a[i][j] -= a[i][k] * a[k][j] / a[k][k];
        }
        if (((double) fabs((double) a[MAXINDEX][MAXINDEX])) < Delt)
        {
            /*singularity();*/
            *s = 1;
            return;
        }
        /* back substitution */
        for (k = 0; k < M; ++k)
        {
            for (i = MAXINDEX; i >= 0; --i)
            {
                for (j = i + 1; j < N; ++j)
                    a[i][k + N] -= a[j][k + N] * a[i][j];
                a[i][k + N] /= a[i][i];
            }
        }
    } else
    if (((double) fabs((double) a[0][0])) < Delt)
    {
      /*  singularity();*/
        *s = 1;
        return;
    } else
    for (j = 0; j < M; ++j)
        a[0][N + j] /= a[0][0];
}


p_ptr
T(p_ptr p1,p_ptr p2,double t[3][3])
{

    int         i;
    void        cm_mult();

    for (i = 0; i <= p1->vert_num; i++)
        p2->vertex[i] = p1->vertex[i];
```

```
    p2->vert_num = p1->vert_num;
    p2->name = p1->name;

    cm_mult(p2, t);
    return(p2);

}


void
coord_transform(double theta,point base,
                double t[3][3])
{
    double          c_theta,
                    s_theta;


    c_theta = cos((theta * 0.0174532925199));
    s_theta = sin((theta * 0.0174532925199));


    t[0][0] = c_theta;
    t[0][1] = -s_theta;
    t[1][0] = s_theta;
    t[1][1] = c_theta;

    t[0][2] = base.x;
    t[1][2] = base.y;
    t[2][0] = 0.0;
    t[2][1] = 0.0;
    t[2][2] = 1.0;

}
void
DH_transform(double b,double theta,double t[3][3])
{
    double          c_theta,
                    s_theta;


    c_theta = cos((theta * 0.0174532925199));
    s_theta = sin((theta * 0.0174532925199));

    t[0][0] = c_theta;
```

```
    t[0][1] = -s_theta;
    t[0][2] = b;

    t[1][0] = s_theta;
    t[1][1] = c_theta;
    t[1][2] = 0.0;

    t[2][0] = 0.0;
    t[2][1] = 0.0;
    t[2][2] = 1.0;

}

void
cm_mult(p_ptr p,double t[3][3])
{
    int             i,
                    j;
    double          x,
                    y,
                    z;

    for (i = 0; i <= p->vert_num; i++)
    {
        x = p->vertex[i].x;
        y = p->vertex[i].y;
        p->vertex[i].x = t[0][0] * x + t[0][1] * y + t[0][2];
        p->vertex[i].y = t[1][0] * x + t[1][1] * y + t[1][2];
    }
}

void
m_copy(double m1[3][3],double m2[3][3],int r,int c)
{
    int         i,
                j;

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            m2[i][j] = m1[i][j];
}

void
m_transpose(double m[3][3],double temp[3][3])
```

```
{
    int         i,
                j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            temp[i][j] = m[j][i];
}

void
m_negate(double **m,int r,int c)
{
    int         i,
                j;

    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            m[i][j] = -m[i][j];
}


double
**m_create(double **m,int r,int c)
{
    int   i,
          j;

    m = (double **)calloc(r, sizeof(double *));
    for (i = 0; i < r; i++)
        m[i] = (double *)calloc(c, sizeof(double));
    return(m);
}

void
m_free(double **m,int r)
{
    int   i;

    for (i = 0; i < r; i++)
        free(m[i]);
    free(m);
}
```

```
void
move_finger(f_ptr f1,f_ptr f2,double theta[2],double BA[3][3],
        rt_ptr *A,rt_ptr *B,rt_ptr *D,rt_ptr *T_frame)
{
   int i;
   double  t0_1[3][3],
           tj_k[3][3],
           save[3][3],
           save2[3][3],
           **m_create();

   struct polygon *T();

   void  m_mult(),
         m_initialize(),
         coord_transform(),
         DH_transform(),
         m_copy(),
         m_free();
   point
         null_coord;

   null_coord.x = 0.0;
   null_coord.y = 0.0;

   f2->num_links = f1->num_links;

   for (i = 0; i < f1->num_links; i++)
   {
       f2->link[i]->vert_num = f1->link[i]->vert_num;
       f2->theta_i[i] = f1->theta_i[i];
      f2->b_i[i] = f1->b_i[i];
   }
   m_initialize(t0_1, 3, 3);
   m_initialize(tj_k, 3, 3);
   m_initialize(save, 3, 3);
   m_initialize(save2, 3, 3);
   m_copy(BA, T_frame[0]->rt, 3, 3);

   DH_transform(0.0, theta[0], t0_1);
   m_mult(BA, t0_1);
   m_copy(BA, save, 3, 3);
   m_copy(BA, save2, 3, 3);
   m_copy(t0_1, tj_k, 3, 3);
```

```
    for (i = 0; i < f2->num_links; i++)
    {
        m_copy(tj_k, A[i+1]->rt, 3, 3);
        m_copy(save2, save, 3, 3);
        m_copy(save, T_frame[i+1]->rt, 3, 3);
       m_mult(save, f1->D[i]->rt);
        m_copy(save, B[i+1]->rt, 3, 3);
        m_copy(f1->D[i]->rt, D[i+1]->rt, 3, 3);
       f2->link[i] = T(f1->link[i], f2->link[i], save);
        m_initialize(tj_k, 3, 3);
        DH_transform(f1->b_i[i], theta[i+1], tj_k);
        m_mult(save2, tj_k);
    }
}

void
move_hand(h_ptr h1,h_ptr h2,double thetaf[2][2],double thetah,
        point base, rt_ptr **A,rt_ptr **B, rt_ptr **D, rt_ptr **T_frame)
{
    int i,
       j;
    double  B00[3][3],
            save[3][3],
            **m_create(),
            x,
            y,
            z;
    point
            null_c;
    void    m_free();

    null_c.x = 0.0;
    null_c.y = 0.0;

    m_initialize(save, 3, 3);
    m_initialize(B00, 3, 3);
    coord_transform(thetah, base, B00);
    m_copy(B00, save, 3, 3);
    m_copy(B00, B[0][0]->rt, 3, 3);
    h2->palm = T(h1->palm, h2->palm, B00);

    for (i = 0; i < h1->finger_num; i++)
    {
        m_mult(save, h1->finger_location[i]->rt);
```

```
        m_copy(h1->finger_location[i]->rt, A[i+1][0]->rt, 3, 3);
        move_finger(h1->finger[i], h2->finger[i], thetaf[i],
                save, A[i+1], B[i+1], D[i+1], T_frame[i+1]);
        m_copy(B00, save, 3, 3);
    }
}

void
move_object(p_ptr ob1,p_ptr ob2,double x,double y,double theta)
{

    double   t[3][3];
    point
            base;

    base.x = x;
    base.y = y;
    m_initialize(t, 3, 3);
    coord_transform(theta, base, t);
    ob2 = T(ob1, ob2, t);
}

void get_hand_list(h_ptr h2,int num_fingers,int num_finger_links[2],
                   p_ptr hand_list[7])
{       int i,j,k;

        k = 1;
        hand_list[0] = h2->palm;
        for (i = 0; i < num_fingers; i++)
            for (j = 0; j < num_finger_links[i]; j++)
            {
                hand_list[k] = h2->finger[i]->link[j];
                k++;
            }
}
```

These routines, ps.c, were written by Dr. Jeff Trinkle to output the hand and object figures to postscript files.

```
#include "minclude.h"

void begin_ps (char fname[10], FILE *fptr)

{   char text[80];
    sprintf (text,"Enter the name of the file to write
            postscript to.  \n");
    _outtext(text);
    scanf ("%s", fname);
    fptr = fopen (fname, "a");
    fprintf (fptr, "%%! PostScript generated by
                    write_ps.c\n");
    fprintf (fptr, "%%%%BoundingBox: (atend)\n\n");
    fclose (fptr);
}
void boxvp_ps (Lmtptr vp, char fname[10], FILE *fptr)

{

  int           ixvmax, ixvmin, iyvmax, iyvmin;
  double        xvmax, xvmin, yvmax, yvmin;

  fptr = fopen (fname, "a");
  xvmin = vp->xmin;
  yvmin = vp->ymin;
  xvmax = vp->xmax;
  yvmax = vp->ymax;

  ixvmin = (72. * xvmin + 0.5);
  ixvmax = (72. * xvmax + 0.5);
  iyvmin = (72. * yvmin + 0.5);
  iyvmax = (72. * yvmax + 0.5);

  fprintf (fptr, "newpath\n");
  fprintf (fptr, "    %d %d moveto\n", ixvmin, iyvmin);
  fprintf (fptr, "    %d %d lineto\n", ixvmin, iyvmax);
  fprintf (fptr, "    %d %d lineto\n", ixvmax, iyvmax);
  fprintf (fptr, "    %d %d lineto\n", ixvmax, iyvmin);
  fprintf (fptr, "    closepath\n");
  fprintf (fptr, "1 setlinewidth\n");
  fprintf (fptr, "stroke\n\n");
```

```
  fclose (fptr);
}
void end_ps (char fname[10], FILE *fp)

{
  fp = fopen (fname, "a");
  fprintf (fp, "showpage");
  fprintf (fp, "\n%%%%%BoundingBox: 0 0 612 792\n");
  fclose (fp);
}
double get_aspect (Lmtptr wlptr)
{
  double       xdiff, ydiff;

  xdiff = wlptr->xmax - wlptr->xmin;
  ydiff = wlptr->ymax - wlptr->ymin;
  return (ydiff / xdiff);
}
int set_vps (double asp, Lmtptr vl)

{
  double       vp_width, vp_height, gut_size;
  double       xdiff, ydiff;
  int          i, j, k, ncols, nrows;
  char         text[80];
  gut_size = 0.26;

  sprintf (text,"How many columns of frames on an 8.5x11 inch
                page?  ");
  _outtext(text);
  scanf ("%d", &ncols);
  vp_width  =  (8.5 - ((double) ncols + 1) * gut_size) /
              (double) ncols;
  vp_height =  vp_width * asp;
  nrows = 11 / (vp_height + gut_size);

  for (i = 0; i < nrows; i++) {
    for (j = 0; j < ncols; j++) {
      k = i * ncols + j;
      (vl+k)->xmax = (vp_width + gut_size) * (j + 1);
      (vl+k)->xmin = (vl+k)->xmax - vp_width;
      (vl+k)->ymin = 11 - (vp_height + gut_size) * (i + 1);
      (vl+k)->ymax = (vl+k)->ymin + vp_height;
    }
```

```
  }

  return (nrows * ncols);
}
double vp2w_scale (Lmtptr wlmts,  Lmtptr vlmts)

{
  double        xvmin, xvmax, xwmin, xwmax;

  xvmin = vlmts->xmin;
  xvmax = vlmts->xmax;
  xwmin = wlmts->xmin;
  xwmax = wlmts->xmax;

  return ((xvmax - xvmin) / (xwmax - xwmin));
}

void call_ps (p_ptr hand_list[7], p_ptr object_list,
char fname[10], FILE *fptr)
{
  void          boxvp_ps(),draw_ps();
  int           set_vps();
  double        get_aspect(), vp2w_scale();

  static int       vp_index = 0;
  static int       called_b4 = 0;
  static int       n_vps_on_page;
  static double          scale;
  static Limits    w_limits, v_limits[150];
  static Lmtptr    vlptr;

  double        aspect;
  Lmtptr        vlptr2;
  char          text[80];

  if (!called_b4) {


    called_b4 = 1;

    w_limits.xmin = 50;
    w_limits.ymin = 270;
    w_limits.xmax = 210;
    w_limits.ymax = 350;
```

```
    aspect = get_aspect (&w_limits);
    vlptr = &v_limits[0];
    n_vps_on_page = set_vps (aspect, vlptr);
    scale = vp2w_scale (&w_limits, vlptr);
   }
   vlptr2 = vlptr + vp_index;
   draw_ps(hand_list,object_list, vlptr2, &w_limits, scale, fname, fptr);
/*  boxvp_ps (vlptr2, fname, fptr);*/

   vp_index += 1;
   if (vp_index == n_vps_on_page) {
    vp_index = 0;
    fptr = fopen (fname, "a");
    fprintf (fptr, "\nshowpage\n\n");
    fclose (fptr);
   }
}

void draw_ps(p_ptr hand[7], p_ptr obj,Lmtptr vpt,
             Lmtptr wpt,double scale,char fname[10],FILE *fptr)
{
   int    i,j,ixv,iyv;
   double xv,yv,xvmin,yvmin,xwmin,ywmin;
   char   text[80];

   fptr = fopen(fname,"a");

   xvmin = vpt->xmin;
   yvmin = vpt->ymin;
   xwmin = wpt->xmin;
   ywmin = wpt->ymin;

   /*  first draw object */

   fprintf(fptr,"newpath\n");
   for( i=0; i<obj->vert_num; i++)
    {
     xv = xvmin + scale *(obj->vertex[i].x - xwmin);
     yv = yvmin + scale *(obj->vertex[i].y - ywmin);
     ixv = (72. * xv + 0.5);
     iyv = (72. * yv + 0.5);
     if(i==0)
      fprintf(fptr,"    %d %d moveto\n",ixv,iyv);
     else
```

```
      fprintf(fptr,"    %d %d lineto\n",ixv,iyv);

    }
   fprintf(fptr,"closepath\n");
   fprintf(fptr,"0 setlinewidth\n");
   fprintf(fptr,"stroke\n\n");

   /* draw hand */
   for(j=0;j<3;j++)
    {
     fprintf(fptr,"newpath\n");
     for( i=0; i<hand[j]->vert_num; i++)
      {
       xv = xvmin + scale *(hand[j]->vertex[i].x - xwmin);
       yv = yvmin + scale *(hand[j]->vertex[i].y - ywmin);
       ixv = (72. * xv + 0.5);
       iyv = (72. * yv + 0.5);
       if(i==0)
         fprintf(fptr,"    %d %d moveto\n",ixv,iyv);
       else
         fprintf(fptr,"    %d %d lineto\n",ixv,iyv);
      }
     fprintf(fptr,"closepath\n");
     fprintf(fptr,"0 setlinewidth\n");
     fprintf(fptr,"stroke\n\n");
    }
  fclose(fptr);
}
```

These routines, contact.c, supply all the procedures to determine contacts.

```
#include "minclude.h"

/* note- polygon1 and polygon2 are in world reference frames */

double **vertex_to_edge_distances(p_ptr polygon1,p_ptr polygon2,
                                  double **distance)
{    int    n1,n2,i,j;
     double lineA[3],
            dist_line();

     void   m_free();

       n1 = polygon1->vert_num;
       n2 = polygon2->vert_num;

       for(i=0; i<n1; i++)
         for(j=0; j<n2; j++)
           {
            line(polygon2->vertex[j],
              polygon2->vertex[j+1],lineA);
             distance[i][j] =
             dist_line(polygon1->vertex[i],lineA);
           }
       return(distance);
}


void update_contact_information( Hpoly_w, Opoly_w, Hpoly, Opoly, j, i, ccase, ci)
p_ptr   Hpoly_w,
        Opoly_w,
        Hpoly,
        Opoly;
int     i,j,ccase;
ci_ptr  ci;
{  int extra,k;

       extra = 0;
       if(ccase == 1)
        {
           if(ci->v[j].contact==1)
             {
              j= j+Opoly->vert_num;
              extra = 1;
```

```
        }
      ci->v[j].A1          = Hpoly->vertex[i];
      ci->v[j].A2          = Hpoly->vertex[i+1];
      ci->v[j].WA1          = Hpoly_w->vertex[i];
      ci->v[j].WA2          = Hpoly_w->vertex[i+1];
      ci->v[j].contact      = 1;
      ci->v[j].contact_with = Hpoly->name;
      ci->v[j].hvertex      = i;
      if(extra)
       {  k                 = j - Opoly->vert_num;
        ci->v[j].overtex      = k;
        ci->v[j].B           = Opoly->vertex[k];
        ci->v[j].WB           = Opoly_w->vertex[k];
       }
      else
       {
        ci->v[j].overtex      = j;
        ci->v[j].B           = Opoly->vertex[j];
        ci->v[j].WB           = Opoly_w->vertex[j];
       }

  }
 else     /* case == 2 */
  {
     if(ci->e[i].contact==1)
      {
       i = i+Opoly->vert_num;
       extra = 1;
      }
     ci->e[i].B           = Hpoly->vertex[j];
     ci->e[i].WB           = Hpoly_w->vertex[j];
     ci->e[i].contact      = 1;
     ci->e[i].contact_with = Hpoly->name;
     ci->e[i].hvertex      = j;
     if(extra)
      {
       k                  = i - Opoly->vert_num;
       ci->e[i].overtex      = k;
       ci->e[i].A1           = Opoly->vertex[k];
       ci->e[i].A2           = Opoly->vertex[k+1];
       ci->e[i].WA1           = Opoly_w->vertex[k];
       ci->e[i].WA2           = Opoly_w->vertex[k+1];
      }
     else
```

```
            {
             ci->e[i].overtex      = i;
             ci->e[i].A1           = Opoly->vertex[i];
             ci->e[i].A2           = Opoly->vertex[i+1];
             ci->e[i].WA1           = Opoly_w->vertex[i];
             ci->e[i].WA2           = Opoly_w->vertex[i+1];
            }

        } /* end */
        ci->N = ci->N + 1;
}


void contact(p_ptr Hpoly_w,p_ptr Opoly_w,p_ptr Hpoly,
p_ptr Opoly, double epsilon,ci_ptr ci,int s, int *legal)
{
double  **OV_HE,
        **HV_OE,
        a,b,
        **m_create();
int     i,j,HN,ON,hv1,hv2,vcase,equivelant();
void    m_free();

        HN = Hpoly_w->vert_num;
        ON = Opoly_w->vert_num;
        ci->vert_num = ON;
        OV_HE = m_create(OV_HE, ON, HN);
        HV_OE = m_create(HV_OE, HN, ON);

        OV_HE = vertex_to_edge_distances(Opoly_w,Hpoly_w,OV_HE);
        HV_OE = vertex_to_edge_distances(Hpoly_w,Opoly_w,HV_OE);

        for(i=0; i<HN; i++)
          for(j=0; j<ON; j++)
            {
             if(i!=1&&i!=3)
               if( fabs(OV_HE[j][i]) <= epsilon )
                  if( between_pts(Opoly_w->vertex[j],
                Hpoly_w->vertex[i],Hpoly_w->vertex[i+1],
                epsilon))
                     update_contact_information( Hpoly_w, Opoly_w,
                  Hpoly, Opoly,j,i,1,ci);

             if( fabs(HV_OE[i][j]) <= epsilon )
```

```
        if( between_pts(Hpoly_w->vertex[i],
      Opoly_w->vertex[j], Opoly_w->vertex[j+1],
       epsilon))
          update_contact_information( Hpoly_w, Opoly_w,
       Hpoly, Opoly, i,j,2,ci);

    }
/*  are any vertecies penitrating objects */
*legal=1;
for(i=0; i<HN; i++)
  {
   if(*legal)
    {*legal = 0;
     for(j=0; j<ON; j++)
      if((HV_OE[i][j] >= 0 ) ||
         ( HV_OE[i][j] < 0 && fabs(HV_OE[i][j]) <= epsilon))
          *legal = 1;
    }
  }

if(*legal)
for(i=0; i<ON; i++)
  {
   if(*legal)
    {*legal = 0;
     for(j=0; j<HN; j++)
      if((OV_HE[i][j] >= 0 ) ||
         ( OV_HE[i][j] < 0 && fabs(OV_HE[i][j]) <= epsilon))
          *legal = 1;
    }
  }

 /* this is the place for the case when two vertecies are
    in epsilon distance  */

for(i=0;i<ON;i++)
   if(ci->v[i].contact_with==s &&
   ci->v[i+ON].contact_with==s  &&
       ci->v[i].contact==1 && ci->v[i+ON].contact==1)
        if(ci->v[i].hvertex == 3 || ci->v[i].hvertex == 1 )
          {ci->v[i].contact = 0;
           ci->N = ci->N - 1;}
        else if(ci->v[i+ON].hvertex == 3
          ||ci->v[i+ON].hvertex == 1 )
```

```
        {ci->v[i+ON].contact = 0;
         ci->N = ci->N - 1;}
       else if(ci->v[i].hvertex < ci->v[i+ON].hvertex)
        {ci->v[i+ON].contact = 0;
         ci->N = ci->N - 1;}
       else
        {ci->v[i].contact = 0;
         ci->N = ci->N - 1;}


for(i=0;i<ON;i++)
  if(ci->e[i].contact_with==s &&
  ci->e[mod(i+1,ON)].contact_with==s
       && ci->e[i].contact==1 &&
      ci->e[mod(i+1,ON)].contact==1)
    { hv1 = ci->e[i].hvertex;
      hv2 = ci->e[mod(i+1,ON)].hvertex;
      if(hv1==hv2)
        { a = HV_OE[hv1][i];
          b = HV_OE[hv1][mod(i+1,ON)];

          if( a>0 && b<0 ) vcase = 1;
          if( a>0 && b>0 ) vcase = 2;
          if( a<0 && b>0 ) vcase = 3;
          if( a<0 && b<0 ) vcase = 4;

          switch(vcase) {
           case 1:
                ci->e[mod(i+1,ON)].contact = 0;
                ci->N = ci->N - 1;
                break;
           case 2:
                if(abs(a)<=abs(b))
                  ci->e[i].contact = 0;
                else
                  ci->e[mod(i+1,ON)].contact = 0;
                ci->N = ci->N - 1;
                break;
           case 3:
                ci->e[i].contact = 0;
                ci->N = ci->N - 1;
                break;
           case 4:
                if(abs(a)>abs(b))
```

```
                    ci->e[i].contact = 0;
                  else
                    ci->e[mod(i+1,ON)].contact = 0;
                  ci->N = ci->N - 1;
                  break;
                }
              }
            }

        for(i=0;i<ON*2;i++)
          for(j=0;j<ON*2;j++)
            if(ci->v[i].contact==1 && ci->e[j].contact==1 &&
                  ci->v[i].contact_with==s &&
             ci->e[j].contact_with==s)
              if(equivelant(ci->v[i].WB, ci->e[j].WB,
              epsilon+0.2))
                if(OV_HE[i][ci->v[i].hvertex] <
                HV_OE[j][ci->e[j].overtex])
                  {ci->e[j].contact=0;
                   ci->N = ci->N - 1;}
                else
                  {ci->v[i].contact=0;
                   ci->N = ci->N - 1;}


       m_free(OV_HE, ON);
       m_free(HV_OE, HN);
}

void copy_fn(ni fn,ni oldfn)
{  int i,j;

   for(i=0;i<7;i++)
    {
     oldfn->n_ind[i] = fn->n_ind[i];
     oldfn->c[i] = fn->c[i];
    for(j=0;j<5;j++)
     oldfn->n[i][j] = fn->n[i][j];
    for(j=0;j<7;j++)
     {oldfn->q[i][j] = fn->q[i][j];
      oldfn->q_ind[i][j] = fn->q_ind[i][j];}
    }
}
```

```
ni
 force_normals(ci_ptr ci, int L[3], p_ptr obj, ni fn, point obase)
{
  int   s,j,a,b;
  ni    make_normal();
  void  choose_ab();

  for(s=0; s<3; s++)
  {
   choose_ab(s,L[s],&a,&b,ci);
   if(L[s]>0)
      for(j=0; j<obj->vert_num; j++)
        {
         if(ci->v[j+obj->vert_num].contact>=1 &&
                      ci->v[j+obj->vert_num].contact_with==s)
          { fn = make_normal(ci, j+obj->vert_num, a, fn,
                /*case*/ 1);
            ci->v[j+obj->vert_num].contact_number = a;
             a = b;
          }
         if(ci->v[j].contact>=1 && ci->v[j].contact_with==s)
          { fn = make_normal(ci, j, a, fn,/*case*/ 1);
            ci->v[j].contact_number = a;
             a = b;
          }
         if(ci->e[j].contact>=1 && ci->e[j].contact_with==s)
          { fn = make_normal(ci, j, a, fn,/*case*/ 2);
            ci->e[j].contact_number = a;
             a = b;
          }
        }
  }
  fn->n[0][0] = 0;
  fn->n[0][1] = -1;
  fn->n[0][2] = 1;
  fn->n[0][3] = 0;
  fn->n[0][4] =-obase.x;
  fn->c[0] = obase;
  fn->n_ind[0] = 1;
  return(fn);
}

ni
 make_normal(ci_ptr ci, int j, int a,ni fn, int ncase)
```

```
{
  int i;
  double line1[3],line2[5],x;
  point m;

  if(ncase==1)
   {line(ci->v[j].WA1,ci->v[j].WA2,line1);
    fn->c[a] = ci->v[j].WB;}
  if(ncase==2)
   {line(ci->e[j].WA2,ci->e[j].WA1,line1);
    fn->c[a] = ci->e[j].WB;}

    m.x = 1000*line1[0] + fn->c[a].x;
    m.y = 1000*line1[1] + fn->c[a].y;
  line(fn->c[a],m,line2);

  for(i=0;i<2;i++)
    fn->n[a][i] = line1[i];
  for(i=2;i<5;i++)
    fn->n[a][i] = line2[i-2];
  fn->n_ind[a] = 1;
  return(fn);
}

void
 choose_ab(int i,int n,int *a,int *b, ci_ptr ci)
{
     if(n==0)
      { *a=0; *b=0;}
     else if(n==1)          /* n is the # of contacts */
      {                   /* i is the # of the polygon*/
       if(i==0)
         { *a=3; *b=0;}
       else if(i==1)
         { *a=1; *b=0;}
       else if(i==2)
         { *a=2; *b=0;}
      }
     else if(n==2)
      {
       if(i==0)
         if(ci->v[0].contact==1&&ci->v[0].contact_with==i)
          { if((ci->v[1].contact==1&&ci->v[1].contact_with==i)
                        || ci->e[0].contact==1)
```

```
            { *a=3; *b=4;}
           else
            { *a=4; *b=3;}
         }
        else
         { *a=3; *b=4;}
      else if(i==1)
        if(ci->v[0].contact==1&&ci->v[0].contact_with==i)
         { if((ci->v[1].contact==1&&ci->v[1].contact_with==i)
                    || ci->e[0].contact==1)
           { *a=1; *b=5;}
           else
            { *a=5; *b=1;}
         }
        else
         { *a=1; *b=5;}
      else if(i==2)
        if(ci->v[0].contact==1&&ci->v[0].contact_with==i)
         { if((ci->v[1].contact==1&&ci->v[1].contact_with==i)
                    || ci->e[0].contact==1)
           { *a=6; *b=2;}
           else        /* 6 is first because the contacts are */
            { *a=2; *b=6;}/* found counter clockwise.          */
         }
        else
         { *a=6; *b=2;}


    }
}


void find_intersections(ni fn)
{
 int     i,j,s,k;
 double  a[2][3];
 void    gauss();

 s=0;
 for(i=0;i<7;i++)
   for(j=i+1;j<7;j++)
     fn->q_ind[i][j] = fn->n_ind[i] * fn->n_ind[j];

 for(i=0;i<7;i++)
```

```
    for(j=i+1;j<7;j++)
      if(fn->q_ind[i][j] == 1)
        {
         for(k=0;k<3;k++)
           {
            a[0][k] = fn->n[i][k+2];
            a[1][k] = fn->n[j][k+2];
           }
         gauss(2,1,a,1.0e-6,&s);
         if(s==1)
           fn->q_ind[i][j] = 0;
         if(s==0)
           {
            fn->q[i][j].x = -a[0][2];
            fn->q[i][j].y = -a[1][2];

            /* The next few lines admit that there is a
           solution,but that it is off the screen! */
            if(fn->q[i][j].x<=0 || fn->q[i][j].y<=0)
                fn->q_ind[i][j] = 2;
            if(fn->q[i][j].x>250 || fn->q[i][j].y>400)
                fn->q_ind[i][j] = 2;

           }
         s=0;
        }
}

void form_closure(double theta[2][2], ni fn, int save[3][4], int *p)
{
  int   m,n,i,j,k,l,e[3];
  double one,two;
  void choose_ijnm();

  for(i=0;i<3;i++)
   for(j=0;j<4;j++)
     save[i][j]=0;
  *p = 0;
  if((theta[0][0] - theta[1][0])<0)
      return;

  for(k=0;k<3;k++) e[k] = 0;
  if( fn->n_ind[3]==1 && fn->n_ind[4]==1 ) e[0] = 1;
  if( fn->n_ind[1]==1 && fn->n_ind[5]==1 ) e[1] = 1;
```

```
  if( fn->n_ind[2]==1 && fn->n_ind[6]==1 ) e[2] = 1;

  for(k=0;k<3;k++)
   if(e[k])
     for(l=0;l<4;l++) /*  i,j are the forces to intersect */
      {              /*  n,m are the edge verteces */
        choose_ijnm(k,l,&i,&j,&n,&m);
        if(fn->q_ind[i][j]==1 || fn->q_ind[i][j]==2)
         {
          one = fn->n[n][2]*fn->q[i][j].x +
              fn->n[n][3]*fn->q[i][j].y + fn->n[n][4];
          two = fn->n[m][2]*fn->q[i][j].x +
              fn->n[m][3]*fn->q[i][j].y + fn->n[m][4];
          if(one*two<0)
           { save[*p][0] = n;
             save[*p][1] = m;
             save[*p][2] = i;
             save[*p][3] = j;
               *p=*p+1;
             break;
           }
         }
      }
}

void choose_ijnm(int k,int l,int *i,int *j,int *n,int *m)
{
   if(k==0)
     {*n=3;*m=4;
      if(l==0)
       {*i=1;*j=2;}
      else if(l==1)
       {*i=1;*j=6;}
      else if(l==2)
       {*i=2;*j=5;}
      else if(l==3)
       {*i=5;*j=6;}
     }
   else if(k==1)
     {*n=1;*m=5;
      if(l==0)
       {*i=2;*j=3;}
      else if(l==1)
       {*i=2;*j=4;}
```

```
      else if(l==2)
       {*i=3;*j=6;}
      else if(l==3)
       {*i=4;*j=6;}
     }
   else
     {*n=2;*m=6;
      if(l==0)
       {*i=1;*j=3;}
      else if(l==1)
       {*i=1;*j=4;}
      else if(l==2)
       {*i=3;*j=5;}
      else if(l==3)
       {*i=4;*j=5;}
     }
}

int in_form_closure(double theta[2][2], ni fn, int fc[4])
{
  int   m,n,i,j;
  double one,two;

  if((theta[0][0] - theta[1][0])<0)
      return(0);

    /*  i,j are the forces to intersect */
    /*  n,m are the edge verteces */
       n=fc[0];
       m=fc[1];
       i=fc[2];
       j=fc[3];
         one = fn->n[n][2]*fn->q[i][j].x +
             fn->n[n][3]*fn->q[i][j].y + fn->n[n][4];
         two = fn->n[m][2]*fn->q[i][j].x +
           fn->n[m][3]*fn->q[i][j].y + fn->n[m][4];
         if(one*two<=0)
           return(1);
         else
           return(0);
}
int good_cf(cf cf[3])
{  int i;
   for(i=0;i<3;i++)
```

```
    if(cf[i].type == -1)
        return(0);
   return 1;
}


void get_cf(ci_ptr ci, cf c[3])
{
 int  L[3],i,j,j1,k,mod(),n,inc1,inc2;
 void getL();

    getL(ci,L);

    for(i=0;i<3;i++)
     {
       c[i].contact_with = i;

       if(L[i]==1)
        {
          for(j=0;j<ci->vert_num*2;j++)
           {
            if(ci->v[j].contact==1 && ci->v[j].contact_with==i)
             {c[i].number = j;
              c[i].type = 0;}
            if(ci->e[j].contact==1 && ci->e[j].contact_with==i)
             {c[i].number = j;
              c[i].type = 3;}
           }
        }
       else if(L[i]==2)
        { c[i].type = 1;
          for(k=0;k<4;k++)
           { n = ci->vert_num;
            if(k==0)
              {inc1 = 0; inc2 = 0;}
            if(k==1)
              {inc1 = n; inc2 = 0;}
            if(k==2)
              {inc1 = 0; inc2 = n;}
            if(k==3)
              {inc1 = n; inc2 = n;}

            for(j=0;j<ci->vert_num;j++)
             {
```

```
                j1 = mod(j+1,ci->vert_num);
                if( ci->v[j+inc1].contact==1 &&
                ci->v[j1+inc2].contact==1 &&
                        ci->v[j+inc1].contact_with==i &&
                ci->v[j1+inc2].contact_with==i)
                    c[i].number=j;

                if( ci->v[j+inc1].contact==1 &&
                ci->e[j+inc2].contact==1 &&
                        ci->v[j+inc1].contact_with==i &&
                ci->e[j+inc2].contact_with==i)
                    c[i].number=j;

                if( ci->e[j+inc1].contact==1 &&
                ci->v[j1+inc2].contact==1 &&
                        ci->e[j+inc1].contact_with==i &&
                ci->v[j1+inc2].contact_with==i)
                    c[i].number=j;
              }
            }
          }
        else
          { c[i].type = -1;
            c[i].number = -1;
            c[i].contact_with = -1;
          }
      }
}

int test_for_samecf(cf oldcf[3],cf cf[3])
{   int i;

    for(i=0;i<3;i++)
    { if(oldcf[i].type!=cf[i].type)
            return(0);
      if(oldcf[i].number!=cf[i].number)
            return(0);
      if(oldcf[i].contact_with!=cf[i].contact_with)
            return(0);
    }
    return(1);
}

int same_cf_numbers(cf c1[3], cf c2[3])
```

```
{
 int i;

 for(i=0;i<3;i++)
  {
  if(c1[i].type!=0)
   { if(c1[i].type!=c2[i].type)
        return(0);
     if(c1[i].number!=c2[i].number)
        return(0);
     if(c1[i].contact_with!=c2[i].contact_with)
        return(0);
   }
  else if(c2[i].type!=0)
    return 0;
 }
 return(1);
}


void copy_cf(cf cf[3],cf oldcf[3])
{ int i;
   for(i=0;i<3;i++)
    { oldcf[i].type=cf[i].type;
      oldcf[i].number=cf[i].number;
      oldcf[i].contact_with=cf[i].contact_with;
    }
}

void get_formc_parameters(int *p1, int *p2, int c[3], int fc[4], int n, cf cf[3], int
*inc1,int *inc2,int *dec1,int *dec2, ni fn)
{    int done;

     c[0] = fc[0];
     c[1] = fc[1];
     *p1 = 0; *p2 = 0;
     *inc1=0; *inc2=0; *dec1=0; *dec2=0;
     done = 0;

     if(c[0]==3)
      { if(n==0)
         {
           *p2=1;
           if(cf[1].type==0 || cf[1].type==3)
```

```
        c[2]=1;
       if(cf[1].type==1)
        c[2]=5;
       *inc1=1;

    }
   else if(n==1)
    {
      *p1=1;
      if(cf[2].type==0 || cf[2].type==3 )
       c[2]=2;
      if(cf[2].type==1)
       c[2]=6;
      *dec2=1;

    }
 }
else if(c[0]==1)
 {
     if(n==0)
      { *p2=1;
        if(cf[0].type==0)
           c[2]=3;
        if(cf[0].type==1)
           c[2]=4;
        *inc1=1;
       }
     else if(n==1)
      { *p2=1;
        c[2]=3;
       *dec1=1;
      }
 }
else if(c[0]==2)
 {
    if(n==0)
     { *p1=1; c[2]=3; *inc2=1;}
    else if(n==1)
     { *p1=1;
       if(cf[0].type==0)
         c[2]=3;
       if(cf[0].type==1)
         c[2]=4;
       *dec2=1;
```

```
        }
      }

}

int number_of_edge_contacts(cf c[3])
{
  int i,j;
  j=0;
  for(i=0;i<3;i++)
    if(c[i].type==1)
      j++;
  return(j);
}

void get_edge_parameters(int *p1,int *p2,int c[3],int n5,int n,cf cf[3], int *inc1,int
*inc2,int *dec1,int *dec2)
{     int  i,j,k,done;

      j=0;
      for(i=0;i<3;i++)
        if(cf[i].type == 1)
          {j++;
           if(j==n5+1) k=i;
          }
      if(k==0)
       {c[0]=3; c[1]=4;}
      else if(k==1)
       {c[0]=1; c[1]=5;}
      else if(k==2)
       {c[0]=2; c[1]=6;}

      *p1 = 0; *p2 = 0;
      *inc1=0; *inc2=0; *dec1=0; *dec2=0;
      done = 0;

      if(c[0]==3)
       { if(n==0)
          {
            *p2=1;
            if(cf[1].type==0 || cf[1].type==3 )
             c[2]=1;
            if(cf[1].type==1)
             c[2]=5;
```

```
        *inc1=1;

      }
     else if(n==1)
      {
        *p1=1;
        if(cf[2].type==0 || cf[2].type==3 )
         c[2]=2;
        if(cf[2].type==1)
         c[2]=6;
        *dec2=1;

      }
  }
 else if(c[0]==1)
  {
      if(n==0)
       { *p2=1;
         if(cf[0].type==0)
            c[2]=3;
         if(cf[0].type==1)
            c[2]=4;
         *inc1=1;
        }
      else if(n==1)
       { *p2=1;
            c[2]=3;
         *dec1=1;
       }

  }
 else if(c[0]==2)
  {
     if(n==0)
      { *p1=1; c[2]=3; *inc2=1;}
     else if(n==1)
      { *p1=1;
        if(cf[0].type==0)
          c[2]=3;
        if(cf[0].type==1)
          c[2]=4;
        *dec2=1;
      }
```

```
      }

}

/*************** tree routines ******************/

int  nodes_equal(cfnode a, cfnode b)
{
 int   i;

 for(i=0;i<3;i++)
  {
   if(a->cf[i].type != b->cf[i].type) return 0;
   if(a->cf[i].number != b->cf[i].number) return 0;
   if(a->cf[i].contact_with != b->cf[i].contact_with)
                return 0;
  }
  return 1;
}
int new_nodes_equal(cfnode a,cf cf[3])
{
 int   i;

 for(i=0;i<3;i++)
  {
   if(a->cf[i].type != cf[i].type) return 0;
   if(a->cf[i].number != cf[i].number) return 0;
   if(a->cf[i].contact_with != cf[i].contact_with) return 0;
  }
  return 1;
}

cfnode talloc(void)
{ cfnode p;
 p=(cfnode)malloc(sizeof(struct cfnode));
 if(!p) nrerror("allocation failure in talloc()");
 return p;
}

cfnode addnode(cf cf[3], point obase, double otheta, double thetaf[2][2],
               cfnode temp, int row, int node_number,
               double thetafdot[2][2],int c[3],int tree,int stype)
{  int i,j;
   cfnode  p;
```

```
    p = talloc();
    for(i=0; i<3; i++)
     p->cf[i] = cf[i];
    p->obase = obase;
    p->otheta = otheta;
    for(i=0; i<2; i++)
     for(j=0; j<2; j++)
       {
        p->thetaf[i][j] = thetaf[i][j];
        p->thetafdot[i][j] = thetafdot[i][j];
       }
    for(j=0; j<3; j++)
      p->c[j] = c[j];
    p->tree = tree;
    p->stype = stype;
    p->back = temp;
    p->next = NULL;
    p->path = 0;
    p->end = 0;
    p->row = row;
    p->node_number = node_number;

    return p;
}

void print_path(cfnode path)
{ char text[80];
  sprintf(text,"%s %d\n"," node ",path->node_number);
  _outtext(text);
  sprintf(text,"%d %d %d\n",path->c[0],path->c[1],path->c[2]);
  _outtext(text);
  sprintf(text,"%d %f %d\n", path->stype, path->otheta,
                  path->tree);
  _outtext(text);
  sprintf(text,"%f %f %s\n", path->thetaf[0][0],
                  path->thetaf[1][0], "end");
  _outtext(text);
  sprintf(text,"%s %f %f\n", "tdot", path->thetafdot[0][0],
                  path->thetafdot[1][0]);
  _outtext(text);

}
```

```
void connect_tree_parameters(cfnode path, double thetaf[2][2],int c[3], int
*edge,double thetafdot[2][2],int *placef1,int *placef2, int *inc1,int *inc2,int *dec1,int
*dec2)
{ int i,j;
  double diff1, diff2;

  *edge = number_of_edge_contacts(path->cf);
  *placef1=0;  *placef2=0;
  *inc1=0; *inc2=0; *dec1=0; *dec2=0;

  if(*edge)
    if(path->cf[0].type == 1)
      { c[0] = 3; c[1] = 4; c[2] = 1;
        *placef2 = 1;
        if(thetaf[0][0] < path->thetaf[0][0])
          *inc1 = 1;
        else
          *dec1 = 1;
      }
    else if(path->cf[1].type == 1)
      { c[0] = 1; c[1] = 5;  c[2] = 3;
        *placef2 = 1;
        if(thetaf[0][0] < path->thetaf[0][0])
          *inc1 = 1;
        else
          *dec1 = 1;
      }
    else if(path->cf[2].type == 1)
      { c[0] = 2; c[1] = 6;  c[2] = 3;
        *placef1 = 1;
        if(thetaf[1][0] < path->thetaf[1][0])
          *inc2 = 1;
        else
          *dec2 = 1;
      }
  else /* no edge contact */
    {
      c[0] = 1; c[1] = 2; c[2] = 3;
      diff1 = fabs(thetaf[0][0] - path->thetaf[0][0]);
      diff2 = fabs(thetaf[1][0] - path->thetaf[1][0]);
      if(diff1 > diff2)
        {
         thetafdot[1][0] = 1;
         thetafdot[0][0] = diff1/diff2;
```

```
        }
        else
        {
         thetafdot[0][0] = 1;
         thetafdot[1][0] = diff2/diff1;
        }
        if(thetaf[0][0] > path->thetaf[0][0])
          thetafdot[0][0] = -thetafdot[0][0];
        if(thetaf[1][0] > path->thetaf[1][0]);
          thetafdot[1][0] = -thetafdot[1][0];
    }
}
```

```
#include "minclude.h"
/************ MISCELANEOUS ROUTINES ********************/

char
*strsave(char *line)
{
   char *p;

   if((p=(char *)calloc(strlen(line)+1, sizeof(char)))!=NULL)
      strcpy(p, line);
   return(p);
}


get_vertex_count(int *pverteces,int **fverteces,
      int  *overteces, char *palm_name,
      char ***link_names,char *object_name,
         int num_fingers, int *num_finger_links)
{

   int       i,
             j,
             temp;
   FILE      *f_ptr;

   f_ptr = fopen(palm_name, "r");
   fscanf(f_ptr, "%d", &temp);
   *pverteces = temp;
   fclose(f_ptr);
   for (i = 0; i < num_fingers; i++)
      for (j = 0; j < num_finger_links[i]; j++)
      {
         f_ptr = fopen(link_names[i][j], "r");
         fscanf(f_ptr, "%d", &temp);
         fverteces[i][j] = temp;
         fclose(f_ptr);
      }

      f_ptr = fopen(object_name, "r");
      fscanf(f_ptr, "%d", &temp);
      *overteces = temp;
      fclose(f_ptr);
```

```
}

get_polygon_coordinates(int pverteces, int **fverteces, int overteces, char
*palm_name,char ***link_names,
char *object_name, int num_fingers, int *num_finger_links, h_ptr h1,p_ptr object)
{
   int       i,
             j,
             k,
             blah;
   float     temp;
   FILE       *f_ptr;

   f_ptr = fopen(palm_name, "r");
   fscanf(f_ptr, "%d", &blah);
   for (i = 0; i < = pverteces; i++)
   {
      fscanf(f_ptr, "%f", &temp);
      h1->palm->vertex[i].x = (double)temp;
      fscanf(f_ptr, "%f", &temp);
       h1->palm->vertex[i].y = (double)temp;
   }
   fclose(f_ptr);
   for (i = 0; i < num_fingers; i++)
      for (j = 0; j < num_finger_links[i]; j++)
      {
         f_ptr = fopen(link_names[i][j], "r");
         fscanf(f_ptr, "%d", &blah);
         for (k = 0; k < = fverteces[i][j]; k++)
         {
            fscanf(f_ptr, "%f", &temp);
            h1->finger[i]->link[j]->vertex[k].x =
                  (double)temp;
            fscanf(f_ptr, "%f", &temp);
            h1->finger[i]->link[j]->vertex[k].y =
                  (double)temp;
         }
         fclose(f_ptr);
      }

      f_ptr = fopen(object_name, "r");
      fscanf(f_ptr, "%d", &blah);
      for (j = 0; j < = overteces; j++)
      {
```

```
            fscanf(f_ptr, "%f", &temp);
             object->vertex[j].x = (double)temp;
            fscanf(f_ptr, "%f", &temp);
             object->vertex[j].y = (double)temp;
        }
        fclose(f_ptr);


}

void move_all(h1,h2,thetaf,thetap,palm_base,A,B,D,T_frame,
num_fingers, num_finger_links, hand_list, ci, object, b0, c, obase, otheta, tobject,
object_list, don)
int     num_fingers,num_finger_links[2],c[3],*don;
h_ptr   h1,h2;
double  thetaf[2][2],thetap,*otheta;
point   palm_base,*obase;
rt_ptr  **A,**B,**D,**T_frame;
p_ptr   hand_list[7],object,tobject,object_list;
ci_ptr  ci;
b_ptr   b0;
{  double  obj_info[3];
   int     done,TI;

          done = *don;
          move_hand(h1, h2, thetaf, thetap, palm_base, A, B, D,
                T_frame);

          get_hand_list(h2, num_fingers, num_finger_links,
                hand_list);

          update_new_hand_eqns_in_ci(hand_list,ci,object);

          if ( ci->N >=3 )
            { obj_info[0] = (*obase).x;
              obj_info[1] = (*obase).y;
              obj_info[2] = *otheta;

              solve_position(obj_info,ci,TI,b0,c);

              (*obase).x = obj_info[0] ;
              (*obase).y = obj_info[1] ;
              *otheta  = obj_info[2] ;
             }
```

```
        if( ((*obase).x <70 || (*obase).x >190) ||
          ((*obase).y <280 || (*obase).y>380) )
           done = 1;

        move_object(object, tobject, (*obase).x, (*obase).y,
                    *otheta);
        object_list = tobject;
        if(!done)
        {
         solve_move_finger(thetaf, h1, h2, palm_base, tobject,
        thetap,A,B,D,T_frame, hand_list, 1, &done);

         solve_move_finger(thetaf, h1, h2, palm_base, tobject,
        thetap,A,B,D,T_frame, hand_list, 2, &done);
        }
        *don = done;

}

void contact_normals(ci,object,hand_list,object_list,h1,tol,leg,
                     L,fn, obase)
ci_ptr  ci;
p_ptr   object,hand_list[7],object_list;
h_ptr   h1;
double  tol;
int     *leg,L[3];
ni      fn;
point   obase;
{  int      legal,i;
   ci_ptr   initialize_ci();
   ni       initialize_fn(),force_normals();
        legal = *leg;

        ci = initialize_ci(object, ci);
        legal = 1;
        for(i=1; i<3; i++)
         if(legal)
          contact(hand_list[i], object_list,
              h1->finger[i-1]->link[0],
                    object, tol, ci, i,&legal);
        if(legal)
          contact(hand_list[0], object_list, h1->palm, object,          tol, ci,0,&legal);
        *leg = legal;
```

```
        getL(ci,L);

        fn = initialize_fn(fn);

        fn = force_normals(ci, L, object, fn, obase);

        find_intersections(fn);
}


void move_all_get_contacts(object,obase,otheta,h1,h2,thetaf,
 thetap,palm_base,A,B,D,T_frame,num_fingers,num_finger_links,
                hand_list,object_list,tobject,ci,tol,leg,L,fn)
p_ptr object,hand_list[7],object_list,tobject;
ci_ptr ci;
point obase,palm_base;
double otheta,thetap,thetaf[2][2],tol;
h_ptr h1,h2;
rt_ptr **A,**B,**D,**T_frame;
int num_fingers,num_finger_links[2],*leg,L[3];
ni fn;
{
    int legal,done;
     legal = *leg;
 move_object(object, tobject, obase.x, obase.y, otheta);
move_hand(h1, h2, thetaf, thetap, palm_base, A, B, D,T_frame);
          get_hand_list(h2,num_fingers,num_finger_links,hand_list);
          object_list = tobject;
          solve_move_finger(thetaf, h1, h2, palm_base, tobject,
              thetap,A,B,D,T_frame, hand_list, 1, &done);

          solve_move_finger(thetaf, h1, h2, palm_base, tobject,
              thetap,A,B,D,T_frame, hand_list, 2, &done);

        contact_normals(ci,object,hand_list,object_list,h1,tol,
                            &legal,L,fn, obase);
    *leg = legal;
}

/*********************************************************/
```

```
#include "minclude.h"
#include "nr.h"
#include "nrutil.h"
# define PI 3.141559265359
# define FW 3.0
#define LENGTH 60
#define OVN 4      /* obj->vert_num - 1*/

update_new_hand_eqns_in_ci(p_ptr hand_list[7],ci_ptr ci,
p_ptr obj)
{  int i,cw,hv;

   for(i=0; i<obj->vert_num*2; i++)
   {
    if(ci->v[i].contact == 1)
     {
      cw = ci->v[i].contact_with;
      hv = ci->v[i].hvertex;
      ci->v[i].WA1 = hand_list[cw]->vertex[hv];
      ci->v[i].WA2 = hand_list[cw]->vertex[hv+1];
     }

    if(ci->e[i].contact == 1)
     {
      cw = ci->e[i].contact_with;
      hv = ci->e[i].hvertex;
      ci->e[i].WB = hand_list[cw]->vertex[hv];
     }
   }
}

int abs(int x)
{ if(x<0) x = -x;
  return(x);
}

int mod(int x,int y)
/* ( x mod y ) */
{  int i,f;
   if(x<0)
     f = -100;
   else
     f = 100;
   i=0;
```

```
   while(abs(f)>=abs(y) && abs(f+x)==(abs(f)+abs(x)))
   { if(x<0)
        f = x + i*y;
     else
        f = x - i*y;
     i++;
   }
   if(f<0) f = f + y;
   return(f);
}

void getL(ci_ptr ci,int L[3])
{ int i,k;

   for(k=0; k<3; k++)
        L[k]=0;
   for(k=0; k<3; k++)
    for(i=0; i<ci->vert_num*2; i++)
     {
       if(ci->v[i].contact_with == k && ci->v[i].contact==1)
          L[k]++;
       if(ci->e[i].contact_with == k && ci->e[i].contact==1)
          L[k]++;
     }
   for(k=0; k<3; k++)
    if(L[k]>2)
       L[k]=2;


}

void get_b0(int c,int theta_increasing,ci_ptr ci,b_ptr b0,
int two, int k,int icase[3])
/*    c is the polygon # */
{
int     test,i;

   test = 1;
   i = 0;
   while(test)
   {
    if(ci->v[i].contact ==1 && ci->v[i].contact_number == c)
     {
          /* case 1 */
```

```
        if(two &&theta_increasing && ci->v[i+1].contact_with==c)
           i++;
        b0->xk[k] = ci->v[i].WA1.x;
        b0->yk[k] = ci->v[i].WA1.y;
        b0->xk1[k] = ci->v[i].WA2.x;
        b0->yk1[k] = ci->v[i].WA2.y;
        b0->xj[k] = ci->v[i].B.x;
        b0->yj[k] = ci->v[i].B.y;
        test = 0;
        icase[k] = 1;
        ci->v[i].contact = 2;

     }
     else if(ci->e[i].contact==1&&ci->e[i].contact_number==c)
     {
          /* case 2 */

        b0->xk[k] = ci->e[i].A1.x;
        b0->yk[k] = ci->e[i].A1.y;
        b0->xk1[k] = ci->e[i].A2.x;
        b0->yk1[k] = ci->e[i].A2.y;
        b0->xj[k] = ci->e[i].WB.x;
        b0->yj[k] = ci->e[i].WB.y;
        test = 0;
        icase[k] = 2;
        ci->e[i].contact = 2;

     }


      i++;
   } /* end while */
}

void assign_b0_verticies(ci_ptr ci,int TI,b_ptr b0,int icase[3],int c[3])
{
 int    i,k,num;

 /* choose the three constraint equations */

        get_b0(c[0],TI,ci,b0,0,0,icase);
        get_b0(c[1],TI,ci,b0,0,1,icase);
        get_b0(c[2],TI,ci,b0,0,2,icase);
```

```
    for(i=0; i<ci->vert_num*2; i++)
       /* i must be atleast as large as the obj->vert_num*/
    {
      if(ci->e[i].contact == 2)
        ci->e[i].contact = 1;
      if(ci->v[i].contact ==2)
        ci->v[i].contact = 1;
    }

}


void
constraint(double *x,double **alpha,double *bet,ci_ptr ci,
           int TI,b_ptr b0,int co[3])
  {  double  c3,s3,pa1,pa2,pa3,pb1,pb2,pb3,pc1,pc2,pc3,
           pxk3,pxk13,pyk3,pyk13,xk[3], yk[3], xj[3],          yj[3], xk1[3],
           yk1[3], a[3], b[3], c[3];
       int     icase[3],i;

       assign_b0_verticies(ci,TI,b0,icase,co);

       c3 = cos(x[3]);
       s3 = sin(x[3]);

       for(i=0; i<3; i++)
          if(icase[i] == 1)
          {
              xj[i] = b0->xj[i] * c3 - b0->yj[i] * s3 + x[1];
              yj[i] = b0->xj[i] * s3 + b0->yj[i] * c3 + x[2];
              a[i] = b0->yk1[i] - b0->yk[i];
              b[i] = b0->xk[i] - b0->xk1[i];
              c[i] = b0->xk1[i] * b0->yk[i] - b0->xk[i] *
                b0->yk1[i];
              bet[i+1] = a[i] * xj[i] + b[i] * yj[i] + c[i];
              bet[i+1] = -bet[i+1];

              alpha[i+1][1] = a[i];
              alpha[i+1][2] = b[i];
              alpha[i+1][3] = a[i] * (-b0->xj[i] * s3 - b0->yj[i]
            *c3) + b[i] * ( b0->xj[i] * c3 - b0->yj[i] *s3);
          }
          else    /* icase[i] = 2 */
          {
```

```
        xj[i] = b0->xj[i];
        yj[i] = b0->yj[i];
        xk[i] = b0->xk[i] * c3 - b0->yk[i] * s3 + x[1];
        yk[i] = b0->xk[i] * s3 + b0->yk[i] * c3 + x[2];
        xk1[i]= b0->xk1[i] * c3 - b0->yk1[i] * s3 + x[1];
        yk1[i]= b0->xk1[i] * s3 + b0->yk1[i] * c3 + x[2];

      a[i] = yk1[i] - yk[i];
      b[i] = xk[i] - xk1[i];
      c[i] = xk1[i] * yk[i] - xk[i] * yk1[i];

        bet[i+1] = a[i] * xj[i] + b[i] * yj[i] + c[i];
        bet[i+1] = -bet[i+1];

      /*  partials for jacobian */
      /*  pa1 is partial of a w.r.t. x[1]  */
      /*  pyk13 is partial yk1 w.r.t. x[3] */

      pa1 = 0;
      pa2 = 0;
        pa3 = c3 * (b0->xk1[i] - b0->xk[i])
              - s3 * (b0->yk1[i] - b0->yk[i]);

      pb1 = 0;
      pb2 = 0;
        pb3 =-s3 * (b0->xk[i] - b0->xk1[i])
             - c3 * (b0->yk[i] - b0->yk1[i]);

      pc1 = -a[i];
      pc2 = -b[i];
        pxk3= -s3 * b0->xk[i] - c3 * b0->yk[i];
        pxk13= -s3 * b0->xk1[i] - c3 * b0->yk1[i];
        pyk3= c3 * b0->xk[i] - s3 * b0->yk[i];
        pyk13= c3 * b0->xk1[i] - s3 * b0->yk1[i];
        pc3 = pxk13 * yk[i] + xk1[i] * pyk3
              - pxk3 * yk1[i] - xk[i] * pyk13;

        alpha[i+1][1] = pa1 * xj[i] + pb1 * yj[i] + pc1;
        alpha[i+1][2] = pa2 * xj[i] + pb2 * yj[i] + pc2;
        alpha[i+1][3] = pa3 * xj[i] + pb3 * yj[i] + pc3;
    }

}
```

```
#define NTRIAL 10
#define TOLX 1.0e-6
#define TOLF 1.0e-6
#define N 3

void
   solve_position(double obj_info[4],ci_ptr ci,int TI,b_ptr b0,int c[3])
      {
          double  *x,
                  **alpha,
                  *bet;
         int     i,j,k;

          alpha = dmatrix(1,N,1,N);
          bet = dvector(1,N);
          x = dvector(1,N);

          x[1] = obj_info[0];          /*base.x;*/
          x[2] = obj_info[1];          /*base.y;*/
          x[3] = obj_info[2]*PI/180;   /*theta;*/

          mnewt(NTRIAL,x,3,TOLX,TOLF,ci,TI,b0,c);
          constraint(x,alpha,bet,ci,TI,b0,c);

          obj_info[0] = x[1];
          obj_info[1] = x[2];
          obj_info[2] = x[3]*180/PI;

          free_dmatrix(alpha,1,N,1,N);
          free_dvector(bet,1,N);
          free_dvector(x,1,N);
      }

double check(double x)
{  char txt[40];
  if( x > 1 || x < -1)
    {
/*    sprintf(txt,"%s %f\n"," x is ",x);
    _outtext(txt);*/
    x = -1;
   /* getch();*/
   }
  return x;
}
```

```
void get_angle(point p1, point p0, int finger, double *l, double *theta)
{
 double   fw,d,dist(),chi,alpha,check();

    fw =(double) FW;
    d = dist(p1,p0);
    chi = asin(check((p1.y-p0.y)/d));
    alpha = asin(check(fw/d));
    *l = d*cos(alpha);


    if(finger==1)
      if(p1.x>p0.x)
       *theta = (chi-alpha)*180/PI;
      else
       *theta = 180-((chi+alpha)*180/PI);
    else if(finger==2)
      if(p1.x>p0.x)
       *theta = (chi+alpha)*180/PI;
      else
       *theta = 180 - (chi-alpha)*180/PI;
}


double solve_finger_angle(h_ptr h1,point palm_base,p_ptr object,int finger, int *done)
 {
   int          i,j,not_done,large,large1,inc,mod();
   double       il,const1,Pd,check(),fw,
                theta[10],length[10],temp,thetaf,
                dist(),A,B,C,m,b,c,d,r;
   point        p0,p1,M,pt1,pt2;

   il = LENGTH;         /* il is the inner finger length */
   const1 = 5.0;     /* const1 is the extra length from the
                        joint to the end*/
   if(finger==1)  /* Pd is the length of the perpendicular*/
     { Pd = (double) -FW; j=0; inc=1; }
   else
     { Pd = (double) FW; j=1; inc=-1;}

   /*theta = dvector(0,OVN);
   length = dvector(0,OVN);*/
```

```
p0.x = palm_base.x + h1->finger[j]->b_i[0];
p0.y = palm_base.y;

for(i=0; i<object->vert_num; i++)
  {
   get_angle(object->vertex[i], p0, finger, &length[i],
             &theta[i]);
  }

if(finger == 1)
  {
   temp = 200;
   for(i=0; i<object->vert_num; i++)
      {
       if(theta[i]<=temp)
        { temp = theta[i];
          large = i;
        }
       }
  }
if(finger == 2)
  {
   temp = -100;
   for(i=0; i<object->vert_num; i++)
      {
      if(theta[i]>=temp)
        { temp = theta[i];
          large = i;
        }
      }
   }
*done = 1;
for(i=0;i<object->vert_num; i++)
 {
  if(length[i]+const1<il-1)
    {*done = 0;
     break;}
 }
if(!*done)
{
 if(( length[large]+const1)<=il)
  {
   return(theta[large]);
  }
```

```
else if(finger==2)
       large++;
      else if(finger==1)
       large--;

large = mod(large,object->vert_num);

not_done = 1;

while(not_done)
{

 if(( length[large]+const1)<=il)
   {
    large1 = mod(large+inc,object->vert_num);
    if(finger== 2)
      {pt1 = object->vertex[large1];
       pt2 = object->vertex[large];}
    if(finger== 1)
      {pt1 = object->vertex[large];
       pt2 = object->vertex[large1];}

    r = sqrt(pow(Pd,2) + pow((il-const1),2));
    if((pt2.x-pt1.x) != 0 &&
        fabs((pt2.y-pt1.y)/(pt2.x-pt1.x))<1)
     {
      m = (pt2.y-pt1.y)/(pt2.x-pt1.x);
      b = (pt2.x*pt1.y - pt1.x*pt2.y)/(pt2.x-pt1.x);

      A = pow(m,2) + 1;
      B = 2*((b-p0.y)*m - p0.x);
      C = pow(p0.x,2) + pow((b-p0.y),2) - pow(r,2);

      for(i=-1;i<=1;i=i+2)
       for(j=-1;j<=1;j=j+2)
        {
         M.x = (-B + i*sqrt(pow(B,2)-4*A*C))/(2*A);
         M.y = p0.y + j*sqrt(pow(r,2) - pow((M.x-p0.x),2));
         if( between_pts(M, pt1, pt2, 0.05))
          break;
        }
     }
    else
     {
```

```
        c = (pt2.x-pt1.x)/(pt2.y-pt1.y);
        d = (pt1.x*pt2.y - pt2.x*pt1.y)/(pt2.y-pt1.y);

        A = pow(c,2) + 1;
        B = 2*((d-p0.x)*c - p0.y);
        C = pow(p0.y,2) + pow((d-p0.x),2) - pow(r,2);

        for(i=-1;i<=1;i=i+2)
         for(j=-1;j<=1;j=j+2)
          {
           M.y = (-B + i*sqrt(pow(B,2)-4*A*C))/(2*A);
           M.x = p0.x + j*sqrt(pow(r,2) - pow((M.y-p0.y),2));
           if( between_pts(M, pt1, pt2, 0.05))
            break;
          }
        }
        /* have found thet point (x,y) where the finger vertex
        intersects the object edge line */

        fw = (double) FW;
        if(finger==1)
         thetaf = 180 - ((asin(check((M.y-p0.y)/r)) +
               atan(fw/r))*180/PI);
        if(finger==2)
         thetaf = (asin(check((M.y-p0.y)/r)) +
               asin(check(fw/r)))*180/PI;
        return(thetaf);

      }
     else if(finger==2)
      large++;
     else if(finger==1)
      large--;

     large = mod(large,object->vert_num);


    }
   } /* !*d */
/*  free_dvector(theta, 0, OVN);
  free_dvector(length, 0, OVN);*/

}
```

```
void solve_move_finger(double thetaf[2][2], h_ptr h1, h_ptr h2,point palm_base, p_ptr
tobject, double thetap, rt_ptr **A, rt_ptr **B, rt_ptr **D, rt_ptr **T_frame, p_ptr
hand_list[7],int i,int *d)
{   int   done;
    double B00[3][3];
        done = *d;
        thetaf[i-1][0] = solve_finger_angle(h1, palm_base,
                        tobject,i, &done);
        if(!done)
         {
          m_initialize(B00, 3, 3);
          coord_transform(thetap, palm_base, B00);
          m_mult(B00, h1->finger_location[i-1]->rt);
          m_copy(h1->finger_location[i-1]->rt,
               A[i][0]->rt,3,3);
          move_finger(h1->finger[i-1], h2->finger[i-1],
            thetaf[i-1], B00, A[i], B[i], D[i],
            T_frame[i]);

          hand_list[i] = h2->finger[i-1]->link[0];
         }
        *d = done;

}

void generate_angular_velocities(int n, int n1, double thetafdot[2][2])
{
 double a;

   a = (PI/180)*(360/n)*n1;
   thetafdot[0][0] = cos(a);
   thetafdot[1][0] = sin(a);
}

void get_LP(double **a, ni fn, point obase,double thetafdot[2][2],
            double gext,point palm_base, h_ptr h)
{
   int      i,j,k,ind;
   point    m;
   double   line[3],dist_line(),td;

   a[1][1] =  0;
   a[2][1] =  0;
   a[3][1] = -gext;
```

```
    a[4][1] =  0;
    a[5][1] =  0;

    j=2;
    for(i=1;i<=6;i++)
     if(fn->n_ind[i]==1)
      {
        a[2][j] = -fn->n[i][0];
        a[3][j] = -fn->n[i][1];

        if(i==1 || i==5)
          {td = thetafdot[0][0];
           ind= 0;}
        else if(i==2 || i==6)
          {td = thetafdot[1][0];
           ind=1;}
        else
          td = 0;

        line[0] = fn->n[i][2];
        line[1] = fn->n[i][3];
        line[2] = fn->n[i][4];

        m.x =0; m.y=0;
        if( i!=3 && i!=4 )
         {m.x = palm_base.x + h->finger_location[ind]->rt[0][2];
          m.y = palm_base.y + h->finger_location[ind]->rt[1][2];}

        a[1][j] =-td*dist_line(m,line);
        a[4][j] = dist_line(obase,line);
        a[5][j] = 0.0;
        j++;
      }
      for(k=2;k<5;k++)
        if(a[k][1]<0)
          for(i=1;i<j;i++)
            a[k][i]=-a[k][i];
}

void get_wrench(double **a, ni fn, point obase)
{
   int      i,j;
   double   line[3],dist_line();
```

```
    j=0;
    for(i=1;i<=6;i++)
     if(fn->n_ind[i]==1)
      {
        a[0][j] = fn->n[i][0];
        a[1][j] = fn->n[i][1];
        line[0] = fn->n[i][2];
        line[1] = fn->n[i][3];
        line[2] = fn->n[i][4];
        a[2][j] = dist_line(obase,line);
        j++;
      }
}

void solve_force(p_ptr object,int n,ci_ptr ci,ni fn,
point obase, double thetaf[2][2], double otheta, FILE *file1, FILE *file2,FILE *file3,
point palm_base, h_ptr h, p_ptr hand_list[7], p_ptr object_list,char filename[10],
FILE *fileptr)
{  double **a,**m_create(),zero,sub,t11,t12,t21,t22;
   int i,j,ind,a1,a2;
   char text[80];
   point    m;
   double   line[3],dist_line(),c[7];

   call_ps (hand_list, object_list, filename, fileptr);
   zero = 0.0;
   t11 = 0.0; t12 = 0.0;
   t21 = 0.0; t22 = 0.0;
   if(n==3)
    {
     a = m_create(a,3,3);
     get_wrench(a, fn, obase);
    }
   else if(n==4)
    {
     a = m_create(a,3,4);
     get_wrench(a, fn, obase);
    }
   else if(n==5)
    {
     a = m_create(a,3,5);
     get_wrench(a, fn, obase);
    }
```

```
/* fprintf(file1,"%f %f\n",thetaf[0][0],thetaf[1][0]);
 fprintf(file2,"%f %f %f\n",obase.x,obase.y,otheta);
 fprintf(file3,"%s %d\n","3",n);
 for(i=1;i<=6;i++)
   if(fn->n_ind[i]==1)
     fprintf(file3,"%d ",i);
 fprintf(file3,"%s\n","  ");

  for(j=0;j<3;j++)
  {
   for(i=0;i<n;i++)
    fprintf(file3,"%f ",a[j][i]);
   fprintf(file3,"%s\n","  ");
  } */

 for(i=0;i<object->vert_num;i++)
  if(ci->v[i].contact==1 &&
     ci->v[i+object->vert_num].contact==1)
   fprintf(file1,"%s%d%d ","t", ci->v[i].contact_number,
           ci->v[i].contact_number);
  else if(ci->v[i].contact==1 &&
          ci->v[i+object->vert_num].contact!=1)
   fprintf(file1,"%d ",ci->v[i].contact_number);
  else
   fprintf(file1,"%d ",0);

 a1 = 0; a2 = 0;
 for(i=0;i<object->vert_num;i++)
  if(ci->e[i].contact==1 && ci->e[i].contact_with == 1)
    a1 = ci->e[i].contact_number;
  else if(ci->e[i].contact==1 && ci->e[i].contact_with== 2)
    a2 = ci->e[i].contact_number;
 fprintf(file1,"%d %d ",a1,a2);
 m_free(a,3);

 for(i=1;i<=6;i++)
  if(i!=3 && i!=4)
   if(fn->n_ind[i]==1)
   {
    line[0] = fn->n[i][2];
    line[1] = fn->n[i][3];
    line[2] = fn->n[i][4];

    if(i==1 || i==5)
```

```
        ind = 0;
      else
        ind = 1;
      m.x = palm_base.x + h->finger_location[ind]->rt[0][2];
      m.y = palm_base.y + h->finger_location[ind]->rt[1][2];

      if(i==1)
        t11 = dist_line(m,line);
      else if(i==5)
        t12 = dist_line(m,line);
      else if(i==2)
        t21 = dist_line(m,line);
      else if(i==6)
        t22 = dist_line(m,line);
     }
    fprintf(file1,"%.4f %.4f %.4f %.4f",t11,t12,t21,t22);
    fprintf(file1,"%s\n"," ");

}
```

## REFERENCES


[1] Brost R.C. "Computing Metric and Topological Properties of Configuration Space Obstacles," *proc. IEEE International Conference on Robotics and Automation*, 1989.

[2] Cutkosky M.R. and Kao I. "Dexterous Manipulation with Compliance Sliding," *proc. IEEE International Conference on Robotics and Automation*, 1989.

[3] Desai R.S. and Volz R.A. "Identification and Verification of Termination Conditions in Fine Motion in Presence of Sensor Errors and Geometric Uncertainties," *proc. IEEE International Conference on Robotics and Automation*, 1989.

[4] Desai R.S. On Fine Motion in Mechanical Assembly in Presence of Uncertainty, Ph.D. Dissertation, University of Michigan, June 1989.

[5] Denavit J. and Hartenberg R.S. "A Kinematic Notation for Lower-Pair Mechaniseus Based on Matricies",*ASME Journal of Applied Mechanics*,1955.

[6] Fearing R.S. "Simplified Grasping and Manipulation with Dexterous Robot Hands," *Proc. of the American Control Conference*, June 1984.

[7] Fearing R.S. "Implementing a Force Strategy for Object Re-orientation," *proc. IEEE International Conference on Robotics and Automation*, 1986.

[8] Jameson, J.W. Analytic Techniques for Automated Grasp, Ph.D. Dissertation, Stanford University, 1985.

[9] Kerr, J.R. An Analysis of Multifingered Hands, Ph.D. Dissertation, Stanford University, 1984.

[10] Lozano-Pèrez T. and Wesley M.A. "An Algorithm for Planning Collision Free Paths Among Polyhedral Obstacles," *Communications of ACM*, October 1979.

[11] Mason M.T. "How to Push a Block Along a Wall,"*NASA Conference on Space Telerobotics*, January 1989.

[12] Mason M.T. and Salisbury J.K., in *Robot Hands and the Mechanics of Manipulation*, MIT Press, Cambridge, Mass.,1985.

[13] Paul R.P., in *Robotic Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Mass., 1981.

[14] Peshkin, M.A. and Sanderson, A.C. "Minimization of Energy in Quasistatic Manipulation," *proc. IEEE International Conference on Robotics and Automation*, 1988.

[15] Trinkle, J. C., Abel, J. M. and Paul, R. P. "An Investigation of Frictionless Enveloping Grasping in the Plane," *International Journal of Robotics Research*, June 1988.

[16] Trinkle, J.C. and Paul, R.P. "Planning for Dexterous Manipulation with Sliding Contacts," *International Journal of Robotics Research*, June 1988.

[17] Trinkle, J.C. The Mechanics and Planning of Enveloping Grasp, Ph.D. Dissertation, University of Pennsylvania, June 1988.

[18] Wolovich W., in *Robotics: Basic Analysis and Design*, CBS College Publishing, New York, NY,1987.