MODEL BASED TACTILE OBJECT RECOGNITION FOR POLYHEDRA

by

Curtis F. Elia

Memorandum No. UCB/ERL M86/83

15 May 1986

MODEL BASED TACTILE OBJECT RECOGNITION FOR POLYHEDRA

by

Curtis F. Elia

Memorandum No. UCB/ERL M86/83

15 May 1986

M.S. report supervised by Professor S. Sastry

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

MODEL BASED TACTILE OBJECT RECOGNITION FOR POLYHEDRA

by

Curtis F. Elia

Memorandum No. UCB/ERL M86/83

15 May 1986

M.S. report supervised by Professor S. Sastry

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Model Based Tactile Object Recognition for Polyhedra

Curtis F. Elia

University of California, Berkeley

May, 1986

ABSTRACT

This report discusses the application of tactile sensing to tactile object recognition. Previous work in the literature is reviewed, and an implementation of an object recognition scheme for polyhedra is presented. The scheme is model based and utilizes sparse modeling data to generate constraints for possible matchings. The recognition algorithm uses template matching to match measured data readings from a tactile pad to possible contact patterns generated from the object models. Test results are reported to demonstrate the performance of the recognition scheme.

Acknowledgements

The author wishes to thank the following people for their support: John Hauser for not only his technical and computational support, but also for his willingness to help debug computer code, Gregory Heinzinger for his typesetting skills, which saved the author more than once, and Dr. Brad Paden for his valuable comments on the original draft. Partial support was also provided by Semiconductor Research Corporation SRC-82-11-008.

Table of Contents

1. Introduction

Robots are used for a variety of tasks in industry today. Typical examples are spot welding, spray painting, assembly, and hazardous element handling. Usually, the robots are preprogrammed to perform defined tasks in a structured environment. Because of their controlled surroundings, the robots are able to perform well with very primitive sensing capabilities.

Although a controlled environment allows the robot to perform satisfactorily with few sensors, it is costly and limiting. As applications for robots increase, and robots become more adept at their tasks, so will the need for robots to be capable of adapting to their environment. Future *intelligent* robots will need a dynamic understanding of their environment. They will need to employ *intelligent* sensors to improve their senses.

Modern robots sense their environment primarily in two ways, through vision and touch. These complementary senses are used in many robotic tasks. Vision systems have found applications in object recognition, obstacle avoidance, and proximity sensing. Touch is necessary when contact with an object must be made. Typical applications of tactile sensors include locating and identifying objects and grasping.

This paper reports on the development of an *intelligent* sensor — a two-dimensional tactile recognition scheme for polyhedra. The scheme is model based; it seeks to match a measured *footprint* of an object, obtained by a tactile pad measurement, with all possible footprints generated using models of a set of known objects.

The outline of this paper is as follows. Chapter 2 surveys the literature on tactile object recognition and template matching. Chapter 3 describes the

developed model based recognition scheme. Chapter 4 makes recommendations for future work.

2. Tactile Object Recognition

Tactile information has many applications. A two-dimensional pad can be used in assembly tasks where the contact pattern of a part can be used to recognize it and determine its orientation. Three or more tactile sensors can be used for recognition and grasping applications.

Tactile object recognition involves identifying an object from a set of known objects using tactile information. Two-dimensional tactile information is obtained by using a matrix sensor, such as a tactile pad. It is useful for obtaining a contact distribution, or footprint, of the object. Three-dimensional information is obtained by using multiple tactile sensors. It is useful for obtaining relative positional data.

Although the recognition scheme discussed in Chapter 3 is two-dimensional, much work has been done using three-dimensional tactile information. This chapter will describe some of these methods as well as a two-dimensional method and two-dimensional template matching schemes.

2.1. Approaches to Tactile Object Recognition

There are basically two types of tactile recognition schemes — statistical schemes and model based schemes. Statistical pattern recognition seeks to compare measured statistics of an object with its reference statistics and classify an object from one of many specified classes into a specific class. A common measurement device used is an articulated hand grasping the object.

Statistical recognition schemes utilize discriminant functions. A discriminant function is a real valued function $d(x)$, $x \in \mathbf{R}^n$, where n is the number of measured properties, which assigns a value to a measurement x. Given m specified object classes, the objective of the recognition scheme is to derive m discriminant

functions such that for any measurement, x, from class i, $d_i(x) > d_j(x)$, $j \neq i$, $1 \leqslant j \leqslant m$.

The discriminant functions are usually learned by using some training pattern of measurements. Typically, the functions are represented as polynomials, and their coefficients are determined by using measurements corresponding to known classes. The process is called *supervised learning* if the classes of the measurement vectors of the training pattern are known and *unsupervised learning* if they are not known.

One method, developed by Okada and Tsuchiya [1], involves recognizing the size and the three-dimensional pattern of an object using sensors placed on multi-jointed fingers grasping the object. The recognition scheme involves two stages. The first stage classifies the objects based on the distribution pattern of the sensors. The second stage uses the finger's joint angles to distinguish the different classifications. The recognition is obtained by evaluating several discriminating functions provided for each contact pattern. Both linear and quadratic functions were computed.

A second method, developed by Briot *et. al.* [2], classifies an object, picked randomly from one of three classes, into the most probable class. The probability density used to make this classification is initially unknown, but is estimated using measurements of joint angles of the hand grasping the object. The initial distribution of the observations is modified by a supervised learning procedure if the classification is incorrect.

Since these statistical methods try to classify an object into a specific class, they allow general object types. However, this generality tends to restrict the classifications, so that these methods only differentiate between a small number of

simple object classes. Further references are contained in [3] and [4].

Model based pattern recognition, on the other hand, restricts the object types so that a more specific classification can be made. These recognition schemes seek to compare measured data values to those of a complete model. Successive measurements create a partial model of the object which is compared to the complete model. Most methods use the relative positional information obtained from many small tactile sensors as their data. Other methods use contact patterns distributions obtained from fewer, but larger sensors.

One group of model based recognition schemes, called feature extraction schemes, utilizes contact patterns to detect global features of the objects for recognition. The measurements of a matrix sensor are used to observe identifying features of an object, such as holes, edges, corners, and distinguishing marks. The features are then compared to a complete model for recognition. Problems can occur, however, in relating successive measurements and when the feature size is greater than the size of the sensor.

Another group of model based schemes uses relative positional information. Gaston and Lozano-Perez [3] proposed a two-dimensional method for recognizing polyhedra in the plane which utilizes *local* information obtained from three or more sensors. Each sensor is assumed to provide the position of contact and a range of surface normals at the contact point. The recognition scheme consists of successively pruning the levels of an interpretation tree.

The interpretation tree for a single object represents the possible pairings of contact points of the data to edges of the model. The tools used to prune inconsistent interpretations are a distant constraint, an angle constraint, and a model constraint. The distant constraint ensures that the distance between each pair of

contact points is a possible distance between corresponding paired edges of the model. The angle constraint ensures that the range of angles between measured surface normals for each pair of contact points includes the angle between surface normals of the corresponding paired edges of the model. It also checks if the approach angle of the sensor is possible. The model constraint ensures that the position of the measured contact points correspond to some point on their corresponding edges for some configuration of the model. Thus, a complete model for this scheme includes a table of distance ranges between pairs of edges and a table of angles between surface normals for each pair of edges.

2.2. Approaches to Template Matching

The template matching, or point set matching, problem has been studied in applications of machine vision recognition and location of rigid planar shapes. It involves being given two sets with an equal number of points (one model and the other a noisy data measurement) and finding a registration which gives a matching minimizing the sum of the squares of the pointwise location errors. The registration is a planar transformation which consists of translation, rotation, and scaling. Thus, template matching is a feature extraction method, where each point can be considered a feature of the object.

There are several approaches to solving this problem. Simon *et. al.* [5] exploit the relative distance between points in each set of points. They compute the distance for each pairing of points in each set and then compare ordered lists of these distances to constrain the matchings. Since this matching uses only relative positions, it is independent of translation, rotation and scale.

Baird and Steiglitz [6] use linear programming techniques to solve this problem. The total number of matchings is constrained by testing the feasibility of a set of linear inequalities. Thus, a small set of possible matchings is determined from which to choose a best matching. The number of partial testings can be reduced by exploiting registration constraints on translation, rotation, and scale.

Both of these approaches assume that the number of points in each set is equal. For recognition applications, however, it is desirable to take missing or spurious points into account. This is especially true when the set of objects contains unsimilar features or when the feature detector is noisy.

To avoid the computations of an exhaustive search, some methods seek only to compare specific subsets of the two point sets. Goshtasby and Stockman [7] proposed a method which can match the greatest number of points in two sets which do not necessarily contain an equal number of points. To reduce the amount of computation, a subset of each set is matched. The subset chosen is the set of points on the boundary of the convex hull of each set. This choice of subset has three advantages. First, it is invariant to translation, rotation, and scale. Second, it is likely to be a sparse set. Finally, it is likely to have longer edges, which will give a better estimate of the registration.

The method is described as follows:

(1) Determine both subsets by finding the points on the boundary of the convex hull of each set.

(2) Determine a transformation consisting of translation, rotation, and scale which maps an edge in the first subset to an edge in the second subset.

(3) For each transformation, determine the total number of points that are matched from all points in the first set to points in the second set.

(4) Using the transformation which matches the most number of points, determine the corresponding points in the two sets which match within some threshold distance.

(5) Determine the optimal transformation by minimizing the sum of the squared errors of those matched points.

There are two types of edges that can be matched in step (2). The edges can be those of the boundary of each subset or those corresponding to complete graphs of each subset. A complete graph consists of all possible pairings of points in a subset. Thus, to reduce calculations, the boundary edges of each set can be matched first. If this does not produce an adequate matching then the edges of complete graphs of each subset can be matched.

3. Model Based Object Recognition Scheme For Polyhedra

This chapter describes the intelligent model based tactile recognition scheme that was developed. This scheme is intended to be used for recognition of an object from a known group of objects using the footprint of the object, obtained by a tactile pad. The pad is assumed to be bigger than the object and to give a binary output. The objects are assumed to be polyhedra or are able to be suitably modeled as polyhedra.

This scheme, like those presented earlier, utilizes an object model to compare to data measurements. The intelligence of this scheme is due to its ability to generate the model constraints it uses to constrain matchings. Methods like that of Gaston and Lozano-Perez [3] require predefined tables for their model constraints. This scheme uses sparse modeling information to generate the model constraints automatically.

The basic flow of the algorithm is illustrated in Figure 3.1. A set of objects is assumed to be known. The objective is to identify an object on a tactile pad as one of the known objects.

There are two branches, labeled *model* and *data*, leading from the set of objects. In the model branch, all objects are modeled mathematically and analyzed to obtain models of all possible *stable footprints* that they can leave on a tactile pad. A stable footprint is the contact pattern an object makes when it is supported solely by the pad. Furthermore, the footprint should be invariant under small perturbations to the object. Thus, stable footprints of an object can not be composed of a sole point, edge, or colinear combination of each. The corresponding area of every stable footprint is also calculated. The stable footprints and areas are saved for the input of the recognition algorithm.

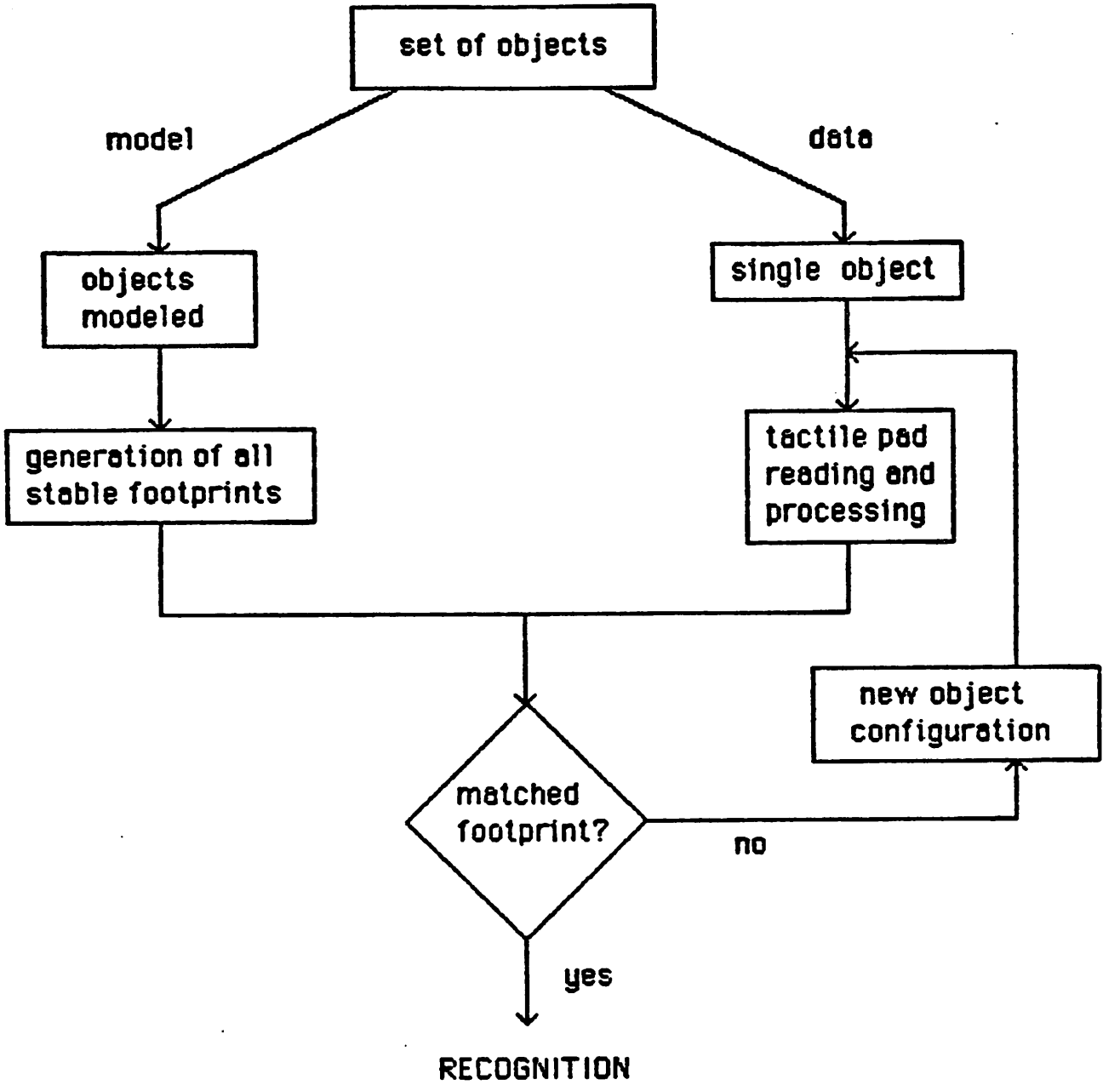


Figure 3.1 Recognition Scheme Flow

In the data branch, a footprint measurement from one object is obtained and processed. The measured footprint is in the form of a two-dimensional binary array. The processing consists of representing the footprint in a similar manner as the generated modeled footprints and determining the corresponding area. The representation of the modeled footprint will require edge and corner detection of the measured footprint.

The recognition algorithm takes the processed footprint generated by the pad measurement and compares it with all possible footprints determined in the model branch. The recognition is complete if the measured footprint is matched to one object. Otherwise, the object will have to be reconfigured on the pad and another measurement taken for another comparison. The recognition scheme is a template matching method which seeks to match the greatest number of corners of the measured footprint to corners of the generated model footprints, as well as insuring that the areas are comparable. The following sections will describe each of the steps in more detail.

3.1. Object Model

The object's geometry is modeled as the union of polytopes. A polytope is the convex hull, co(S), of a given set of points $S = \{p_1, p_2, \ldots, p_m\}$ in R^n, i.e.

$$\mathrm{co}(S) = \{ x : \sum_{k=1}^{m} w_k p_k, \ \sum_{k=1}^{m} w_k = 1, \ w_k \geq 0 \quad 1 \leq k \leq m \}.$$

A polytopal representation of a cube, for example, is the set of its corners. To obtain an arbitrary polyhedron, it is necessary to model it as the union of polytopes since the union can be nonconvex. Figure 3.2 shows an example of a modeled table.

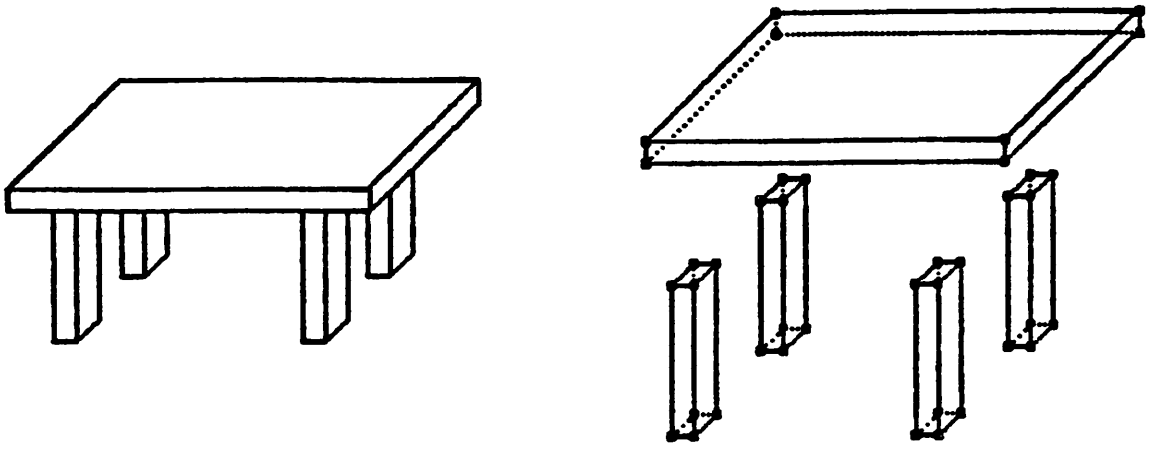


Figure 3.2 Modeled Table

The polytopal representation has two big advantages. First, convexity is a strong property which simplifies many computations. Second, representing an object by a set of points is computationally easy . There is one disadvantage to this model, however. Unioning polytopes to model nonconvex polyhedra can introduce additional points which are not corners of the object. Although decomposing a polyhedron into polytopes is not difficult, recomposing must be done carefully. Figure 3.3 shows the decomposition of a two-dimensional block L. Note that point 1 should not be treated as a corner of the union.

In addition to the geometrical modeling, there must be some physical modeling as well. Specifically, it will be necessary to estimate the center of mass of the resulting object for determination of stable footprints. Thus, the center of mass and total mass of each polytope must be given. Although this may not be trivial in some cases, the estimates for each separate polytope should be easier than that of the composite object.

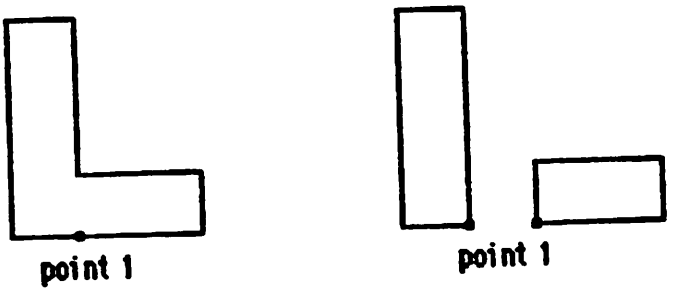


Figure 3.3 Modeled Block L

In total, the model of an object consists of distinct sets of points, an associated center of mass and total mass for each set, and a flag in each set indicating all noncorner points created in modeling nonconvex objects.

3.2. Generation of Stable Footprints

Due to the choice of model, the generation of stable footprints is relatively simple. This task is broken into two steps: (1) determination of all possible footprints that an object can leave (stable or not) and, (2) application of a constraint which determines the stability of the footprint. These two steps will be considered separately.

3.2.1. Determination of All Footprints

Determining all footprints that an object can leave on a tactile pad is a geometrical problem. Thus, the geometry of the object model should be exploited. The problem is to find the intersection of specific support planes with the object. In R^3, a support plane of an object is a plane in which all points of the object lie on one side of the plane and at least one point of the object lies in the plane. To obtain a non trivial footprint of the object, the support plane must intersect the

object in three or more noncolinear points. Thus, single point and single edge contacts are excluded. By taking the convex hull of the object and finding a face of the resulting polytope, a planar intersection of a support plane and the convex hull of a footprint is found. The problem, therefore, reduces to finding all faces of a polytope and extracting the corresponding footprints from them.

The method used to find all faces of a polytope was developed by Chand and Kapur [8]. It has been called the "gift-wrapping method" [9]. The algorithm is based on the observation that every edge of the polytope is the intersection of exactly two faces. Thus, given one edge and one corresponding face, the second face can be found by rotating the known face around the known edge by a specific angle. Once a face has been found, its edges are determined similarly and the process proceeds until all edges have two corresponding faces which are found. A method is also given to compute the initial face and edge.

Let $S = \{p_0, p_1, \ldots, p_N\}$ be the set of points in R^n describing the polytope co(S). Let $p_0, p_1, \ldots, p_{n-2}$ be (n - 1) linearly independent points of S that define an n-edge E of co(S). (A point p is linearly independent on a subset $X \subset R^r$ if there are no scalars λ_i $1 \leqslant i \leqslant r$, not all zero, and points $x_i \in X$, such that $p = \sum_{i=1}^{r} \lambda_i x_i$. Likewise, a set of points is linearly independent if no point in the set can be expressed as a linear combination of the remaining points.) Let n be the inward unit normal of a support plane H, containing E and one face of co(S). Letting v_i be the unit vector along $p_i - p_0$,

$$(n \cdot v_i) \geqslant 0 \quad i = 1, 2, \ldots, N.$$

The objective is to find the appropriate support plane containing the adjacent face.

Adjoining a point p_k, $p_k \in S$, $p_k \notin H$, to E gives n linearly independent points

which describe a hyperplane with unit normal n_k $((n_k \cdot n) \geq 0)$ which is possibly the support plane containing the adjacent face. In general, an (n - 1) × n linear homogeneous system of equations must be solved to determine n_k. The adjacent face is contained in the hyperplane which makes the largest convex angle ($< \pi$) with n_k since this is the only hyperplane created which is a support plane. Thus, to find the appropriate normal a (n - 1) × n linear system of homogeneous equations would need to be solved (N - n - 1) times.

By comparing the cotangent of the angles between n and all n_k, this problem can be solved by solving one (n - 1) × n system of linear homogeneous equations. Let e be a unit vector orthogonal to both n and E. To determine e, the following (n - 1) × n linear homogeneous systems of equations must be solved.

$$(e \cdot n) = 0$$

$$(e \cdot v_i) = 0 \quad i = 1, 2, \ldots, n-2$$

The orientation of e is given by $(e \cdot v_k) > 0$, $p_k \in H$, $p_k \notin E$, i.e. e points in the direction of the other points of the face contained in H.

The n vectors $v_1, v_2, \ldots, v_{n-2}, n, e$ form a basis for R^n. Since n is normal to E, n_k must lie in the plane spanned by n and e. Letting $n_k = \lambda_k n + \mu_k e$, the cotangent of the angle between n and n_k is given by

$$\tau_k = -\frac{(e \cdot v_k)}{(n \cdot v_k)} = \frac{\lambda_k}{\mu_k}.$$

The normal of the adjacent face is found by computing

$$\max_k \frac{\lambda_k}{\mu_k} \quad \text{for } p_k \in S, p_k \notin H.$$

Figure 3.4 illustrates this for the case where co(S) is in R^3. In this example, the plane given by p_0, p_1, and p_4 contains the adjacent face. Since $(e \cdot v_2)$ and

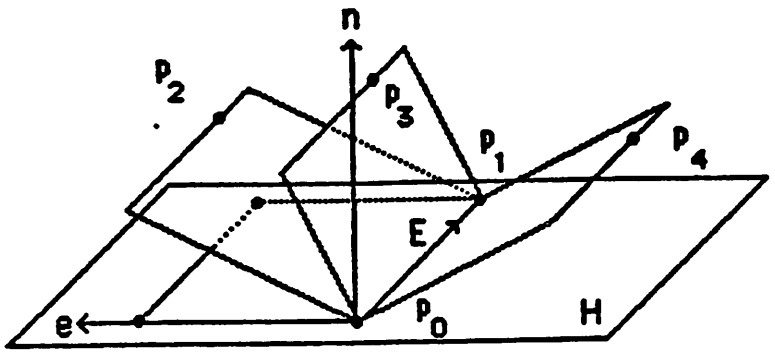


Figure 3.4 Gift-Wrapping Method in R^3

$(e \cdot v_3) > 0$, τ_2 and $\tau_3 < 0$, but $\tau_4 > 0$ since $(e \cdot v_4) < 0$.

Once an n-face of co(S) is found, its edges can be found similarly since they are an (n - 1) face of co(S'), where S' equals the set of points in the n-face of co(S) contained in S.

Finally, the initial face can be obtained through the application of the following theorem (for a proof see [8]).

> Let H be a support plane of S, containing r linearly independent points of S, whose normal $n = (a_1, a_2, ..., a_r, 0, ..., 0)$. Then there exists at least one point of S which when joined to the r points of S on H forms a linearly independent subset S* of S which lies in a support plane H* of S whose normal is of the form $n^* = (b_1, b_2, ..., b_{r+1}, 0, ..., 0)$.

Thus, finding points in S with least first component determines a hyperplane H with normal, $n_1 = (1, 0, ..., 0)$ and the theorem above can be applied n - 1 times to obtain the normal of the support plane containing an n-face of co(S).

Therefore, the faces of the polytope can be found as follows:

Step 1: Find an initial face and a corresponding edge. Determine and store all edges of the face with its normal.

Step 2: Search the storage for an edge for which only one corresponding face has been found.

Step 3: If no edge exists then exit.
Otherwise, determine the adjacent face and its corresponding edges and store all edges with the normal of the adjacent face.
Go to step 2.

The gift-wrapping method generates all faces of the polytope. Since the polytope is the convex hull of the modeled object, each face represents the convex hull of a possible footprint of the object. The actual footprint can be extracted by determining all model points in the support plane containing the face and unioning the corresponding faces of the individual polytopes unioned to model the object.

3.2.2. Constraint for Stable Footprints

After a face of the convex hull of an object has been found, two things must be done. The footprint must be extracted from the face, and it must be tested to see if it is stable. Extracting the footprint from the convex hull is easy due to the way the object has been modeled; however, the latter problem is more difficult and is the subject of this section.

A footprint is stable if the object creating it is sitting on the tactile pad, supported solely by the pad, and if the footprint is invariant under small perturbations of the object. The invariance constraint rules out any single point or single edge footprints. Satisfying the support constraint means that the object must sit on the pad without "falling over".

One way to determine if the object will fall over is to see if its center of mass projected perpendicularly down to the plane of the pad is contained in the convex

hull of the footprint. If this is the case, forces will balance and there will be no resultant torque which could cause the object to topple over. If the projection point of the center of mass is not contained in the convex hull of the footprint then a resultant torque is produced which will cause the object to fall. Thus, checking the stability of the footprint reduces to checking if the center of mass of the object projected perpendicularly down to the plane of the tactile pad lies in each face found by the gift-wrapping method described in the last section.

To determine if a point is in the convex hull of a set of points, a method developed by Wolfe [10] is used. This method determines the nearest point to the origin of a polytope. After subtracting the projected center of mass point from each point on the boundary of the convex hull of the footprint, the projected point is in the convex hull of the footprint if the distance from the nearest point to the origin is zero. The corresponding footprint is stable if the containment holds.

Given a set of points $P = \{p_0, p_1, \ldots, p_m\}$ in R^n, the problem of finding the nearest point of co(P) to the origin is

$$\min x'x \quad \text{subject to } x = \sum_{i=1}^{m} p_i w_i,\ \sum_{i=1}^{m} w_i = 1,\ w_i \geqslant 0 \quad 1 \leqslant i \leqslant m. \qquad \text{NR1}$$

Letting Q be a $n \times l$ $(l \leqslant m)$ matrix of a subset of points in P, Q is affinely independent when no point of Q belongs to the affine hull of the remaining points, where the affine hull is given by

$$\text{aff}(Q) = \{ x : x = Qw,\ \sum_{i=1}^{m} w_i = 1 \}.$$

When Q is affinely independent, the solution to the problem

$$\min x'x = \min w'Q'Qw \quad \text{where } \sum_{i=1}^{m} w_i = 1 \qquad \text{NR2}$$

can be obtained by using Lagrange multipliers. Note that NR2 is a subset problem of NR1 since P may not necessarily be affinely independent. If it turns out that

$w_i \geq 0$, $1 \leq i \leq l$, then x^*, the solution of NR2, will belong to co(Q), the convex hull of Q. If Q is a suitable subset of P then x^* may minimize over co(P), solving NR1 and determining the nearest point of the polytope co(P) to the origin. Specifically, if the hyperplane $H(x) = \{ y : x'y = |x|^2 \}$ is a support plane for the convex hull of P then x is the unique solution (since $|x|^2$ is strictly convex) and the original problem, NR1, is solved.

The Wolfe algorithm determines points x^* for certain sets Q called *corrals*. A corral is defined to be an affinely independent subset Q of points in P such that the nearest point in Q to the origin is in its relative interior. The algorithm finds the corral which contains the solution to the original problem, NR1, by solving NR2 a finite number of times.

The algorithm is as follows:

Data: points in P

Parameters: points x, y, z and corral Q

Step 0: Find a point in P with smallest norm. Set x = that point. Q = $\{x\}$.

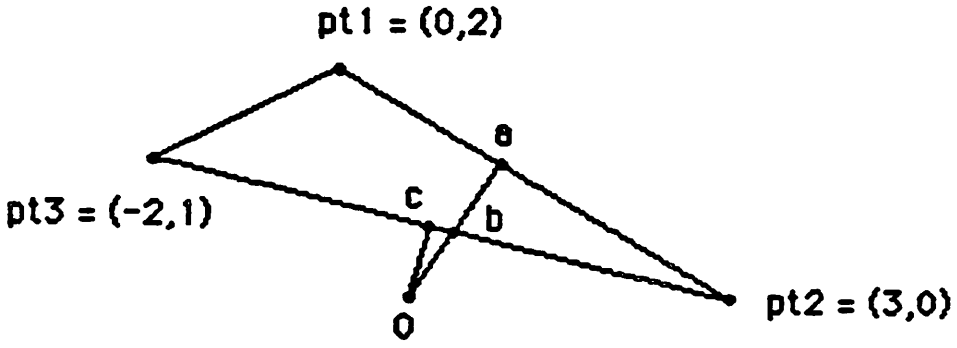
Step 1: If $x = 0$ or $H(x)$ separates P from the origin then stop.
Otherwise, choose $p_i \in$ P on the near side of $H(x)$ and set Q = Q $\bigcup \{p_i\}$.

Step 2: Set y = point of smallest norm in aff(Q).
If y is in the relative interior of co(Q) then set $x = y$ and go to step 1,
else

Step 3: Set z = nearest point to y on the line segment co(Q) $\bigcap \overline{xy}$ (the line segment from x to y). Delete from Q one of the points not on the face of co(Q) which contains z. Set $x = z$ and go to step 2.

The algorithm is composed of a finite number of major cycles (started in step 1), each of which may consist of a finite number of minor cycles (repetition of steps 3 and 2). Each major cycle begins with a corral Q and a nearest point in Q. The cycle chooses a new point to add to Q and determines if the result is a corral.

If it is not, a minor cycle begins. The minor cycle begins with an affinely independent set Q and a point $x \in co(Q)$. The cycle removes a point from Q and alters x until Q is a corral and x is the nearest point in Q. Figure 3.5 illustrates the operation of the algorithm for a simple example presented by Wolfe.



Step	x	Q	y
0	pt_1	pt_1	
1	pt_1	pt_1, pt_2	
2	a	pt_1, pt_2	a
1	a	pt_1, pt_2, pt_3	a
2	a	pt_1, pt_2, pt_3	0
3	b	pt_2, pt_3	0
2	c	pt_2, pt_3	c
1			

Figure 3.5 Example Of Wolfe's Algorithm

3.3. Data Analysis

The data provided by the tactile pad is assumed to be a two-dimensional array of binary values corresponding to one footprint of the object. It is assumed

that the background is given value zero and the footprint value one. In order to compare this data with generated footprints, the data must be processed to put it in the same form, i.e. the corners of the measured footprint must be obtained. To obtain these corners there are two levels of processing — edge detection and corner detection.

3.3.1. Edge Detection

The first step of processing the tactile data is to separate the object footprint boundary from the background. Edge detection techniques are designed to do this. They are based on the assumption that there is a contrast in data levels at an object's edge. Thus, most edge detection techniques use a spatial convolution mask which is applied to each pixel, checking a small neighborhood around that pixel to determine any discontinuity in the data.

One edge detection method, the gradient method, uses a 3 × 3 pixel area and approximates the gradient in the x direction of position (i, j) by

$$G_x = (d[i+1][j-1] + 2d[i+1][j] + d[i+1][j+1]) - (d[i-1][j-1] + 2d[i-1][j] + d[i-1][j+1])$$

and the gradient in the y direction by

$$G_y = (d[i-1][j+1] + 2d[i][j+1] + d[i+1][j+1]) - (d[i-1][j-1] + 2d[i][j-1] + d[i+1][j-1]),$$

where d[i][j] is the data value of position (i, j). The magnitude of the gradient is given by

$$G=[G_x^2 + G_y^2]^{\frac{1}{2}}$$

or approximated as

$$G = |G_x| + |G_y|.$$

The corresponding convolution masks are shown in Figures 3.6 (a) and 3.6 (b) and are typically called the Sobel operator masks.

-1	-2	-1
0	0	0
1	2	1

(a) G_x

-1	0	1
-2	0	2
-1	0	1

(b) G_y

Figure 3.6 Sobel Operator Masks

The data analysis scheme uses a method similar to the gradient method for edge detecting the binary tactile data. One disadvantage of the gradient method, however, is that it smears edges. This is due to the 3 × 3 size of the mask. This is undesirable since it could create errors in determining the corners of the unprocessed data. To avoid smearing, the computed values could be thresholded, so small values would be set to zero. Likewise, the computed value could be unioned with the original data, setting a value that was initially zero to zero automatically. For edge detecting the data, the latter was used. Due to the simplifications of the binary output, a rule based edge detection scheme can be created by checking the number of ones appearing in the eight adjacent neighboring pixels. The rule based scheme is shown in Figure 3.7. With this scheme, background points are kept the same, the outer two "levels" of the object boundary are kept the same, and the remaining interior points are set to zero.

neighboring 1's	new pixel value
0, 8	0
1, 2, ..., 7	same

Figure 3.7 Rule Based Edge Detector

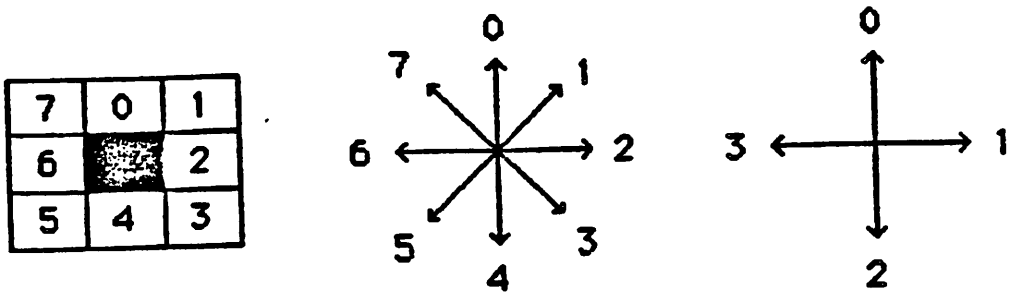
3.3.2. Corner Detection

It is necessary to extract the corners from the measured footprint in order to compare it with possible calculated footprints. The method used to corner detect is similar to that used in binary image chain coding schemes.

A chain coding scheme is a method which describes the outline of an object in an image. One such scheme described by Wilf [11] traverses the boundary of an object, storing the direction traversed at each pixel as well as updating additional information, such as the area and moments of inertia in the x and y direction. Possible direction codes to use are shown in Figure 3.8. A pixel and its eight neighbors are shown in Figure 3.8 (a). The eight possible directions to travel from the center pixel are represented in Figure 3.8 (b) by an eight-directional code. It is also possible to describe an objects boundary using a four-dimensional code as shown in Figure 3.8 (c). The resulting array of values and associated starting location is called a *chain code* and can be used in a recognition scheme by analyzing its values along with the values of the additional information.

The method used to determine the corners of the measured footprint is similar to chain coding. Each isolated region of the footprint is traversed using an eight-directional direction code, as shown in Figure 3.8 (b). A global list of directions traversed is not saved, however. Rather, a local list of directions is used. A corner

is detected whenever the direction changes by more than one unit or if the direction changes by one unit but fails to return to the original direction and maintain it for a specified number of pixels. In addition, each new corner found is checked to see if it is greater than a threshold distance from the last, thus preventing two corners close together from impersonating one actual corner.



(a) Neighboring Pixels (b) Eight-Directional Code (c) Four-Directional Code

Figure 3.8 Direction Codes

3.3.3. Data Analysis Scheme

The data analysis scheme uses both the modified edge detection and the corner detection methods mentioned previously. After the edge detection, the data is composed of all zeros except for ones along the boundary of the footprint. This boundary is no more than two pixels wide. The corner detection scheme finds the first one it comes to and traverses the boundary of that part of the footprint, leaving a different flag for edge pixels and corner pixels. The "leftover" ones immediately interior to the edge are then changed to zeros and the process is repeated for the remainder of the footprint, in the event it is composed of multiple isolated

parts.

Because the corners are detected using boundary traversal, edge detection is not strictly necessary. The same procedure can be used without the edge detection, except that after traversal of the boundary, all interior pixels would have to be ignored when looking for the next part of the footprint. However, edge detecting first could filter some noise and smooth noisy edges if a more sophisticated scheme is used. For these reasons, it was kept in the processing scheme to accommodate easy modification in the future.

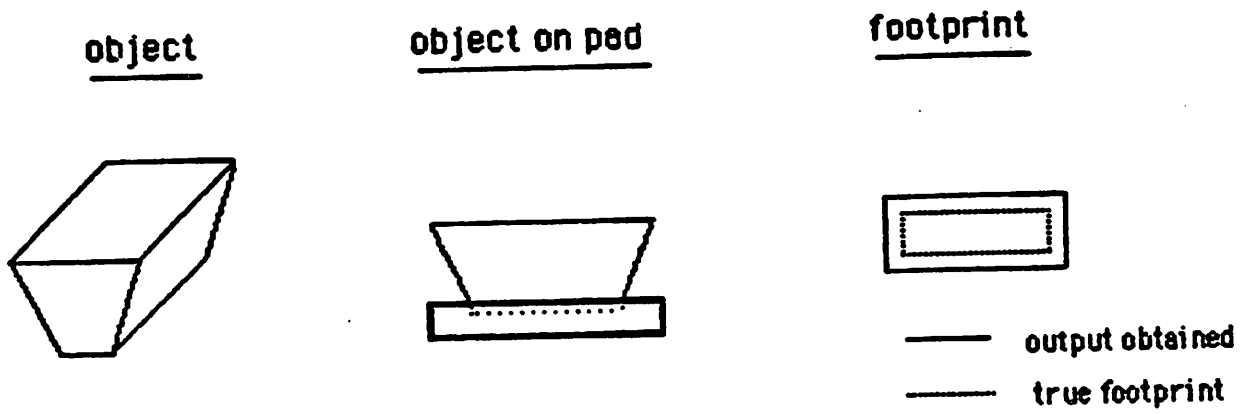
3.4. Recognition Scheme

The input to the recognition scheme consists of one set of data points, D, corresponding to all corners of the measured footprint and a database of many sets of points, FP_i^k, corresponding to the corners of all possible stable footprints, i, of each object, k. The corresponding area of each footprint is available as well. The objective is to find a k for which D is matched to the greatest number of points in FP_i^k as well as matching the corresponding area. The problem fits nicely into the format of a template matching problem since sets of points are being matched.

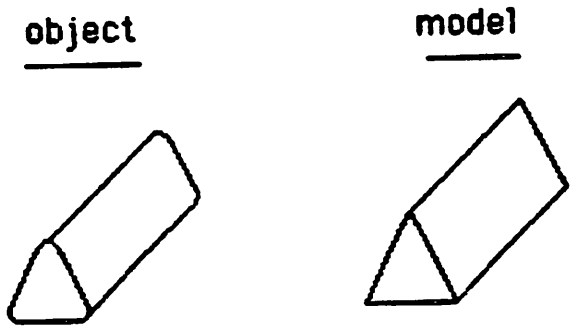
3.4.1. Robustness Issues

The recognition algorithm developed must be able to function in the presence of errors. There are two main types of errors possible — errors in the tactile pad measurement and modeling errors.

Errors in the tactile pad measurement can be due to a variety of uncertainties. One problem that can result is due to the compliance of the pad. As illustrated in Figure 3.9 (a), the measured footprint will be larger than the actual footprint if



(a) Compliance Error



(b) Modeling Error

Figure 3.9 Possible Errors

the the objects sinks in the pad. The resulting error causes the corners to be off by some error range. The same problem can occur if the pad has hysteresis problems or if its parameters drift.

Errors in modeling are due mainly to the restriction of polyhedral objects. Figure 3.9 (b) illustrates modeling a rounded triangular prism as a triangular prism. Once again, it is necessary that the matching scheme be able to match points within some error range.

3.4.2. Algorithm

The algorithm for comparing the corners of the measured data and the corners of a possible footprint is similar to that of Goshtasby and Stockman [7]. First, the points on the boundary of the convex hull are computed and ordered for each set. These points are already obtained for each FP_i^k in the process of the gift-wrapping method. These points can be obtained for D by a similar procedure. Let these sets be $\overline{FP}_i^k$ and $\overline{D}$.

Next, every edge of $\overline{D}$ is matched to all possible edges in $\overline{FP}_i^k$ for every face i, of every object, k. For every edge there are two possible orientations to match as well as two different endpoint configurations. Figure 3.10 shows these matchings. Figure 3.11 shows the corresponding matchings for matching only one edge of the convex hulls of two footprints. The best match results in the greatest number of corners matching.

For each possible matching, all points of the measured footprint, D, are transformed by a planar rigid motion to the space containing the calculated footprint, FP_i^k. The transformation which results in the greatest number of points being matched within some threshold distance determines the group of *possible* best matches. An area constraint is applied to this group to determine the final group of best matches. The footprint area constraint is necessary since two footprints can match the same number of points if they are identical except that one has an

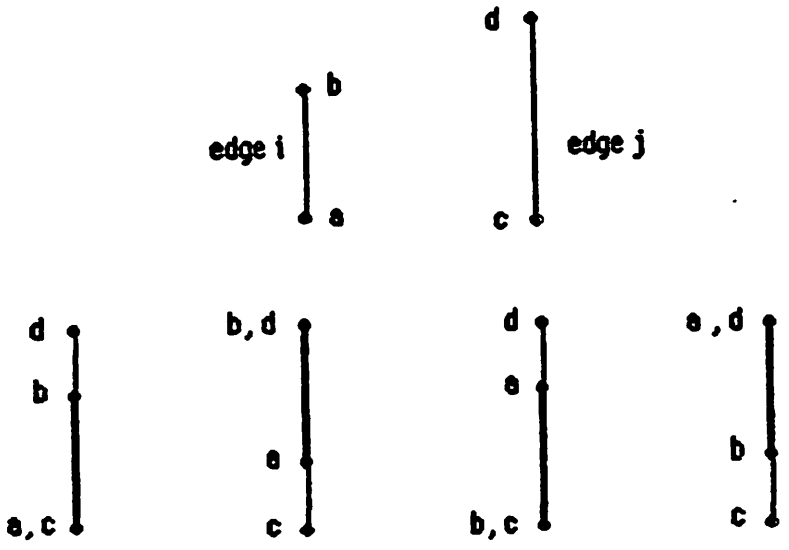


Figure 3.10 Possible Edge Matchings

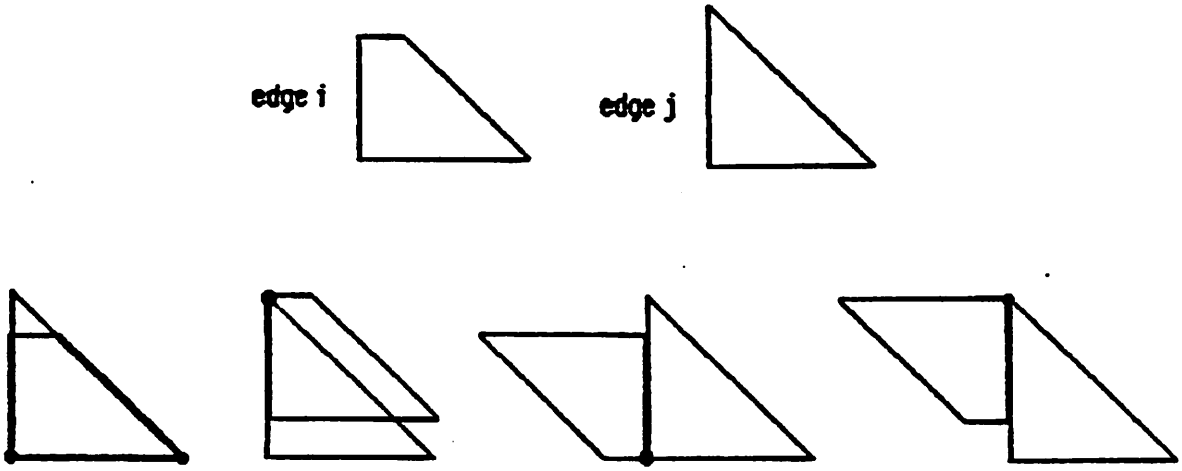


Figure 3.11 Matching One Edge of Two Footprints

additional isolated region in the interior of the convex hull, and the measured footprint is from the object without the additional part.

The algorithm is as follows:

Data: Sets D, $\overline{D}$, FP_i^k, $\overline{FP_i}^k$. Areas of D and each FP_i^k.
For each possible object k:
  For each face of each possible object i:
    For each edge in $\overline{D}$:
      For each edge in $\overline{FP_i}^k$:
        For both orientations and endpoint matchings:
          Determine the planar rigid motion to match the edges.
          Determine and store the number of points in D matched to points in FP_i^k within some threshold distance.
  Determine the objects which matched the maximum number of points with the appropriate area.

In order to match edge i to edge j, planar linear homogeneous transformations including translation, rotation, and fixed scale are used. From Figure 3.12 it is easy to see the steps needed for the transformation. Consider points p_1 and p_2, which define edge i. Their coordinates are respectively (x_1, y_1) and (x_2, y_2). Four transformations are necessary for the matching. First, edge i is translated to the origin by translating p_1 to the origin. This will cause p_2 to be transformed to a new point, p_2'. Next, edge i is scaled to correspond to the dimensions used in obtaining edge j to relate the scale of the tactile sensor to the scale of the model. Then, edge i is rotated by θ radians by rotating p_2' by θ radians. Finally, edge i is translated to edge j by translating the origin (where p_1 had been translated) to the point (x_1', y_1'). The transformation T is given by:

$$T = \text{Trans}(x_1', y_1')\text{Rot}(z, \theta)\text{Scale}(\frac{1}{RATIO})\text{Trans}(-x_1, -y_1),$$

where

$$Rot(z,\theta) = \begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

$$Trans(t_x, t_y) = \begin{vmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{vmatrix}$$

$$Scale(\frac{1}{RATIO}) = \begin{vmatrix} \frac{1}{RATIO} & 0 & 0 \\ 0 & \frac{1}{RATIO} & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

and RATIO equals the ratio of the pixel dimension to the model dimension.

Transformation T matches edge i to edge j with points (x_1, y_1) and (x_1', y_1') matching. Likewise,

$$T' = \text{Trans}(x_2', y_2')\text{Rot}(z,\theta)\text{Scale}(\frac{1}{RATIO})\text{Trans}(-x_2, -y_2)$$

matches edge i to edge j with points (x_2, y_2) matching to point (x_2', y_2'). In addition,

$$T'' = \text{Trans}(x_1', y_1')\text{Rot}(z,\pi-\theta)\text{Scale}(\frac{1}{RATIO})\text{Trans}(-x_2, -y_2)$$

$$T''' = \text{Trans}(x_2', y_2')\text{Rot}(z,\pi-\theta)\text{Scale}(\frac{1}{RATIO})\text{Trans}(-x_1, -y_1)$$

match the opposite orientations.

The main difference between this algorithm and that of [7] is that the optimal transformation is not computed. The criterion of a good match is the number of points matched; the actual transformation matching the points is not needed. Also, the ordered edges of the boundary of the convex hull of each set is used to determine the transformation. Complete graphs of the convex hull boundaries are

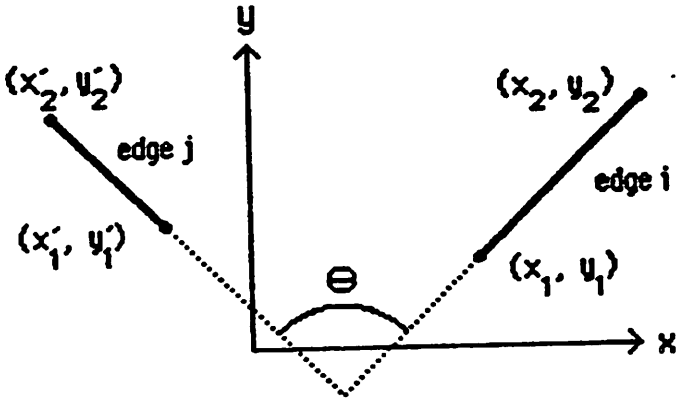


Figure 3.12 Transforming Footprint Edges

not used.

There are two advantages to using this algorithm. First, errors in pointwise matches can be taken into account by setting a threshold distance. Also, the maximum number of points will be matched as long as one edge of $\overline{D}$ correctly corresponds to an edge in one of the $\overline{FP_i}^k$. Second, the ordered edges of each $\overline{FP_i}^k$ are automatically generated in the process of the gift-wrapping method, so no additional computations are necessary. The ordered edges of $\overline{D}$ can be obtained using the same procedure.

3.4.3. Test Results

The recognition scheme was tested and good results were obtained. Typical objects used are shown if Figure 3.13. These sample objects were chosen to illustrate the operation of the recognition scheme even though they are relatively uncomplicated. Sample footprints used included isolated points and edges.

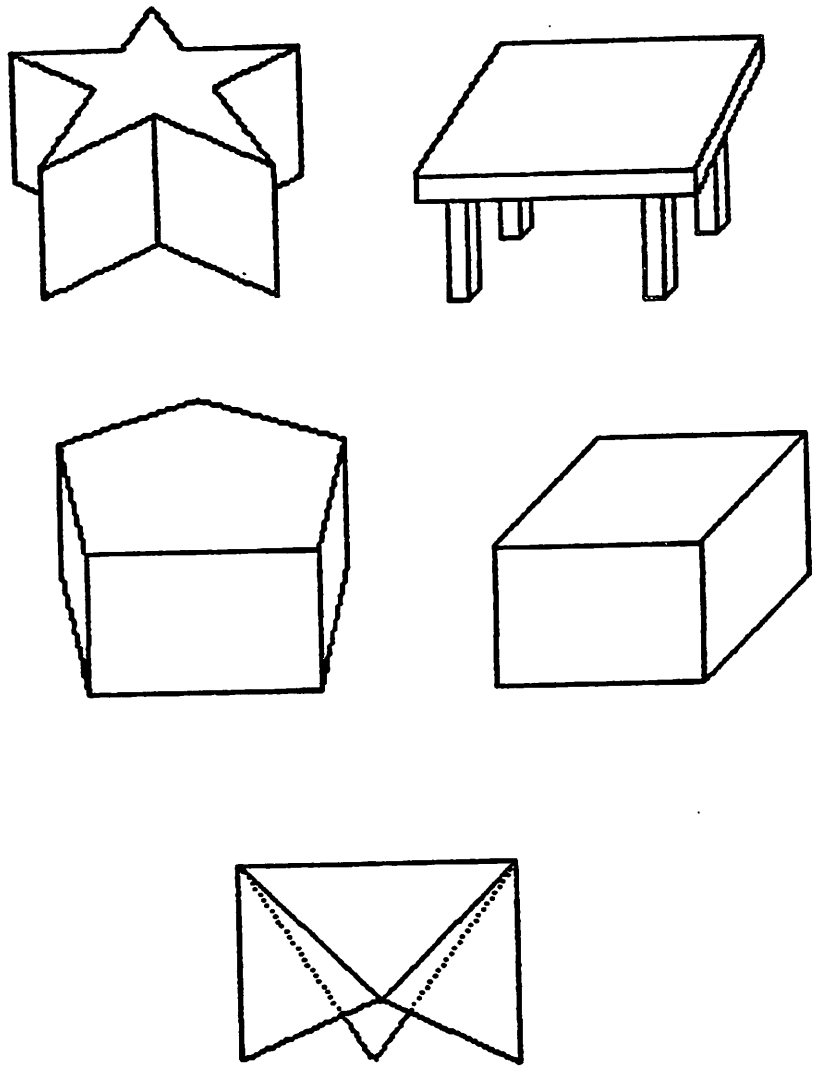


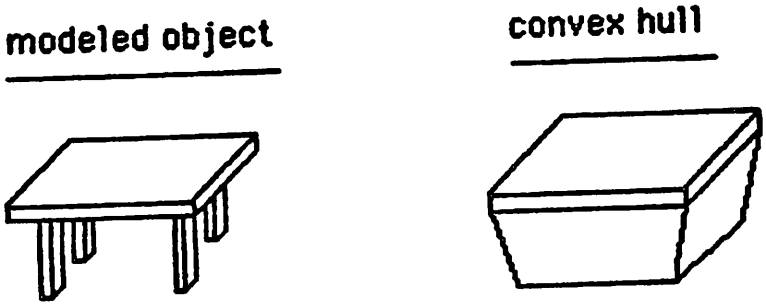
Figure 3.13 Sample Objects Used

In all cases, an *approximate* tactile footprint was obtained by thresholding a grey level image of a two-dimensional object. This approximation was necessary since no tactile pad was available. The resolution used was 80×80, although smaller resolutions would work also.

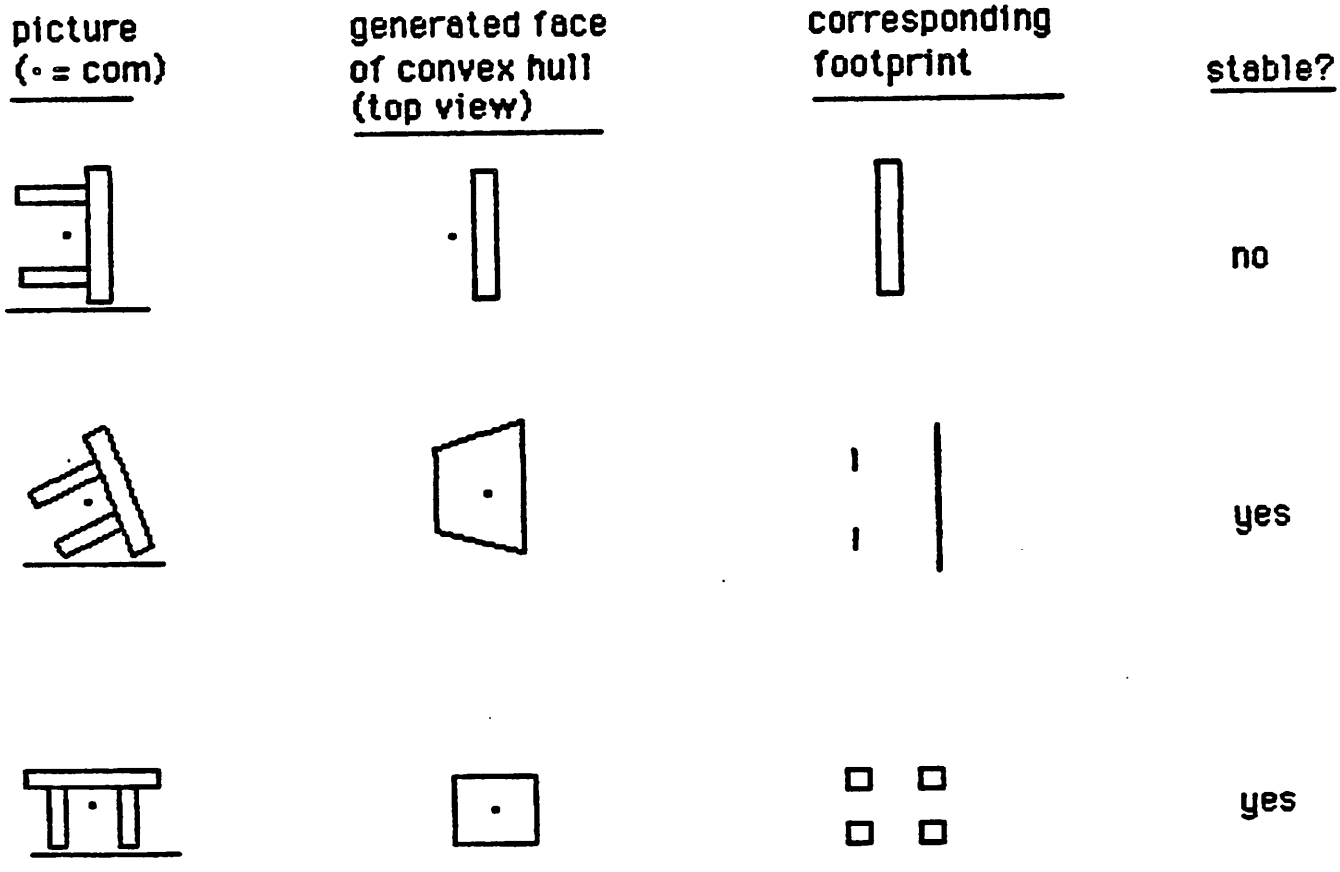
The recognition software was run on a time shared VAX 11/750 computer. With the five objects of Figure 3.13, the generation of all stable footprints and recognition with one measured footprint was completed in times ranging from 32 seconds using the simplest measured footprint to 120 seconds using the most complicated measured footprint.

Figures 3.11, 3.14, and 3.15 illustrate some of the steps that the recognition scheme takes in determining a match. Figure 3.14 shows the basic operation of the gift-wrapping method for the case of the modeled table. Figure 3.14 (c) shows some of the generated faces of the convex hull of the table shown in Figure 3.14 (a) as well as their corresponding footprints.

Figure 3.15 (a) shows a sample measured footprint of one face of the three-dimensional star. Figure 3.15 (b) shows the corresponding detected corners. Note that one spurious corner was detected incorrectly. Although an incorrectly detected corner presents a potential threat of incorrectly matching to the corner of a different object and possibly leading to a wrong recognition, this additional corner did not interfere with the recognition scheme for the objects used. In addition, the additional corner did not alter the matching scheme since it is contained in the interior of the convex hull of the footprint.

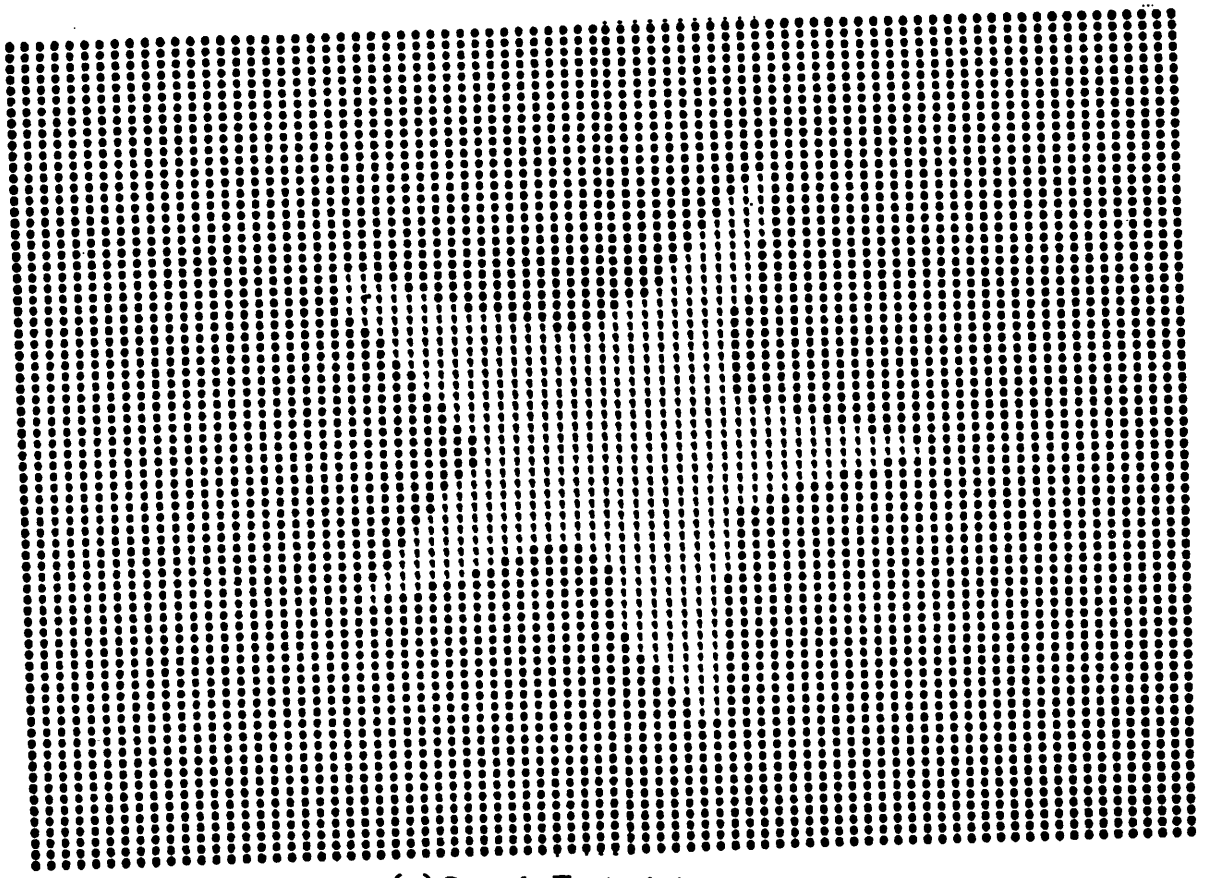


(a) Table (b) Convex Hull



(c) Sample Generated Faces

Figure 3.14 Sample Steps Taken in the Gift-Wrapping Method



(a) Sample Footprint

(b) Detected Corners

Figure 3.15 Sample Measured Footprint and Detected Corners

The sizes of the objects used were chosen to present ambiguities to the recognition scheme. Although the convex hull boundaries of the three-dimensional star and pentagon are the same, recognition was obtained given a measured footprint of the star as data. In this case, the greatest number of corners was matched to the star. Given a measured footprint of the pentagon (the pentagon face), the recognition was also obtained in one cycle. Even though the same number of corners was matched to the three-dimensional star and pentagon, the area constraint of the matching scheme ruled out the possibility of the star. The faces of the box were chosen to be equivalent to faces of the pentagon, table, and the convex hull of faces of the star. The recognition scheme behaved as expected, being able to distinguish between footprints with equivalent convex hull boundaries and different areas, and requiring additional measurements in the cases where the areas were equivalent.

The computational efficiency of the gift-wrapping method is reported by Preparata and Shamos [8]. The time needed for computing all faces of a polytope in R^3 grows as n^2, where n is the number of points of the polytope. The times of the test results supported this analysis. Note that the time used to generate all stable footprints of a modeled object is a one-time expense. Once generated, the footprints are stored.

The time required for the matching scheme, however, is not a one-time expense. Although the run time is dependent on the number of points in each footprint being matched as well as the number of points on the convex hull boundary of each footprint, the majority of time is spent in transforming a point from one footprint to the other. The average time required to match a point in one set to a point in another set was measured as approximately 3.6 milliseconds. In the matching algorithm, the time required to match one footprint to another grows

approximately as $\bar{n}\bar{m}n$, where $\bar{n}$ and $\bar{m}$ are the number of points on the convex hull boundary of the each footprint and n is the number of points of the measured footprint.

4. Future Work

Tactile data is used in many applications. The recognition scheme presented in this paper utilized two-dimensional binary data to recognize an object from a set of objects. Future projects, however, may wish to exploit grey level sensing. The footprint, in this case, not only gives positional information, but also gives force information. Possible recognition applications include inspection through feature extraction, center of mass analysis through the use of multiple footprints, and medical applications through the analysis of the force distribution of an actual footprint.

Future developments in tactile sensors will lead to even more possibilities. Present research is involved in giving future sensors a skin-like sensation. Articulated hands will then have human-like properties which will permit greater dexterity. Possible applications include texture detection, automated assembly, and grasping applications. A detailed survey of future trends of tactile sensing is described by Harmon [12].

References

[1] Okada, T. and Tsuchiya, S., "Object Recognition by Grasping," *Pattern Recognition*, vol. 9, no. 3, 1977, pp. 111-119.

[2] Briot, M., Renaud, M., and Stojilkovic, Z., "An Approach to Spatial Pattern Recognition of Solid Objects," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-8, Sept. 1978, pp. 690-694.

[3] Gaston, P. and Lozano-Perez, T., "Tactile Recognition and Localization Using Object Models: The Case of Polyhedra on a Plane," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 3, May 1984, pp. 257-266.

[4] Gonzalez, R. and Safabakhsh, R. "Computer Vision Techniques for Industrial Inspection and Robot Control: A Tutorial," *Tutorial on Robotics* Lee, C., Gonzalez, R., and Fu, K. (eds.), IEEE Computer Society Press: Maryland, 1983, pp. 300-324.

[5] Simon, J., Checroun, A., and Roche, C., "A Method of Comparing Two Patterns Independent of Possible Transformations and Small Distortions," *Pattern Recognition*, vol. 4, 1972, pp. 73-81.

[6] Baird, H. and Steiglitz, K., "A Linear Programming Approach to Noisy Template Matching," *IEEE 1982 Pattern Recognition and Image Processing Conference*, 1982, pp. 50-57.

[7] Goshtasby, A. and Stockman, G., "Point Pattern Matching Using Convex Hull Edges," *IEEE Transactions on Systems, Man, and Cybernetics*, vol SMC-15, Sept. 1985, pp. 631-637.

[8] Chand, D. and Kapur, S., "An algorithm for Convex Polytopes," *Journal of the Association for Computing Machinery*, vol. 17, no. 1, Jan. 1970, pp. 78-86.

[9] Preparata, F., Shamos, M., *Computational Geometry An Introduction*, Springer-Verlag New York Inc.: New York, 1985, pp. 125-130.

[10] Wolfe, P., "Finding the Nearest Point in a Polytope," *Mathematical Programming*, 11, 1976, pp. 128-149.

[11] Wilf, J. "Chain-Code," *Robotics Age: In the Beginning* Helms, C. (ed.), Hayden Book Co.: New Jersey, 1983, pp. 143-149.

[12] Harmon, L., "Automated Tactile Sensing," *International Journal of Robotics Research*, vol. 1, no. 2, Summer 1982, pp. 3-32.

[13] Hauser, J., "Proximity Algorithms: Application and Implementation," Master's Project, EECS Department, University of California, Berkeley, 1986.

Appendix A. Recognition Scheme Implementation

This appendix describes how the recognition scheme were implemented in the form of a computer code. These programs are written in standard C, and are documented to allow future projects to be able to use and modify the code as necessary. The code is organized into five modules.

Face.c contains the code that generates and stores points of all stable footprints of an object and their convex hull boundary points. The objects geometry is input from files named "aux1", "aux2", ..., "aux$n$", where n is the number of polytopes needed to model the object. The number n, as well as the object number i, are entered interactively from a terminal. The ordered points of all stable footprints of the object i, as well as the number of faces stored, the number of points in each face, and the face's area is stored in a file called "obj$i$.convex". All points of all stable footprints of the object i, as well as the number of points in each face is stored in a file called "obj$i$.all".

Corner.c contains the code for the edge and corner detection. A two-dimensional binary array of data values is input, and a list of all corners as well as an ordered list of the corners of the convex hull boundary is output. The total number of points in each list is also output. This function is used in Recog.c.

Recog.c contains the code for the template matching scheme. The number of objects, l, as well as the error threshold distance number are entered interactively from a terminal. The generated model footprints are input from the files "obj$k$.convex" and "obj$k$.all" for $k = 1, 2, ..., l$.

Vector.c contains routines to perform matrix and vector manipulations. These routines are used in Face.c, Corner.c, and Recog.c.

Wolfe.c contains the code to implement the Wolfe Algorithm. This code was obtained from John Hauser [13]. This routine is used in Face.c, Corner.c, and Recog.c.

In addition to the five modules above, there are two small header files defining parameters and common declarations. The file *header.h* contains some macros and declarations for standard library functions. The file *include.h* contains some global constants used in Face.c and Recog.c.

```
/*                          Face.c                                          */
/*                                                                          */
/*      This program determines the stable footprints of an object.         */
/*      The object is modeled as a union of polytopes and all possible      */
/*      footprints of the object are obtained by finding the faces of       */
/*      resulting polytope.  The "giftwrapping" method (in R3) is used      */
/*      to compute these faces, i.e. a given face is rotated around a       */
/*      given edge until it corresponds to the adjacent face.               */

                                    /* storage for all point matchings      */
#define TOP (((MAX -1)*MAX)/2)      /* 1+2+..+(MAX-1) = (MAX-1)*MAX/2       */
#define EPSILON 1.0e-5              /* error range */
#define EMPTY -1                    /* empty flag  */

#include <stdio.h>
#include "constants.h"
#include "header.h"

                /* structure for storing all points of the model            */
struct pointdata1 {
          int partno, noncorner;
          float coord[3];
};
                                    /* structure for storing points of the  */
                                    /* convex hull of the model             */

struct pointdata2 {
          int partno;
          float coord[3];
};
                                          /* structure for storing faces    */
                                          /* and edges found                */

struct edgedata {
          int face;
          float coord[3];
          float intercept;
};
                                    /* structure for storing the c.o.m.     */
                                    /* and total mass of each polytope      */
                                    /* used to model the object             */

struct inputdata {
          float cofm[3];
          float mass;
};



                                  /* storage for all points of the model    */
struct pointdata1  points[MAX];
struct pointdata2  cpoints[MAX];    /* storage for points on the convex     */
                                    /* hull boundry (CHB) of the object     */

struct edgedata  edgecheck[TOP];          /* storage for each found edge    */
                                          /* and corresponding faces        */

int auxnum, objnum;       /* number of polytopes used to model the object   */
                          /* and the object number                          */

int facepts[MAX-1], numfacepts, numfaces; /* indices of the points on a     */
                                          /* face, the number of points     */
                                          /* and the number of faces        */

FILE *fopen(), *fptemp, *fpall;  /* file pointers to files off all face     */
                                 /* points and those on the CHB             */

float inner(), norm(), dist();



main()

{
```

```
                                /* indices of points of an edge, and  */
                                /* the total number of CHB points     */
    int pt1, pt2, total;        /* c.o.m. of the object, and a normal */
    float com[3], normal[3], D; /* and intercept of a support plane   */

    Initialize ();

    InputModel (&total, com);

    FindStartData (total, &pt1, &pt2, normal, &D);

    FindFaces (pt1, pt2, total, com, normal, D);

    StorePoints ();

    printf ("\nALL DONE\n");

}



Initialize ()
                /* Initialize all global storage structures                */
{
    int i, j;

    numfaces = 0;

    for (i = 0; i < TOP; ++i) {
        edgecheck[i].face = 0;
        for (j = 0; j < 3; ++j)
            edgecheck[i].coord[j] = 0;
        edgecheck[i].intercept = 0;
    }

    for (i = 0; i < MAX; ++i) {
        points[i].partno = points[i].noncorner = cpoints[i].partno = 0;
        for (j = 0; j < 3; ++j)
            points[i].coord[j] = cpoints[i].coord[j] = 0;
    }
}



InputPoints (i, ptotal1, pmass, com, samepts)

int i, *ptotal1, samepts[MAX];
float *pmass, com[3];
                /* Input points of one polytope of the object model       */
{
    FILE *fp, *fopen ();
    int j, addipts;

        if (i == 1)
            fp = fopen ("aux1", "r");
        else if (i == 2)
            fp = fopen ("aux2", "r");
        else if (i == 3)
            fp = fopen ("aux3", "r");
        else if (i == 4)
            fp = fopen ("aux4", "r");
        else if (i == 5)
            fp = fopen ("aux5", "r");
        else if (i == 6)
            fp = fopen ("aux6", "r");
        else if (i == 7)
            fp = fopen ("aux7", "r");
                                            /* input c.o.m. and total mass  */
    fscanf (fp, "%f %f %f %f", &com[0], &com[1], &com[2], pmass);
                /* input points of polytope and flag repeated points      */
    fscanf (fp, "%d", &addipts);

    while (fscanf (fp, "%f %f %f", points[*ptotal1].coord,
```

```
      points[*ptotal1].coord + 1, points[*ptotal1].coord + 2) != EOF) {
      points[*ptotal1].partno = i;
      points[*ptotal1].noncorner = ((addipts--) > 0) ? 1 : 0;
      for (j = 0; j < *ptotal1; ++j)
         if (dist (3, points[j].coord, points[*ptotal1].coord) < EPSILON)
            samepts[*ptotal1] = j;
      (*ptotal1)++;
   }

   fclose (fp);
}


InputModel (ptotal, com)

int *ptotal;
float com[3];
                  /* Input all points of the object and compute the       */
                  /* c.o.m.

{
   int i, j, k, l, total1, samepts[MAX];
   float tmass;
   struct inputdata auxfile[AUXMAX];
   double wolfep_();                  /* storage for the Wolfe algorithm   */
   double pp[3 * MAX], rr[28], x[3];
   int s[4];
                                      /* input object number and data file */
   do {
      printf ("\nEnter object number (0 to exit): ");
      scanf ("%d", &objnum);

      if (objnum == 0) {
         printf ("\nBYE\n");
         exit (0);
      }
      else if ((objnum > OBJMAX) || (objnum < 1))
         printf ("Maximum number is %d.", OBJMAX);
   } while ((objnum > OBJMAX) || (objnum < 1));

   if (objnum == 1)
      fpall = fopen ("obj1.all", "w");
   else if (objnum == 2)
      fpall = fopen ("obj2.all", "w");
   else if (objnum == 3)
      fpall = fopen ("obj3.all", "w");
   else if (objnum == 4)
      fpall = fopen ("obj4.all", "w");
   else if (objnum == 5)
      fpall = fopen ("obj5.all", "w");
   else if (objnum == 6)
      fpall = fopen ("obj6.all", "w");
   else if (objnum == 7)
      fpall = fopen ("obj7.all", "w");
   else if (objnum == 8)
      fpall = fopen ("obj8.all", "w");
   else if (objnum == 9)
      fpall = fopen ("obj9.all", "w");
   else if (objnum == 10)
      fpall = fopen ("obj10.all", "w");

   fptemp = fopen ("temp", "w");
                                      /* input number of files needed for model */
   do {
      printf ("\nEnter number of convex polytopes to be unioned: ");
      scanf ("%d", &auxnum);

      if ((auxnum > AUXMAX) || (auxnum < 1))
         printf ("Maximum number is %d.", AUXMAX);
   } while ((auxnum > AUXMAX) || (auxnum < 1));
                                                  /* input model points   */
   total1 = 0;
   for (j = 0; j < MAX; ++j)
```

```
    samepts[j] = EMPTY;
  for (i = 0; i < auxnum; ++i)
    InputPoints (i + 1, &total1, &auxfile[i].mass, &auxfile[i].cofm[0], samepts);
                                                /* compute total mass    */
  tmass = 0;
  for (i = 0; i < auxnum; ++i)
    tmass += auxfile[i].mass;
                                                /* compute c.o.m.        */
  for (i = 0; i < 3; ++i) {
    com[i] = 0;
    for (j = 0; j < auxnum; ++j)
      com[i] += auxfile[j].cofm[i] * auxfile[j].mass;
    com[i] /= tmass;
  }
                                              /* determine convex hull   */
  *ptotal = 0;
  for (i = 0; i < total1; ++i) {
    k = 0;
    for (j = 0; j < total1; ++j)
      if ((j != i) && (samepts[j] == EMPTY))
        for (l = 0; l < 3; ++l)
          pp[k++] = points[j].coord[l] - points[i].coord[l];

    j = 3;
    k /= 3;
    if (wolfep_(pp, &j, &k, x, s, rr) > EPSILON) {
      cpoints[*ptotal].partno = points[i].partno;
      for (l = 0; l < 3; ++l)
        cpoints[*ptotal].coord[l] = points[i].coord[l];
      (*ptotal)++;
    }
  }
}



f (i, j)

int i, j;
                /* Convert indices of an upper diagnol matrix to an      */
                /* index for an equivalent one-dimensional array         */
{
  int index;

  index = (int) (j - i + i*(MAX - (i + 1)/2) - 1 + 0.5);
  return (index);
}



PosDir (P, e, total)

int P[MAX], total;
float e[3];
                /* Orient vector e in the direction of the points with   */
                /* indices not in P
{
  int i, j, found;
  float vk[3];

  i = 0;
  found = 1;
  while (found) {                   /* find index of point not in plane  */
    found = 0;
    for (j = 0; j < total; ++j)
      if (P[j] == i)
        found = 1;
    if (!found) {
      diff (3, cpoints[i].coord, cpoints[P[0]].coord, vk);
      if (!inner (3, e, vk))
        found = 1;
      else found = 0;
    }
    if (found)
```

```
        ++i;
    }                                           /* make <e,vk> >= 0      */
    pos (e, vk);

}


MaxAngle (i, v, normal, e, P, pj, start, pmax)

int i, P[MAX], *pj, start;
float v[3], normal[3], e[3], *pmax;
                /* Determine the cotangent of the angle between the     */
                /* normal and a hyperplane.  Store the indices of the   */
                /* points which make the largest angle in P             */
{
    int k;
    float ratio;
                                        /* determine the cotangent      */
    ratio = -( inner (3, e, v) / inner (3, normal, v));
                                    /* if maximum, store the index in P */
    if (P[start] == EMPTY) {
        P[start] = i;
        *pj = start + 1;
        *pmax = ratio;
    }
    else if ((abs (ratio - *pmax)) < EPSILON)
        P[(*pj)++] = i;
    else if (ratio > *pmax) {
        for (k = start; k < MAX; ++k)
            P[k] = EMPTY;
        P[start] = i;
        *pj = start + 1;
        *pmax = ratio;
    }
}


FindStartData (total, ppt1, ppt2, n, pD)

int total, *ppt1, *ppt2;
float n[3], *pD;
                /* Determine an initial edge and face to start the      */
                /* algorithm

{
    int i, j, k;
    float e[3], v[3], vk[3];
    int P[MAX];
    float max, min, angle;

    for (i = 0; i < total; ++i)
        P[i] = EMPTY;
    j = 0;
    min = cpoints[0].coord[0];
                                        /* find pts with min x value    */
    for (i = 0; i < total; ++i)
        if (cpoints[i].coord[0] < min) {
            for (k = 0; k < total; ++k)
                P[k] = EMPTY;                           /* store min pt */
            P[0] = i;                   /* j = # of pts with min x value */
            j = 1;
            min = cpoints[i].coord[0];
        }
        else if (cpoints[i].coord[0] == min)
            P[j++] = i;

    n[0] = 1;
    n[1] = n[2] = 0;

    if (j == 1) {
        e[0] = e[2] = 0;
```

```
    e[1] = 1;                                   /* make <e, vk> >= 0    */
    PosDir (P, e, total);

    for (i = 0; i < total; ++i)
        if (i != P[0]) {
            diff (3, cpoints[i].coord, cpoints[P[0]].coord, v);
            MaxAngle (i, v, n, e, P, &j, 1, &max);
        }                                       /* update the normal    */
    for (i = 0; i < 3; ++i)
        n[i] = n[i] * max + e[i];
    normalize (3, n);
}

if (j == 2) {
    diff (3, cpoints[P[1]].coord, cpoints[P[0]].coord, v);
    cross (v, n, e);                            /* make <e, vk> >= 0    */
    PosDir (P, e, total);

    for (i = 0; i < total; ++i)
        if ((i != P[0]) && (i != P[1])) {
            diff (3, cpoints[i].coord, cpoints[P[0]].coord, v);
            MaxAngle (i, v, n, e, P, &j, 2, &max);
        }                                       /* update the normal    */
    for (i = 0; i < 3; ++i)
        n[i] = n[i] * max + e[i];
    normalize (3, n);
}

if (j >= 3) {
    diff (3, cpoints[P[1]].coord, cpoints[P[0]].coord, v);
    diff (3, cpoints[P[2]].coord, cpoints[P[0]].coord, vk);
                                            /* compute normal of plane  */
    cross (v, vk, n);                       /* make <n, vk> >= 0        */
    PosDir (P, n, total);

    *pD = - inner (3, n, cpoints[P[0]].coord);
    *ppt1 = P[0];
                                            /* lowest possible value = -1 */
    max = -2;                               /* determine the second point of the edge */
    for (i = 2; i < j; ++i) {
        diff (3, cpoints[P[i]].coord, cpoints[P[0]].coord, vk);
        angle = - inner (3, v, vk);
        if (angle > max) {
            max = angle;
            *ppt2 = P[i];
        }
    }
}
}


Face (total, normal, D, P)

int total;
int P[MAX];
float normal[3], D;
                /* Determine the adjacent face of the CHB.  Store the   */
                /* the indices of the face points in P                  */
{
    float edge[3], e[3], v[3];
    int i, j;
    float maximum;

    j = 3;
                /* compute the unit vector in the direction of the edge */
    diff (3, cpoints[P[1]].coord, cpoints[P[0]].coord, edge);
```

```
                        /* determine the third orthogonal vector e      */
    cross (edge, normal, e);

    i = 0;
    while ((iplane (normal, cpoints[i].coord, D)) || (i == P[0]) || (i == P[1]))
        ++i;
    diff (3, cpoints[i].coord, cpoints[P[0]].coord, v);
    pos (e, v);
                                /* determine the points of the plane    */
    for (i = 0; i < total; ++i)
        if (iplane (normal, cpoints[i].coord, D)) {
            diff (3, cpoints[i].coord, cpoints[P[0]].coord, v);
            MaxAngle (i, v, normal, e, P, &j, 2, &maximum);
        }
}


UpdateEdgeCheck (a, b, normal, D)

int a, b;
float normal[3], D;
                    /* Update the faces and corresponding edges found   */
{
    int i, index;

    if (a < b)
        index = f(a, b);
    else index = f(b, a);

    edgecheck[index].face += 1;
                                    /* stop if an incorrect edge was    */
                                    /* computed due to numerical errors */
    if (edgecheck[index].face > 2) {
        printf ("\n\nError in determining second face.");
        printf ("\nChange the value of EPSILON appropriately and recompile.");
        exit (0);
    }

    for (i = 0; i < 3; ++i)
        edgecheck[index].coord[i] = normal[i];
    edgecheck[index].intercept = D;
}


Edges (normal, pD, total, P)

int total, P[MAX];
float normal[3], *pD;                                                   */
                    /* Determine the edges of the face

{
    int i, j, a, b, c;
    float x[3], y[3];
    float angle, maximum;

    diff (3, cpoints[P[1]].coord, cpoints[P[0]].coord, x);
    diff (3, cpoints[P[2]].coord, cpoints[P[0]].coord, y);
    cross (x, y, normal);                   /* compute normal of new plane  */

    PosDir (P, normal, total);              /* make <normal, vk> >= 0       */

                                    /* compute intercept of new plane   */
    *pD = - inner (3, normal, cpoints[P[0]].coord);
                                            /* update edgecheck         */
    UpdateEdgeCheck (P[0], P[1], normal, *pD);
                                            /* compute and store all edges  */
    a = P[0];
    b = P[1];
```

```
    facepts[0] = a;
    numfacepts = 1;
    do {
        facepts[numfacepts++] = b;                          /* compute all edges    */
        diff (3, cpoints[a].coord, cpoints[b].coord, x);
        i = 0;                                              /* lowest possible value = -1   */
        maximum = -2;
                                                            /* find adjacent edge point     */
        while ((P[i] != EMPTY) && (i <= MAX)) {
            if ((P[i] != a) && (P[i] != b)) {
                diff (3, cpoints[P[i]].coord, cpoints[b].coord, y);
                angle = - inner (3, x, y);
                if (angle > maximum) {
                    maximum = angle;
                    c = P[i];
                }
            }
            ++i;
        }
        UpdateEdgeCheck (b, c, normal, *pD);

        a = b;
        b = c;
    } while (b != P[0]);
}



PrintPlane (normal, D)

float normal[3], D;
                /* Print the corresponding footprint of a face of the   */
                /* CHB. Call StoreData to store the footprint in a file */
{
    int i, j, numpts, AP[MAX], numdiffpts, diffpts[MAX], same;

    numpts = numdiffpts = 0;
    i = 0;                                  /* determine all points in the plane    */
    while ((points[i].partno != 0) && (i < MAX)) {
        if (plane (normal, points[i].coord, D)) {
            same = 0;
            for (j = 0; j < i-1; ++j)
                if (dist (3, points[j].coord, points[i].coord) < EPSILON)
                    same = 1;
            if (!same)
                diffpts[numdiffpts++] = i;

            AP[numpts++] = i;
        }
        ++i;
    }
                                    /* print the footprint in terms of the  */
                                    /* decomposed model parts               */
    printf ("\nFACE:\n");

    for (i = 1; i <= auxnum; ++i)
        printf ("%15d", i);
    printf ("\n\n");
    for (i = 1; i <= auxnum; ++i)
        printf ("     X     Y     Z");
    printf ("\n");

    for (i = 0; i < numpts; ++i) {
        for (j = 0; j < (points[AP[i]].partno - 1); ++j)
            printf ("                    ");
        for (j = 0; j < 3; ++j)
            printf ("%5.1f", points[AP[i]].coord[j]);
        printf ("\n");
    }
                                        /* store footprint in a file    */
```

```
   StoreData (numpts, AP, numdiffpts, diffpts);
}


StoreData (numpts, AP, numdiffpts, diffpts)
int numpts, AP[MAX], numdiffpts, diffpts[MAX];
                /* Compute the area of the footprint and store the      */
                /* points
{
   int i, j, k;
   int a, b, c, fppts[MAX], numfppts, numnoncorner;
   float x[3], y[3], z[3], angle, maximum, sin_theta, area, midpt[3];

   area = 0;
                                /* determine the area for each isolated */
                                /* part of the footprint                */
   for (i = 0; i < AUXMAX; ++i) {
      numfppts = 0;
      for (j = 0; j < numpts; ++j)
         if (points[AP[j]].partno == (i+1))
            fppts[numfppts++] = AP[j];

      if (numfppts > 2) {
         for (j = 0; j < 3; ++j) {
            midpt[j] = 0;
            for (k = 0; k < numfppts; ++k)
               midpt[j] += points[fppts[k]].coord[j];
            midpt[j] /= numfppts;
         }

         a = fppts[0];
         b = fppts[1];
         do {                                   /* compute all edges    */
            diff (3, points[a].coord, points[b].coord, x);
            maximum = -2;                       /* lowest possible value = -1   */

            for (j = 0; j < numfppts; ++j) {
               if ((fppts[j] != a) && (fppts[j] != b)) {
                  diff (3, points[fppts[j]].coord, points[b].coord, y);
                  angle = - inner (3, x, y);
                  if (angle > maximum) {
                     maximum = angle;
                     c = fppts[j];
                  }
               }
            }
                                        /* compute area of sector       */
            for (j = 0; j < 3; ++j) {
               y[j] = points[b].coord[j] - midpt[j];
               z[j] = points[c].coord[j] - midpt[j];
            }
            sin_theta = sqrt(1 - square(inner(3,y,z)/(norm(3,y) * norm(3,z))));
                                        /* update the area      */
            area += 0.5 * abs (norm (3, y) * norm (3, z) * sin_theta);

            a = b;
            b = c;
         } while (b != fppts[1]);
      }
   }
                        /* store the area, number of points on the CHB  */
                        /* of the footprint, and the points             */

   fprintf (fptemp, "%f\n", area);
   fprintf (fptemp, "%d\n", numfacepts);
   for (i = 0; i < numfacepts; ++i) {
      for (j = 0; j < 3; ++j)
         fprintf (fptemp, "%9.5f", cpoints[facepts[i]].coord[j]);
      fprintf (fptemp, "\n");
   }
   numfaces++;
```

```
                        /* determine the number of noncorner points      */
                        /* contained in the footprint

    numnoncorner = 0;
    for (i = 0; i < numdiffpts; ++i)
        if (points[diffpts[i]].noncorner)
            numnoncorner++;
                        /* store the first 3 points the same             */
                        /* (necessary for the recognition scheme)
                        /* (necessary for the recognition scheme);
    fprintf (fpall, "%d\n", numdiffpts - numnoncorner);
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < 3; ++j)
            fprintf (fpall, "%9.5f", cpoints[facepts[i]].coord[j]);
        fprintf (fpall, "\n");
    }                                   /* store the remaining points    */
    for (i = 0; i < numdiffpts; ++i)
        if ((dist (3, points[diffpts[i]].coord, cpoints[facepts[0]].coord) > EPSILON)
        && (dist (3,points[diffpts[i]].coord,cpoints[facepts[1]].coord) > EPSILON)
        && (dist (3,points[diffpts[i]].coord,cpoints[facepts[2]].coord) > EPSILON)
        && (points[diffpts[i]].noncorner == 0)) {
            for (j = 0; j < 3; ++j)
                fprintf (fpall, "%9.5f", points[diffpts[i]].coord[j]);
            fprintf (fpall, "\n");
        }
}



Constraint (P, com, normal, D)

int P[MAX];
float com[3], normal[3], D;
                        /* Determine if the footprint is stable, i.e. project   */
                        /* the c.o.m. perpendicularly down to the plane and     */
                        /* check if it is contained in the convex hull of the   */
                        /* footprint

{
    float a[3];

    int i, j, k;
                                        /* storage for Wolfe algorithm  */
    double wolfep_();
    double pp[3 * MAX], x[3], rr[28];
    int s[4];

    projectcom (com, normal, D, a);

    i = 0;
    j = 0;
    while (P[i] != EMPTY) {
        for (k = 0; k < 3; ++k)
            pp[j++] = cpoints[P[i]].coord[k] - a[k];
        ++i;
    }

    j = 3;
    if (wolfep_(pp, &j, &i, x, s, rr) < EPSILON)
        PrintPlane (normal, D);
}



FindFaces (pt1, pt2, total, com, normal, D)

int pt1, pt2, total;
float com[3], normal[3], D;
                        /* Determine all faces of the convex hull of the object */
                        /* Determine all faces of the convex hull of the object */
                        /* This is done by checking edgecheck for edges with    */
                        /* only one face found and determining the second face  */

{
    int i, j, found;
    int P[MAX];
```

```
    for (j = 0; j < MAX; ++j)
       P[j] = EMPTY;
    P[0] = pt1;
    P[1] = pt2;
    j = 2;
                                    /* determine all points in the plane    */
    for (i = 0; i < total; ++i)
       if (plane (normal, cpoints[i].coord, D))
          if ((i != pt1) && (i != pt2))
             P[j++] = i;
                                /* determine the edges of the face of the CHB   */
    Edges (normal, &D, total, P);
                                /* check the stability of the corresponding footprint   */
    Constraint (P, com, normal, D);
                                /* determine next edge with only one face found */
    do {
       i = 0;
       j = 1;
       while ((i < (total - 1)) && (edgecheck[f(i, j)].face != 1)) {
          ++j;
          if (j == total) {
             ++i;
             j = i + 1;
          }
       }
                                                        /* if none then stop    */
       if (i == (total - 1))
          found = 0;
                                        /* otherwise find the second face       */
       else {
          found = 1;
          normal = edgecheck[f(i, j)].coord;
          D = edgecheck[f(i, j)].intercept;
          P[0] = i;
          P[1] = j;
          for (j = 2; j < MAX; ++j)
             P[j] = EMPTY;

          Face (total, normal, D, P);
          Edges (normal, &D, total, P);
          Constraint (P, com, normal, D);
       }

    } while (found);
}



StorePoints ()
                        /* Store the footprint points and area data in files    */
{
    int i, j, k;
    float pt, area;
    FILE *fp;

    fclose (fpall);
                        /* open file to store the CHB points of each footprint  */
    if (objnum == 1)
       fp = fopen ("obj1.convex", "w");
    else if (objnum == 2)
       fp = fopen ("obj2.convex", "w");
    else if (objnum == 3)
       fp = fopen ("obj3.convex", "w");
    else if (objnum == 4)
       fp = fopen ("obj4.convex", "w");
    else if (objnum == 5)
       fp = fopen ("obj5.convex", "w");
    else if (objnum == 6)
       fp = fopen ("obj6.convex", "w");
```

```
    else if (objnum == 7)
        fp = fopen ("obj7.convex", "w");
    else if (objnum == 8)
        fp = fopen ("obj8.convex", "w");
    else if (objnum == 9)
        fp = fopen ("obj9.convex", "w");
    else if (objnum == 10)
        fp = fopen ("obj10.convex", "w");

    fclose (fptemp);
                            /* rewrite data stored in "temp", but include   */
                            /* the total number of faces at the beginning   */
    fptemp = fopen ("temp", "r");

    fprintf (fp, "%d\n", numfaces);

    for (i = 0; i < numfaces; ++i) {
        fscanf (fptemp, "%f", &area);
        fprintf (fp, "%f\n", area);
        fscanf (fptemp, "%d", &numfacepts);
        fprintf (fp, "%d\n", numfacepts);
        for (j = 0; j < numfacepts; ++j) {
            for (k = 0; k < 3; ++k) {
                fscanf (fptemp, "%f", &pt);
                fprintf (fp, "%9.5f", pt);
            }
            fprintf (fp, "\n");
        }
    }
    fclose (fptemp);
    fclose (fp);
    unlink ("temp");
}
```

```
                                                                    */
                                                                    */
                        Corner.c                                    */
/*                                                                  */
/*      This procedure obtains the corners of a measured footprint. */
/*      The footprint is assumed to be in the form of a two-        */
/*      dimensional array of binary values.  Value 0 corresponds    */
/*      to the background and 1 corresponds to the footprint.  The  */
/*      corner points are obtained by traversing each isolated part */
/*      of the footprint and noting the points where a significant  */
/*      direction change occurs.  Once the corners are obtained,    */
/*      the corners on the convex hull of the boundry (CHB) of the  */
/*      footprint are obtained and ordered.  Both sets of points are*/
/*      stored in arrays passed from the calling program.           */

#define RES 80                  /* resolution of the binary footprint   */
#define PLENGTH 1               /* minimum number of pixels along any   */
                                /* straight subsection of an edge       */
#define EPSILON2 1.0e-6         /* error range for wolfe alg. output    */
#define EMPTY -1                /* empty flag                           */
#define EDGE 2                  /* pixel value flag for an edge point   */
#define CORNER 9                /* pixel value flag for a corner point  */

#include <stdio.h>              /* file containing global constants     */
#include "constants.h"          /* file containing macro definitions    */
#include "header.h"

float inner();



InputData (fpnum, dpts, pnumdpts, cdpts, pnumcdpts, parea)

int fpnum, dpts[AUXMAX * MAX][2], *pnumdpts, cdpts[MAX][2], *pnumcdpts;
float *parea;
                /* determine all corner points of the measured data     */
                /* footprint as well as all corners on the CHB          */
{
    int data[RES][RES];  /* binary array containing measured footprint  */
    int edge[RES][RES];  /* binary array containing edges of footprint  */
                                        /* input measured footprint     */
    InputPadData (fpnum, data, parea);
                                        /* detect edges of measured footprint */
    EDetect (data, edge);
                                        /* detect corners of measured footprint */
    CDetect (edge, dpts, pnumdpts, parea);
                                        /* detect CHB corners of the footprint */
    ConvexBoundry (dpts, *pnumdpts, cdpts, pnumcdpts);
                                        /* order the CHB corners in adjacent order */
    OrderCorners (cdpts, *pnumcdpts);
}



InputPadData (fpnum, data, parea)

int fpnum, data[RES][RES];
float *parea;
                /* Input the measured footprint, printing the number    */
                /* of points input
{
    FILE *fp, *fopen ();
    int i, j, k;
                                                /* open the data file   */

    if (fpnum == 0)
        fp = fopen ("data1", "r");
    else if (fpnum == 1)
        fp = fopen ("data2", "r");
```

```
    else if (fpnum == 2)
        fp = fopen ("data3", "r");
    else if (fpnum == 3)
        fp = fopen ("data4", "r");
    else if (fpnum == 4)
        fp = fopen ("data4", "r");
    else if (fpnum == 5)
        fp = fopen ("data5", "r");
    else if (fpnum == 6)
        fp = fopen ("data6", "r");
    else if (fpnum == 7)
        fp = fopen ("data7", "r");
    else if (fpnum == 8)
        fp = fopen ("data8", "r");
    else if (fpnum == 9)
        fp = fopen ("data9", "r");
    else if (fpnum == 10)
        fp = fopen ("data10", "r");
                        /* input points and determine number of 1 pixels */
    *parea = 0.0;
    k = 0;
    for (i = 0; i < RES; ++i)
        for (j = 0; j < RES; ++j) {
            k += fscanf (fp, "%1d", &data[i][j]);
            if ((i > 0) && (i < RES - 1) && (j > 0)
                && (j < RES - 1) && (data[i][j] == 1))
                *parea = *parea + 1.0;
        }

    printf ("\n%1d", k);
    printf (" points input\n");

    fclose (fp);
}



Neighbor (i, j, data)
int i, j, data[RES][RES];
                /* Return the number of adjacent pixels with value 1    */
{
    int count;

    count = 0;

    if (data[i-1][j-1])
        ++count;
    if (data[i-1][j])
        ++count;
    if (data[i-1][j+1])
        ++count;
    if (data[i][j-1])
        ++count;
    if (data[i][j+1])
        ++count;
    if (data[i+1][j-1])
        ++count;
    if (data[i+1][j])
        ++count;
    if (data[i+1][j+1])
        ++count;

    return (count);
}



Mask (i, j, count, data, edge)
int i, j, count, data[RES][RES], edge[RES][RES];
                /* Mask the data value to obtain the footprint edges    */
{
```

```
    switch (count) {
        case 0:
        case 8:
            edge[i][j] = 0;
            break;
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
            edge[i][j] = data[i][j];
            break;
    }
}


EDetect (data, edge)
int data[RES][RES], edge[RES][RES];
                    /* Detect the edges of the footprint and print result   */
{
    int i, j;

    for (i = 1; i < (RES - 1); ++i)
        for (j = 1; j < (RES - 1); ++j)
            Mask (i, j, Neighbor (i, j, data), data, edge);

    printf ("\ndetected edges\n");
    for (i = 1; i < (RES - 1); ++i) {
        for (j = 1; j < (RES - 1); ++j)
            printf ("%1d", edge[i][j]);
        printf ("\n");
    }
    printf ("\n");

}


Traverse (a, b, dir, pedgelength, mindist, edge, pindex, dpts)
int a, b, dir, *pedgelength, edge[RES][RES], *pindex, dpts[AUXMAX * MAX][2];
float mindist;
                    /* Traverse the boundry of the footprint (or isolated   */
                    /* part of the footprint) to obtain the corner points.  */
                    /* The corner points are obtained by noting when the    */
                    /* direction of the boundry changes by more than one    */
                    /* or when a change of one unit occurs and the original */
                    /* direction is not maintained for PLENGTH pixels.      */
{
    int i, j, k, l, dir1;
    int itemp, jtemp, ktemp, ltemp, dirtemp;
    int dirfound, count;
    int lastcorner[2];

    k = a;
    l = b;
    count = 0;
    dir1 = dirtemp = dir;
    ktemp = lastcorner[0] = a;
    ltemp = lastcorner[1] = b;

    do {
        dir1 = (dir1 + 5)%8;  /* start at direction furthest c-clockwise */
        dirfound = 0;
                              /* determine position of next pt. on boundry */
        do {
            switch (dir1) {
                case 0:
                    i = k - 1;
                    j = l;
```

```
            break;
         case 1:
            i = k - 1;
            j = l + 1;
            break;
         case 2:
            i = k;
            j = l + 1;
            break;
         case 3:
            i = k + 1;
            j = l + 1;
            break;
         case 4:
            i = k + 1;
            j = l;
            break;
         case 5:
            i = k + 1;
            j = l - 1;
            break;
         case 6:
            i = k;
            j = l - 1;
            break;
         case 7:
            i = k - 1;
            j = l - 1;
            break;
      }

      if (edge[i][j] == 0)
         dir1 = (dir1 + 1)%8;
      else dirfound = 1;
   } while (!dirfound);

   switch (abs(dir1 - dir)) {                    /* directions match      */
      case 0:
         edge[k][l] = EDGE;
         (*pedgelength)++;
         k = i;
         l = j;
                        /* if a direction change of one unit occured     */
                        /* recently, then check to see if PLENGTH        */
                        /* pixels with the original dir. have passed     */
         if ((count < PLENGTH) && (count > 0))
            ++count;
         else count = 0;
         break;                              /* if directions are adjacent */
      case 1:
      case 7:                                          /* a possible corner */
         if (count++ == 0) {
            edge[ktemp = k][ltemp = l] = EDGE;
            (*pedgelength)++;
            dirtemp = dir1;
                                             /* store point of poss. corner */

            k = itemp = i;
            l = jtemp = j;
         }                          /* past three directions are different */
                                                       /* a corner exists   */
         else {
            if (sqrt((double) square(k - lastcorner[0]) +
                square(l - lastcorner[1])) > (mindist)) {
               edge[ktemp][ltemp] = CORNER;
               lastcorner[0] = ktemp;
               lastcorner[1] = itemp;
            }
                           /* if not a minimum distance from the          */
                           /* last corner

            else
               edge[ktemp][ltemp] = EDGE;

            k = itemp;
            l = jtemp;
            dir = dirtemp;
            count = 0;
         }
         break;
```

```
                        /* directions changed by more than one unit     */
                                                    /* a corner is found    */
        default:
            if (sqrt((double) square(k - lastcorner[0]) +
                square(l - lastcorner[1])) > (mindist)) {
                edge[k][l] = CORNER;
                lastcorner[0] = k;
                lastcorner[1] = l;
            }                       /* if not a minimum distance from the   */
                                    /* last corner
            else edge[k][l] = EDGE;

            (*pedgelength)++;
            dir = dir1;
            k = i;
            l = j;
            count = 0;
            break;
    }
} while ((k != a) || (l != b));
                        /* if first point is a min distance away from   */
                        /* the last corner then ...
if (sqrt((double) square(a - lastcorner[0]) +
    square(b - lastcorner[1])) > (mindist)) {
                        /* if the previous PLENGTH pixels are in the    */
                        /* same direction then a corner is found

    if (dir == 2) {
        dirfound = 1;
        --l;
        for (i = 0; i < PLENGTH; ++i) {
            if (l > 0) {
                if (edge[k+1][l--] == 0)
                    dirfound = 0;
            }
            else dirfound = 0;
        }
        edge[a][b] = ((dirfound) ? EDGE : CORNER);
    }
    else
        edge[a][b] = CORNER;
}
else
    edge[a][b] = EDGE;
                                /* delete interior "leftover" 1's on    */
                                /* the edge for this part of the f.p.

for (i = 1; i < (RES - 1); ++i)
    for (j = 1; j < (RES - 1); ++j) {
        if (edge[i][j] == 1) {
            if ((edge[i][j+1] == EDGE) || (edge[i][j+1] == CORNER))
                edge[i][j] = 0;                     /* erase interior point */

        }                                           /* erase edge */
        else if (edge[i][j] == EDGE) {
            edge[i][j] = 0;                         /* erase interior point */
            if (edge[i][j+1] == 1)
                edge[i][j+1] = 0;

        }                                           /* keep corner */
        else if (edge[i][j] == CORNER) {            /* erase interior point */
            if (edge[i][j+1] == 1)
                edge[i][j+1] = 0;                   /* store corner */
            dpts[*pindex][0] = i;
            dpts[(*pindex)++][1] = j;
        }
    }
}


CDetect (edge, dpts, pnum, parea)

int edge[RES][RES], dpts[AUXMAX * MAX][2], *pnum;
float *parea;                                                       */
                /* Detect the corners of the footprint

{
```

```
    int i, j, found, dir, objects, edgelength;
    float mindist;                  /* minimum distance a corner can be     */
                                    /* from another corner                  */

    printf ("\nEnter minimum distance (in pixels) between corners: ");
    scanf ("%f", &mindist);

    *pnum = edgelength = 0;
    objects = 0;
    for (i = 0; i < (AUXMAX * MAX); ++i)
       for (j = 0; j < 2; ++j)
          dpts[i][j] = EMPTY;

    i = j = 1;
    found = 1;
    do {                                        /* find a corner of an object   */
       while ((i < (RES - 1)) && (j < (RES - 1)) && (edge[i][j] != 1)) {
          ++j;
          if (j == (RES - 1)) {
             ++i;
             j = 1;
          }
       }

       if (i == (RES - 1))
          found = 0;                            /* record isolated point        */
       else if (Neighbor (i, j, edge) == 0) {
          edge[i][j] = CORNER;
          dpts[*pnum][0] = i;
          dpts[(*pnum)++][1] = j;
       }
       else {
          *pnum = 0;
                                /* determine direction of next pt. on boundry   */
          if (edge[i][j+1])
             dir = 2;
          else if (edge[i+1][j+1])
             dir = 3;
          else if (edge[i+1][j])
             dir = 4;
          else if (edge[i+1][j-1])
             dir = 5;
                                            /* find corners of object           */
          Traverse (i, j, dir, &edgelength, mindist, edge, pnum, dpts);

          ++objects;
       }
    } while (found);
                            /* determine the number of interior 1 pixels        */
    *parea = *parea - edgelength / 2  - objects;
                                            /* print detected corners           */

    printf ("\ndetected corners\n");
    for (i = 1; i < (RES - 1); ++i) {
       for (j = 1; j < (RES - 1); ++j)
          printf ((edge[i][j] == CORNER) ? "*" : " ");
       printf ("\n");
    }
    printf ("\n");

    printf ("\n\nnumber of isolated parts: %1d", objects);

    printf ("\n\ncorner points:\n");
    for (i = 0; i < *pnum; ++i) {
       printf ("%4d %4d", dpts[i][0], dpts[i][1]);
       printf ("\n");
    }
}



ConvexBoundry (dpts, num, cdpts, pcnum)
```

```
int dpts[AUXMAX * MAX][2], num, cdpts[MAX][2], *pcnum;
                /* Determine the corners on the CHB of the footprint    */
{
   int i, j, k, l;
   double wolfep_();                          /* variables for the wolfe alg. */
   double pp[AUXMAX * MAX * 2], rr[(2+1)*(2+4)], x[2];
   int s[2 + 1];

                                              /* determine convex hull        */
   *pcnum = 0;
   for (i = 0; i < num; ++i) {                /* load points for wolfe alg.   */
      k = 0;
      for (j = 0; j < num; ++j)
         if (j != i)
            for (l = 0; l < 2; ++l)
               pp[k++] = dpts[j][l] - dpts[i][l];

      j = 2;
      k = num - 1;
      if (wolfep_(pp, &j, &k, x, s, rr) > EPSILON2) {
         for (l = 0; l < 2; ++l)
            cdpts[*pcnum][l] = dpts[i][l];
         (*pcnum)++;
      }
   }
}


OrderCorners (cdpts, num)
int cdpts[MAX][2], num;
                /* Order the points on the CHB of the footprint         */
{
   int i, j, a, b, c, index;
   int temp[MAX][2];
   float x[2], y[2], maximum, angle;

   for (i = 0; i < num; ++i)
      for (j = 0; j < 2; ++j)
         temp[i][j] = cdpts[i][j];

   a = 0;
   b = 1;
   index = 2;
   do {                                       /* compute all edges    */
      idiff (2, &temp[a][0], &temp[b][0], x);
      i = 0;
      maximum = -2;                           /* lowest possible value = -1   */
                                              /* determine adjacent corner    */
      for (i = 0; i < num; ++i) {
         if ((i != a) && (i != b)) {
            idiff (2, &temp[i][0], &temp[b][0], y);
            angle = 0;
            angle = - inner (2, x, y);
            if (angle > maximum) {
               maximum = angle;
               c = i;
            }
         }
      }                                       /* store ordered corner */

      for (j = 0; j < 2; ++j)
         cdpts[index][j] = temp[c][j];
      index++;
      a = b;
      b = c;
   } while (b != 1);
                                              /* print results        */
   printf ("\n\nordered corners of convex hull\n");
```

```
    for (i = 0; i < num; ++i) {
        printf ("%4d %4d", cdpts[i][0], cdpts[i][1]);
        printf ("\n");
    }
}
```

```
/*                                                                      */
/*                      Recog.c                                         */
/*                                                                      */
/*      This program matches a measured footprint of an object and      */
/*      compares the footprint to all possible footrints generated      */
/*      by face.c.  The matching is done by template matching.          */
/*      The corners of the measured footprint are obtained and the      */
/*      resulting set of points is compared to the sets                 */
/*      corresponding to the generated footprints.  The comparison      */
/*      is obtained by computing the transformation which matches       */
/*      an edge of the convex hull boundry (CHB) of the measured        */
/*      footprint to each edge of every CHB of the generated            */
/*      footprints.

#define EPSILON2 1.0e-1                 /* error range  */
#define EMPTY -1                        /* empty flag   */

#include <stdio.h>
#include "constants.h"                  /* file containing global constants     */
#include "header.h"

                                        /* storage for CHB points of a face     */
struct face {
          int numberpts, matchedpts;
          float coord[MAX][3], area;
          struct face *next;
};
                                        /* storage for all faces of an object   */
struct object {
          int numfaces;
          struct face *pts;
};


struct object Objects[OBJMAX];          /* structure storing all CHB    */
                                        /* points for all faces         */

int cdpts[MAX][2];                      /* storage for CHB points of    */
                                        /* the measured footprint       */

int alldpts[AUXMAX * MAX][2];           /* storage for all points of    */
                                        /* the measured footprint       */

float inner();

float dist();

char *calloc();

FILE *fptemp, *fopen ();                /* pointer to a temporary file  */
                                        /* used to store all the        */
                                        /* corners of all generated     */
                                        /* footprint                    */


float RATIO;


main ()

{
    int numobjects, numalldpts, numcdpts;
    int fpnum, possobjs[OBJMAX], npossobjs, notdone;
    float area;
                /* input all corners and CHB corners of the measured    */
                /* footprint
    fpnum = 0;
    InputData (fpnum, alldpts, &numalldpts, cdpts, &numcdpts, &area);
                /* input the CHB points for all generated footprints    */
                /* for all modeled objects
    InputFaces (&numobjects, possobjs);
```

```
                /* match footprints until recognition or end of data    */
    do {
        notdone = 1;
        Match (numobjects, numalldpts, numcdpts, possobjs, &npossobjs, area);

        if (npossobjs == 1)
            notdone = 0;
        else {
            printf ("\n\nAnother try? (1 for yes, 0 for no): ");
            scanf ("%d", &notdone);
            if (notdone)
                InputData (++fpnum, alldpts, &numalldpts, cdpts, &numcdpts, &area);
        }
    } while (notdone);
                /* remove temporary file created for storing all points */
                /* of each footprint of each modeled object             */
    CleanUp (numobjects);
    printf ("\nALL DONE\n");
}


InputFaces (pnum, possobjs)

int *pnum, possobjs[OBJMAX];
                /* Input the CHB corner points for all generated        */
                /* possible footprints
{
    int i;
                                        /* input number of objects      */
    do {
        printf ("\nEnter number of possible objects (0 to exit): ");
        scanf ("%d", pnum);

        if (*pnum == 0) {
            printf ("\nBYE\n");
            exit (0);
        }
        else if ((*pnum > OBJMAX) || (*pnum < 1))
            printf ("Maximum number is %1d.", OBJMAX);
    } while ((*pnum > OBJMAX) || (*pnum < 1));

    for (i = (*pnum + 1); i < OBJMAX; ++i)
        possobjs[i] = EMPTY;
                                        /* input object footprints      */
    for (i = 0; i < *pnum; ++i) {
        InputPoints (i);
        possobjs[i] = 1;
    }
}


InputPoints (i)

int i;
                /* Input and store all possible footprints for one      */
                /* object.  Transform each point input from R3 into R2. */
{
    FILE *fp1, *fp2, *fopen ();

    int j, k, l, numpts;
    float temp1[AUXMAX * MAX][3], temp2[AUXMAX * MAX][3], farea;
    struct face *tail;
                                        /* open file of object points   */
    switch (i) {
        case 0:
            fp1 = fopen ("obj1.convex", "r");
            fp2 = fopen ("obj1.all", "r");
            fptemp = fopen ("temp1", "w");
```

```
    break;
case 1:
    fp1 = fopen ("obj2.convex", "r");
    fp2 = fopen ("obj2.all", "r");
    fptemp = fopen ("temp2", "w");
    break;
case 2:
    fp1 = fopen ("obj3.convex", "r");
    fp2 = fopen ("obj3.all", "r");
    fptemp = fopen ("temp3", "w");
    break;
case 3:
    fp1 = fopen ("obj4.convex", "r");
    fp2 = fopen ("obj4.all", "r");
    fptemp = fopen ("temp4", "w");
    break;
case 4:
    fp1 = fopen ("obj5.convex", "r");
    fp2 = fopen ("obj5.all", "r");
    fptemp = fopen ("temp5", "w");
    break;
case 5:
    fp1 = fopen ("obj6.convex", "r");
    fp2 = fopen ("obj6.all", "r");
    fptemp = fopen ("temp6", "w");
    break;
case 6:
    fp1 = fopen ("obj7.convex", "r");
    fp2 = fopen ("obj7.all", "r");
    fptemp = fopen ("temp7", "w");
    break;
case 7:
    fp1 = fopen ("obj8.convec", "r");
    fp2 = fopen ("obj8.all", "r");
    fptemp = fopen ("temp8", "w");
    break;
case 8:
    fp1 = fopen ("obj9.convex", "r");
    fp2 = fopen ("obj9.all", "r");
    fptemp = fopen ("temp9", "w");
    break;
case 9:
    fp1 = fopen ("obj10.convex", "r");
    fp2 = fopen ("obj10.all", "r");
    fptemp = fopen ("temp10", "w");
    break;
}

fscanf (fp1, "%d", &Objects[i].numfaces);
for (j = 0; j < Objects[i].numfaces; ++j) {
                                        /* input footprint area */
    fscanf (fp1, "%f %d", &farea, &numpts);
                                        /* input footprint points */
    for (k = 0; k < numpts; ++k)
        fscanf (fp1, "%f %f %f", &temp1[k][0], &temp1[k][1], &temp1[k][2]);
                                        /* transform points to R2 */

    Convert (numpts, temp1, temp2);
                                        /* store CHB points in global structue */

    if (j == 0) {
        Objects[i].pts = (struct face *) calloc(1, sizeof(struct face));
        tail = Objects[i].pts;
    }
    else {
        tail->next = (struct face *) calloc(1, sizeof(struct face));
        tail = tail->next;
    }
                                        /* store area of footprint */
    tail->area = farea;
    tail->numberpts = numpts;   /* store number of points in footprint */
    for (k = 0; k < numpts; ++k)        /* store 2-dimensional points */
        for (l= 0; l < 2; ++l)
            tail->coord[k][l] = temp2[k][l];
    tail->next = NULL;
                                        /* input all points of footprint */

    fscanf (fp2, "%d", &numpts);
```

```
      for (k = 0; k < numpts; ++k)
         fscanf (fp2, "%f %f %f", &temp1[k][0], &temp1[k][1], &temp1[k][2]);

      Convert (numpts, temp1, temp2);              /* convert to R2        */

                                        /* store points in temp file    */
      fprintf (fptemp, "%d\n", numpts);
      for (k = 0; k < numpts; ++k)
         fprintf (fptemp, "%9.5f %9.5f\n", temp2[k][0], temp2[k][1]);

   }
   fclose (fp1);
   fclose (fp2);
   fclose (fptemp);

}



Convert (numpts, temp1, temp2)

int numpts;
float temp1[AUXMAX * MAX][3], temp2[AUXMAX * MAX][3];
                /* Transform points in a plane in R3 to R2 by rotating  */
                /* the plane about the xy intercept line                */

{
   float x[3], y[3], n[3];
   float cos_theta;
                        /* determine normal and intercept of footprint  */
   diff (3, temp1[1], temp1[0], x);
   diff (3, temp1[2], temp1[0], y);
   cross (x, y, n);                        /* determine angle of rotation  */

   x[0] = x[1] = 0;
   x[2] = 1;
   pos (x, n);
   cos_theta = inner (3, x, n);
                        /* determine direction of xy plane intercept    */
   x[0] = -n[1];
   x[1] = n[0];
   x[2] = 0;                           /* rotate points about x to xy plane    */
   Rotate (x, cos_theta, numpts, temp1, temp2);
}



Rotate (k, cos_theta, numpts, temp1, temp2)

float k[3], temp1[AUXMAX * MAX][3], temp2[AUXMAX * MAX][3];
float cos_theta;
int numpts;
                /* Rotate points in temp1 about x by theta radians      */
                /* and store the resulting points in temp2              */

{
   int i, counter;
   float x[3], y[3], z[3], A[3][3], c, s, v;

   counter = 0;
   do {
      c = cos_theta;
      s = sqrt (1 - square(c));
      v = 1 - cos_theta;                   /* determine rotaional transformation   */

      A[0][0] = k[0] * k[0] * v + c;
      A[0][1] = k[1] * k[0] * v;
      A[0][2] = k[1] * s;
      A[1][0] = k[0] * k[1] * v;
      A[1][1] = k[1] * k[1] * v + c;
      A[1][2] = -k[0] * s;
      A[2][0] = -k[1] * s;
      A[2][1] = k[0] * s;
      A[2][2] = c;
```

```
                        /* determine if rotation is c-wise or cc-wise   */
        mult (A, temp1[0], x);
        mult (A, temp1[1], y);
        mult (A, temp1[2], z);

        if (((abs(x[2]-y[2])) < EPSILON2) && ((abs (x[2]-z[2])) < EPSILON2))
           counter = 2;
        else if (counter == 0) {
           cos_theta = - cos_theta;
           counter = 1;
        }
                /* stop if an error due to numerical inaccuracies       */
        else {
           printf ("\nRotate didn't work.\n");
           printf ("Adjust EPSILON2 appropriately.\n");
           exit(0);
        }
    } while (counter < 2);                  /* rotate points to xy plane    */

    for (i = 0; i < numpts; ++i)
        mult (A, temp1[i], temp2[i]);
}


Match (numobjects, numalldpts, numcdpts, possobjs, pindex, area)

int numobjects, numalldpts, numcdpts, possobjs[OBJMAX], *pindex;
float area;
                /* Match footprints for recognition by determining      */
                /* the object which matches the most number of          */
                /* points to the measured footprint

{
    int i, j, max, tempmax;
    float temparea, objsarea[OBJMAX], errorball;
    struct face *tail;

    printf ("\nEnter error radius (in model units): ");
    scanf ("%f", &errorball);

    printf ("\nEnter ratio of data units to model units: ");
    scanf ("%f", &RATIO);

    area /= (square (RATIO));

    max = 0;
    *pindex = 0;
    for (i = 0; i < numobjects; ++i)
        objsarea[i] = EMPTY;

    for (i = 0; i < numobjects; ++i) {
        tail = Objects[i].pts;                  /* for every possible object */
        if (possobjs[i] == 1) {                 /* for each footprint of that object */
                                                /* determine the best match */

            if (i == 0)
                fptemp = fopen ("temp1", "r");
            else if (i == 1)
                fptemp = fopen ("temp2", "r");
            else if (i == 2)
                fptemp = fopen ("temp3", "r");
            else if (i == 3)
                fptemp = fopen ("temp4", "r");
            else if (i == 4)
                fptemp = fopen ("temp5", "r");
            else if (i == 5)
                fptemp = fopen ("temp6", "r");
            else if (i == 6)
                fptemp = fopen ("temp7", "r");
            else if (i == 7)
                fptemp = fopen ("temp8", "r");
            else if (i == 8)
                fptemp = fopen ("temp9", "r");
            else if (i == 9)
                fptemp = fopen ("temp10", "r");
```

```
        tempmax = 0;
        temparea = 0;
        for (j = 0; j < Objects[i].numfaces; ++j) {
           MatchEdges (tail, numalldpts, numcdpts, errorball);

           printf ("\nobject number: %d   pts. matched: %d",i+1,tail->matchedpts);

                                /* determine objects with the most matches */
           if (tail->matchedpts >= tempmax) {
              tempmax = tail->matchedpts;
              temparea = (((abs(tail->area - area)) < (abs(temparea - area))) ?
                          tail->area : temparea);

           }

           tail = tail -> next;
        }
                                           /* store possible objects       */
        if (tempmax > max) {
           for (j = 0; j < i; ++j)
              possobjs[j] = objsarea[j] = EMPTY;
           possobjs[i] = 1;
           objsarea[i] = temparea;
           *pindex = 1;
           max = tempmax;
        }
        else if (tempmax == max) {
           possobjs[i] = 1;
           objsarea[i] = temparea;
           (*pindex)++;
        }
        else if (tempmax < max) {
           possobjs[i] = objsarea[i] = EMPTY;
        }
        fclose (fptemp);
     }
  }
                                                   /* area constraint       */

  if (*pindex > 1)
     for (i = 0; i < numobjects; ++i)
        if ((possobjs[i] == 1) && (area != 0.0)
           && (abs(1 - objsarea[i] / area) > 0.33)) {
              possobjs[i] = 0;
              (*pindex)--;
           }
                                                              /* print result */
  printf ("\n\nMeasured footprint area: %6.2f\n", area);

  printf ("\nRECOGNIZED OBJECT(S):\n");
  for (i = 0; i < numobjects; ++i)
     if (possobjs[i] == 1)
        printf ("%4d       area = %6.2f\n", i + 1, objsarea[i]);
  printf ("\nNumber of points matched: %1d\n", max);

  if (*pindex == 0) {
     for (i = 0; i < numobjects; ++i)
        possobjs[i] = 1;
     *pindex = numobjects;
  }
}



MatchEdges (tail, numalldpts, numcdpts, errorball)

int numalldpts, numcdpts;
float errorball;
struct face *tail;
                /* Match the corners of the CHB of the measured          */
                /* footprint to the corners of the CHB of all possible   */
                /* footprints generated by the object models             */
{
   int i, j, k, l, m, n, o;
```

```
int pt1, pt2, matches, max, maximum, close;
float x[3], y[3], testpt[3], transfpt[3], R[3][3], c, s;
float temp[AUXMAX * MAX][2];
int num;
                        /* input all points in the model footprint      */
                        /* (these points were stored after being        */
                        /* rotated to R2)
fscanf (fptemp, "%d", &num);
for (i = 0; i < num; ++i)
   fscanf (fptemp, "%f %f", &temp[i][0], &temp[i][1]);

maximum = 0;
                        /* for CHB every edge of the data footprint     */
for (i = 0; i < numcdpts; ++i) {
   idiff (2, cdpts[i], cdpts[(i+1) % numcdpts], x);
   x[2] = 1;

   max = 0;
                        /* for every CHB edge of the model footprint    */
   for (j = 0; j < (tail -> numberpts); ++j) {
      diff(2, (tail->coord)[j], (tail->coord)[(j+1)%(tail->numberpts)], y);
      y[2] = 1;

            /* determine the absolute value of the rotation angle       */
      c = inner (2, x, y);
      s = sqrt (1 - square(c));
      pt1 = i;
      pt2 = j;
                              /* determine rotation transformation      */
                              /* for both possible edge orientations    */
      for (k = 0; k < 2; ++k) {
            /* determine if transformation is c-wise or cc-wise         */
         transfpt[0] = c * x[0] - s * x[1];
         transfpt[1] = s * x[0] + c * x[1];

         if (dist (2, transfpt, y) > EPSILON2)
            s = - s;
                        /* match up edges for both sets of endpoints    */
         for (l = 0; l < 2; ++l) {
                        /* determine the matching transformation        */
            R[0][0] = R[1][1] = c / RATIO;
            R[0][1] = -s / RATIO;
            R[1][0] = s / RATIO;
            R[0][2] = - cdpts[pt1][0] * c / RATIO +
                        cdpts[pt1][1] * s / RATIO + (tail->coord)[pt2][0];
            R[1][2] = - cdpts[pt1][0] * s / RATIO -
                        cdpts[pt1][1] * c / RATIO + (tail->coord)[pt2][1];
            R[2][0] = R[2][1] = 0;
            R[2][2] = 1;
                        /* determine the number of points matched       */
                        /* from all points of each footprint            */
            matches = 0;
            for (m = 0; m < numalldpts; ++m) {
               for (n = 0; n < 2; ++n)
                  testpt[n] = (float) alldpts[m][n];
               testpt[2] = 1;
               mult (R, testpt, transfpt);   /* transform one point    */
                        /* determine if it matches to a point in the    */
                        /* model footprint
               close = 0;
               for (n = 0; n < num; ++n) {
                  for (o = 0; o < 2; ++o)
                     testpt[o] = temp[n][o];
                  if (dist (2, testpt, transfpt) < errorball)
                     close = 1;
               }
               if (close)
                  ++matches;
            }
            if (matches > max)
               max = matches;
```

```
                                        /* switch pairs of endpoints    */
                (pt1 == i) ? (pt1 = (i+1)%numcdpts) : (pt1 = i);
                (pt2 == j) ? (pt2 = (j+1)%(tail->numberpts)) : (pt2 = j);
            }
                                        /* switch orientation of edge   */
            for (i = 0; i < 2; ++i)
                x[i] = -x[i];
            c = - c;
            pt1 = (i + 1) % numcdpts;
            pt2 = j;
        }
    }
                        /* determine the max number of matches for      */
                        /* all matchings of the CHB edges of the model  */
                        /* to one edge of measured footprint            */
    if (max > maximum)
        maximum = max;
  }
  tail -> matchedpts = maximum;
}


CleanUp (numobjects)

int numobjects;
                /* Delete the temporary files created for storing the   */
                /* transformed points of all footprints of the modeled  */
                /* objects
{
    if (numobjects >= 1)
        unlink ("temp1");
    if (numobjects >= 2)
        unlink ("temp2");
    if (numobjects >= 3)
        unlink ("temp3");
    if (numobjects >= 4)
        unlink ("temp4");
    if (numobjects >= 5)
        unlink ("temp5");
    if (numobjects >= 6)
        unlink ("temp6");
    if (numobjects >= 7)
        unlink ("temp7");
    if (numobjects >= 8)
        unlink ("temp8");
    if (numobjects >= 9)
        unlink ("temp9");
    if (numobjects >= 10)
        unlink ("temp10");
}
```

```
/*                                                                      */
/*                      Vector.c                                        */
/*                                                                      */
/*      This file contains vector and matrix routines used in Face.c    */
/*      Corner.c and Recog.c.                                           */
/*                                                                      */

#define EPSILON 1.0e-3
#include "header.h"
#include <stdio.h>


                                /* compute inner product of vect1 and vect2 */
float inner (dim, vect1, vect2)

int dim;
float *vect1, *vect2;

{
   int i;
   float sum;

   sum = 0;
   for (i = 0; i < dim; ++i)
      sum += *vect1++ * *vect2++;
   return (sum);
}


normalize (dim, vect)           /* normalize vector vect */

int dim;
float *vect;

{
   int i;
   float mgnsqd;

   mgnsqd = 0;
   for (i = 0; i < dim; ++i) {
      mgnsqd += square(*vect);
      vect++;
   }

   if (mgnsqd != 0) {
      vect -= dim;
      for (i = 0; i < dim; ++i)
         *vect++ /= sqrt(mgnsqd);
   }
}


cross (vect1, vect2, vect3)         /* take the cross product of vect1 and
                                       vect2 and put thee result in vect3 */

float vect1[3], vect2[3], vect3[3];

{
   vect3[0] = vect1[1] * vect2[2] - vect1[2] * vect2[1];
   vect3[1] = vect1[2] * vect2[0] - vect1[0] * vect2[2];
   vect3[2] = vect1[0] * vect2[1] - vect1[1] * vect2[0];

   normalize (3, vect3);
}


                                /* determine if point pt is in the plane */
plane (normal, pt, D)

float normal[3], pt[3], D;

{
```

```
    if (abs (inner (3, normal, pt) + D) < EPSILON)
        return (1);
    else return (0);
}


diff (dim, vect1, vect2, vect3)      /* compute the normalized difference of
                                        vect1 and vect2 and put the result in vect3 */

int dim;
float *vect1, *vect2, *vect3;

{
    int i;

    for (i = 0; i < dim; ++i)
        *vect3++ = *vect1++ - *vect2++;

    normalize (dim, vect3 - dim);
}


idiff (dim, vect1, vect2, vect3)     /* compute the normalized difference of
                                        integer vectors vect1 and vect2 and
                                        put the result in float vector vect3 */

int dim, *vect1, *vect2;
float *vect3;

{
    int i;

    for (i = 0; i < dim; ++i)
        *vect3++ = *vect1++ - *vect2++;

    normalize (dim, vect3 - dim);
}


projectcom (com, normal, D, pt)      /* project point com perpendicularly down
                                        to the plane specified by normal normal
                                        and intercept D and store the result in
                                        pt */

float com[3], normal[3], D, pt[3];

{
    int i;
    float t;

    t = -(inner (3, com, normal) + D);

    for (i = 0; i < 3; ++i)
        pt[i] = com[i] + normal[i] * t;
}


pos (vect1, vect2)                   /* set vect 1 = -vect1 if <vect1, vect2> < 0 */

float vect1[3], vect2[3];

{
    int i;

    if (inner (3, vect1, vect2) < 0)
        for (i = 0; i < 3; ++i)
            vect1[i] = - vect1[i];
}
```

```
/* multiply vector x by matrix A (on the left)
   and put the result in vector y                */
mult (A, x, y)

float A[3][3], x[3], y[3];

{
   int i, j;

   for (i = 0; i < 3; ++i) {
      y[i] = 0;
      for (j = 0; j < 3; ++j)
         y[i] += A[i][j] * x[j];
   }
}


/* compute euclidean distance between
   vect1 and vect2 */
float dist (dim, vect1, vect2)

int dim;
float *vect1, *vect2;

{
   int i;
   float sum;

   sum = 0;
   for (i = 0; i < dim; ++i) {
      sum += square ((*vect1 - *vect2));
      *vect1++;
      *vect2++;
   }

   return (sqrt(sum));
}


/* compute euclidean norm of vect1 */
float norm (dim, vect1)

int dim;
float *vect1;

{
   return (sqrt(inner (dim, vect1, vect1)));
}
```

```
/* ---------------------- wolfep.c ------------------------
**
**      Title:  wolfep_
**
**      Author: John Hauser
**              hauser@esvax.berkeley.edu
**              (415) 642-4235
**
**
**      This function implements Philip Wolfe's procedure for
**      finding the nearest point in a polytope as described in
**
**              P.Wolfe, "Finding the nearest point in a polytope",
**              Mathematical Programming 11 (1976) 128-149.
**
**      The function is written in C with parameters passed by
**      address (pointers) to allow linking with FORTRAN routines.
**      Under normal circumstances, the norm of the nearest point
**      is returned as the value of the function with nearest point
**      in the vector 'x'.  If a negative number is returned (-1. to
**      be precise), the test in step 1(d) has failed.  This test
**      is a guard on the affine independence of the points in the
**      corral and may fail when nearly identical points are present.
**      If this happens consult the paper and consider increasing
**      the value of 'Z1' (used in stop condition).
**
**      The actual calling sequence is given in the declaration and
**      comments below.  The integer array 's' and the double array
**      'rr' are only used as working areas -- no solution information
**      is return in them.
**
**      In order to keep the size of the work area a function of the
**      dimension of the points ('n') only and not of the number of
**      points ('m'), I have used dynamic allocation ('malloc' and
**      'free') to get space to store the squared norms of all the
**      points when first computed.  If this is undesireable or
**      unavailable, a larger 'rr' can be used instead by replacing
**      the line with 'malloc' in it by:
**
**              pn2 = &r(0,nn+4);
**
**      and deleting the line
**
**              free((char *) pn2);
**
**      The 'rr' must point to a work area of at least
**
**              (n + 1)*(n + 4) + m
**
**      double precision elements.
**
*/

#define Z1                      1e-12
#define Z2                      1e-10
#define Z3                      1e-10
#define FALSE                   0
#define TRUE                    ~FALSE
#define max(a,b)                {(a)>(b)?(a):(b)}
#define min(a,b)                {(a)<(b)?(a):(b)}

#include <math.h>

double wolfep_(pp,  n,  m,  x,  s,  rr)
double          *pp,        *x,         *rr;
int                 *n, *m,     *s;
{

    /*
    **  pp --   pointer to points array (n * m elements at least)
    **  n  --   dimension of points (pointer to)
    **  m  --   number of points (pointer to)
    **  x  --   minimizing point in polytope (n element vector)
    **  s  --   index array for corrals (at least n + 1 elements)
    **  rr --   work space (at least (n + 1) * (n + 4) elements)
    **          first n + 1 columns are used for r
    **          n + 2 column is w
    **          n + 3 column is v
```

```
    **          n + 4 column is a scatch vector
    */

#define p(i,j)  pp[(i) + (j)*nn]                    /* stored columnwise */
#define r(i,j)  rr[(i) + (j)*qq]                    /* stored columnwise */

    int i,
        j,
        mini,
        nn,                 /* dimension of points */
        mm,                 /* number of points */
        sn,                 /* number of points in current corral */
        qq,                 /* length of column in r */
        small,              /* boolean toggle */
        zero1;              /* point with zero weight */

    double d,
        dmin,
        pn2max,             /* max 2 norm of corral points */
        *pn2,               /* pointer to norms of points */
        *w,                 /* pointer to weights for 'x' */
        *v,                 /* pointer to weights for 'y' */
        *bb,                /* pointer to scratch vector */
        theta,              /* convex multiplier */
        a, b, c,            /* used for plane rotation */
        dot(),
        norm2();

    char *malloc();

    /* initialize pointers, etc. */

    nn = *n;                            /* dimension of points */
    mm = *m;                            /* number of points */
    qq = nn + 1;                        /* size of maximal corral */
    w = &r(0,nn + 1);                   /* weights for 'x' */
    v = &r(0,nn + 2);                   /* weights for 'y' */
    bb = &r(0,nn + 3);                  /* scratch vector */
                /* squared norms of points */
    pn2 = (double *) malloc((unsigned) (mm*(sizeof(double))));

    /*  step 0  --  get initial corral  */

    mini = i = mm - 1;
    dmin = pn2[i] = norm2(&p(0,i),nn);
    for (; i--; )
        if ((pn2[i] = norm2(&p(0,i),nn)) < dmin)  {
            dmin = pn2[i];
            mini = i;
            }

    sn = 1;                     /* one point in the corral */
    w[0] = 1.;
    s[0] = mini;

    /* initial R matrix (1 by 1) and set max squared norm */

    r(0,0) = sqrt(1. + (pn2max = pn2[mini]));


    /* main loop of algorithm */

    for (;;) {

        /* step 1(a) - set X = P[S]*w  */

        for (i = nn; i-- ; ) {
            d = 0.;
            for (j = sn; j--  ; )
                d += p(i,s[j])*w[j];
            x[i] = d;
            }

        /* step 1(b) -- define mini to minimize <X, P[i]>, i < m */

        mini = i = mm - 1;
        dmin = dot(x, &p(0,i), nn);
        for (; i--  ; )
```

```
        if ((d = dot(x, &p(0,i), nn)) < dmin) {
            dmin = d;
            mini = i;
            }

    /* step 1(c) - test for end condition */

    if (dmin > (d = norm2(x, nn)) - Z1*max(pn2[mini], pn2max)) {
        /* stop - we have a solution */
        /* clean up for return */

        free((char *) pn2);

        return (sqrt(d));           /* d contains the squared distance */
        }

    /* step 1(d) - check for mini in S */

    for (i = sn; i--; )
        if (mini == s[i]) {
            /* temporary disaster see note 3 */

            return (-1.);
            }

    /* step 1(e) - add new point to the corral */

    s[sn] = mini;
    w[sn] = 0;
    pn2max = max(pn2max, pn2[mini]);

    /* steps 1(f) & (g) - compute new column of R */

    for (i = sn; i--; )
        bb[i] = 1. + dot(&p(0,s[i]), &p(0,mini), nn);
    solve_t_upper(rr, &r(0,sn), bb, sn, qq);
    r(sn,sn) = sqrt(1. + pn2[mini] - norm2(&r(0,sn), sn));
    sn++;

    /* loop for steps 2 & 3 */

    for (;;) {

        /* step 2 - solve system for new v (try for w) */

        for (i = sn; i-- ; )        /* init vector of ones (e) */
            v[i] = 1.;

        solve_t_upper(rr, bb, v, sn, qq);
        solve_upper(rr, v, bb, sn, qq);

        /* normalize v & check for small values */

        d = 0.;
        for (i = sn; i--; )
            d += v[i];
        small = FALSE;
        theta = 1.;                 /* also compute theta for 3(a) & (b) */
        for (i = sn; i--; )
            if ((v[i] /= d) <= Z2) {
                small = TRUE;
                if ((dmin = w[i] - v[i]) > Z3)
                    theta = min( theta, w[i]/dmin);
                }

        if (!small) {           /* good positive v - put in w */
            for (i = sn; i--; )
                w[i] = v[i];
            break;              /* break out (go to step 1) */
            }

        /* step 3 */

        /* theta for steps 3(a) & 3(b) computed above */

        /* steps 3(c) & 3(d) - intersect segment & convex hull */
            /* note difference from Wolfe's paper:              */
            /*      w  =  theta * v  +  (1 - theta) * w         */
```

```
            for ( i = sn; i--; )
                if ((w[i] = theta*v[i] + (1.-theta)*w[i]) <= Z2) {
                    w[i] = 0.;
                    zeroi = i;
                    }

            /* steps 3(e) & 3(f) - delete zeroi-th element from S, w, & R   */

            d = pn2[s[zeroi]];  /* save to update max Pn2 */

            sn--;

            for (mini = (i = zeroi) + 1; i < sn; i++, mini++) {
                s[i] = s[mini];
                w[i] = w[mini];
                for (j = sn + 1; j--; )
                    r(j,i) = r(j,mini);
                }

            /* update Pn2max */

            if (d == pn2max) {
                pn2max = pn2[s[(i = sn - 1)]];
                for ( ; i--; )
                    pn2max = max( pn2max, pn2[s[i]]);
                }

            /* step 3(g) - use plane rot. to maintain upper triangular R */

            for (mini = (i = zeroi) +1; i < sn; i++, mini++) {
                a = r(i,i);
                b = r(mini,i);
                a /= (c = sqrt(a*a + b*b));
                b /= c;
                for (j = i; j < sn; j++) {
                    r(i,j) = a*(c = r(i,j)) + b*(d = r(mini,j));
                    r(mini,j) = -b*c + a*d;
                    }
                }

            }   /* end of step 2 & 3 loop */

        }       /* end of main loop */

#undef r
#undef p
    }           /* end of dist function */


/*      solve_upper     solve R*x = b w/ R upper triangular
**
**              note:  do not link with fortran
*/

solve_upper(R,  x,  b, n, q)
double     *R, *x, *b;
int                   n, q;
{
#define RR(i,j) R[(i) + (j)*q]          /* stored columnwise */

    int i, j;
    double z;

    for (i=n; i--; ) {
        z = 0.;
        for (j=i+1; j<n; j++)
            z += RR(i,j)*x[j];
        x[i] = (b[i] - z)/RR(i,i);
        }

#undef RR
    }   /* end of solve_upper

/*      solve_t_upper   solve R-tr*x = b w/ R upper triangular
**
**              note:  do not link with fortran
*/
```

```
solve_t_upper(R,  x,  b, n, q)
double       *R, *x, *b;
int                  n, q;
{
#define RR(i,j) R[(i) + (j)*q]

    int i, j;
    double z;

    for (i=0; i<n; i++)  {
        z = 0.;
        for (j=i; j--; )
            z += RR(j,i)*x[j];
        x[i] = (b[i] - z)/RR(i,i);
        }

#undef RR
    }   /* end of solve_t_upper


/* dot  returns dot product of 2 n-vectors
**
**      note:  not to be linked with fortran routines
*/

double dot(x,  y, n)
double    *x, *y;
int              n;
{
    double d;

    d = 0;
    for (; n--; x++, y++)
        d += *x * *y;
    return(d);
    }

/* norm2  returns sqared norm of n-vector
**
**      note:  not to be linked with fortran routines
*/

double norm2(x, n)
double      *x;
int            n;
{
    double d;

    d = 0;
    for (; n--; x++)
        d += *x * *x;
    return(d);
    }
```

```
#define MAX 40
#define AUXMAX 7
#define OBJMAX 5
```

```
#define abs(x) ((x) < 0 ? -(x) : (x))
#define square(x) (x) * (x)
double sqrt();
```