Geometric Interpretation and Comparisons of Enhancements of GJK Algorithm for Computing Euclidean Distance Between Convex Polyhedra

Jing-Sin Liu*, Yuh-ren Chien, Shen-Po Shiang and Wan-Chi Lee
Institute of Information Science 20
Academia Sinica
Nankang, Taipei 115, Taiwan
R.O.C.
Email: liu@iis.sinica.edu.tw
(*correspondence address)

Abstract

The computation of Euclidean distance between two convex polyhedra is an important problem in robotics, computer graphics and real-time animation. An efficient and reliable distance computation procedure for a pair of convex sets is developed by Gilbert, Johnson and Keerthi (GJK) in 1988. GJK's convex set-theoretic approach is general and suitable for iterative numerical computation, however the GJK algorithm has the drawback of lack geometric intuition, especially in pairwise distance computation of convex polyhedra defined by the convex hull of its vertices in Euclidean 2D and 3D space. This paper investigates the steps of GJK algorithm from a geometric but intuitive point of view. By geometric reasoning, we present an improvement of the Euclidean distance computation algorithm made by GJK. Some comparative simulations for both the static and tracking cases, in particular, collision-free motion planning of redundant robots, are shown to verify the algorithmic improvement in the process of distance computation. In addition, our work provides a simple and efficient algorithm for finding out the information where the closest point of a convex polyhedron to a reference point is: on the face, the edge, or on one vertex of the polyhedron.

Keywords. Distance computation, convex polyhedron, GJK algorithm, Euclidean distance, minimum distance.

Ⅰ Introduction

In robotics, computer graphics and animation, the Euclidean distance between a detected object and the obstacles around is an indispensable information when moving the object by manipulation robots or for realistic, 3D environment modeling. It lies at the base of many applications, such as CAD/CAM, intersection detection[7], collision avoidance[5, 18, 19], path planning[3, 20], and path modification[4], where knowledge of the distance measures between the robot and its environment (or in general two moving objects) is crucial. Polyhedral objects are often used in graphical simulation of robot systems due to their simple and reduced computation in simulation. There are considerable interests in developing efficient minimum distance computation algorithms, especially for convex polyhedra. In computational geometry, several asymptotically efficient algorithms are developed for 2D convex polygons[22, 24] or 3D convex polyhedra[23]. For applications to collision-free robot motion planning and graphical simulations involving collision detection, there are works on minimum distance computation, e.g. [1, 7, 9, 10, 13, 17, 19]. In general, the problem of finding the distance between convex bodies[1, 13, 17] can be equivalent to the direct minimization of distance function, or the procedure of computing the closest point of a body to the reference point in a translated space. For example, Rimon and Boyd[10] used the L-J ellipsoid to enclose the convex or non-convex objects to formulate a convex optimization problem for estimated distance computation. The computation of the distance estimate (within a user's specified error) is shown to be an eigenvalue problem. Lin and Canny[6] provided an incremental algorithm of almost-constant time complexity for tracking a pair of closest points of convex bodies, one on each body, in three dimensional space. The framework presented by Johnson and Cohen[11] for minimum distance computation also gives an efficient solution for objects described by different surface representations. Among these works, of special concern in this paper is the computation procedure developed by Gilbert, Johnson, and Keerthi[1] (GJK) in 1988. Gilbert, Johnson, and Keerthi (GJK), based on their prior work[18], presented an efficient distance computation algorithm[1] in which an object is defined by the convex hull of its vertices and the convex set is represented in terms of their support properties. For polytopes that is defined as the convex hull of a finite set of vertices, in particular, the properties can be easily obtained from their vertices. Using GJK, it is possible to eliminate a large number of pairwise distance computation between faces of two convex polyhedron, thus the efficiency is competitive. Attempts on improvements on GJK algorithm were made in last years. The GJK algorithm has been further extended [6, 7, 16] and organized[2, 8]. With some modification of support function computation by Cameron[2], this method can also provide constant time updates for slowly moving polyhedra. Chung Tat Leung [7] presented an efficient means of updating the Minkowski difference to create a collision detection method for convex polyhedra.

The paper is organized as follows. Section 2 is a summary of GJK algorithm. GJK algorithm, which involves the computationally intensive step of computing the supporting function of the set of vertices, is a set-theoretic approach in essence. Since in applications the distance between two objects needs to be updated from time to time, every possible enhancement of distance computation procedure can speed up the repetitive process as the time goes on. In Section 3, we present some modifications after carefully examining the steps of GJK algorithm by geometric reasoning. The changes we made would give more geometric meaning and thus readability of the output of the algorithm, while keep the results that confirm an almost-linear time complexity. In Section 4, a recent enhancement of GJK algorithm made by Cameron[2] in the computation of support function is introduced for comparisons study. Efficiency comparisons for distance computation between two static bodies, and two moving bodies are made. Also application to collision-free motion planning of redundant robots is shown to illustrate the long-time efficiency of the modified distance computation, which is important in on-line applications. A convenient method is developed to find out the neighbor points, so-called vicinity matrix, which is the information required in the "hill climbing"[8, 9] of Cameron's enhancement. The verification of neighbor points of a vertex is the key to achieve the constant time complexity. In other words, if the adjacency information of vertices is available before computing the distance, the computation will be much more efficient. Because the body is assumed rigid, the neighboring points in a body won't change during motion and the verification can be viewed as a preprocessing procedure. The preprocessing, however, can consume a lot of time even more than the distance algorithm itself, as shown in Section 4. Section 5 is the conclusion.

Ⅱ The GJK Algorithm

To find out the distance of two convex polyhedra named K_1 and K_2, suppose the vertices of K_1 are s_1, s_2, …, s_n (n points), and the vertices of K_2 are t_1, t_2, …, t_p (p points), respectively. From K_1 and K_2 , the set $K = \{ s_i\text{-}t_j,\ s_i \in K_1,\ t_j \in K_2\}$ (also a convex polyhedron) ,called the Minkowski difference of the two polyhedra, can be constructed, where the elements are the relative translations of K_1 and K_2, in translational configuration (TC) space[2]. There should be n*p vertices in K. The separation between the original two polyhedra is equal to the distance between the origin of TC space and the convex obstacle formed by the points of K is called TCSO (translational C-space obstacle)[1, 2, 8].

Let d(x,y) be the Euclidean distance between two points x and y, and K_1 , K_2 be two convex polyhedra. The minimum distance between K_1 and K_2 is defined as d (K_1, K_2)=min{d (x,y): x is on the boundary of K_1, y is on the boundary of K_2} =d (x*,y*) for $x^* \in K_1$ and $y^* \in K_2$. x*, y* are called the pair of closest points of K_1 , K_2 . The GJK algorithm for computing the distance between two convex polyhedra is shown in Fig.1.

It is essential to describe the GJK algorithm first before we introduce the algorithmic modifications that we make. Essentially, the GJK algorithm iteratively searches a subset of the original set of vertices that contains the global minimum of distance functions. Essentially, the GJK algorithm iteratively searches a subset of TCSO which contains near point closer to origin than near point of the subset last step. The process of subset modification proceeds by eliminating the vertices unnecessary to determine the nearest point and adding a closer vertex found from supporting property.

2.1 Definitions

The main advantage of GJK algorithm is the specification of the convex sets in terms of their support properties. Recall the definition of the support function[1] $H_X : R^m \to R$ for a polyhedron X is the evaluation of inner products of a fixed vector with all vertices of polyhedron and looking for the maximum:

$$H_X(\eta) = \max\{x \cdot \eta : x \in X\}$$

where $\eta \in R^m$ is a given vector, · is the inner product. Define the support mapping S_X :

```
GJK Algorithm:
$V_0 \leftarrow$ initial set ;
$i \leftarrow 0$ ;
Repeat{
$Y_s \leftarrow$ find_affinely independent_set($V_i$) ;
$\nu_i \leftarrow$ nu_compute($V_i$) ;
$Y_s \leftarrow$ refine_set($V_i$)
$(S_i, H_i) \leftarrow$ support_functions($-\nu_i$) ;
$V_{i+1} \leftarrow Y_s \cup \{S_i\}$ ;
$i \leftarrow i + 1$ ;
} until ( $\nu_i \bullet \nu_i + H_i = 0$)
```

Fig. 1 The GJK algorithm

$R^m \to X$ to be any mapping that, given a direction η, is the solution of the support function, i.e. one of the points in X which is farthest in the direction η:

$$H_X(\eta) = S_X(\eta) \cdot \eta$$

The (nonunique) points $S_X(\eta)$ of X is called the supporting vertex of X in the direction η. It has the property that the hyperplane passing through it with normal η is a supporting hyperplane of X. For two polyhedra K_1 and K_2, we have

$$H_K(\eta) = H_{K1}(\eta) + H_{K2}(-\eta); \quad S_K(\eta) = S_{K1}(\eta) - S_{K2}(-\eta).$$

i.e. find supporting vertex $S_{K1}(\eta)$ on K_1 and $S_{K2}(-\eta)$ on K_2 in the direction η and - η, respectively.

The GJK algorithm shown in Fig. 1 finds the closer vertex for each polyhedron by using the support function which makes the updated point S_i of set V_i remain on the TCSO. The set will satisfy the goal, $\nu_i \bullet \nu_i + H_i = 0$, by dropping the affinely independent set vertices irrelevant in determining local near point and taking a new point S_i into consideration. The goal is achieved by checking if the points A, B in Fig. 2 satisfy

$$H_{K1}(A-B) = A\cdot(A-B) \text{ and } H_{K2}(A-B) = B\cdot(-(A-B))$$

Or whether K_1 lies to the left of the line through A and orthogonal to (A-B), and similarly for K_2 and B.

Note that the goal $\nu_i \bullet \nu_i + H_i = 0$ can be replaced by the more robust condition "reoccurrence of supporting vertices" in the iterations of GJK algorithm to avoid numerical imprecision due to roundoff error of the floating point addition[7].

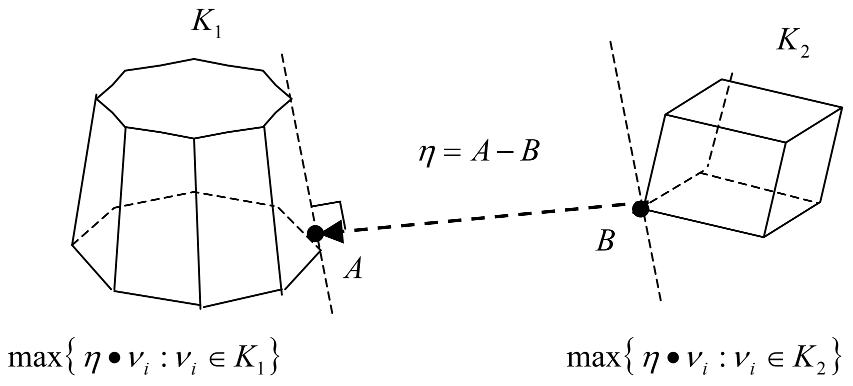
K_1
K_2
$\eta = A - B$
A
B
$\max\{\eta \bullet v_i : v_i \in K_1\}$
$\max\{\eta \bullet v_i : v_i \in K_2\}$

Fig. 2. Illustration of support function evaluation.

Definition[12]. A set of $m + 1$ points $\{b_0, b_1, \ldots, b_m\}$ is said to be *affinely independent* if

$$aff\{b_0, b_1, \ldots, b_m\} = \left\{ \sum_{i=1}^{l} \lambda_i x_i : x_i \in \{b_0, b_1, \ldots, b_m\},\ \lambda_1 + \ldots + \lambda_l = 1 \right\} \qquad (1)$$

is m-dimensional.

From the above definition,

$$\text{aff}\{b_0, b_1, \ldots, b_m\} = L + b_0,$$

where

$$L = \text{aff}\{0, b_1 - b_0, \ldots, b_m - b_0\}.$$

L is the same as the smallest subspace containing $b_1 - b_0, \ldots, b_m - b_0$. Its dimension is m if and only if these vectors are linearly independent.

Thus $b_0, b_1, \ldots, b_m$ are affinely independent if and only if $b_1 - b_0, \ldots, b_m - b_0$ are linearly independent. Furthermore, the coefficients λ_i in such an expression (1) of a point in aff{b_0,

$b_1, \ldots, b_m\}$ are unique if and only if $b_0, b_1, \ldots, b_m$ are affinely independent.

2.2 GJK algorithm (Fig. 1)

The algorithm is more conveniently described by using the pictures in TC space, although the algorithm itself never needs to explicitly construct the TCSO. The key element of the approach is the algorithm for computing the distance between origin and convex sets in m-dimensional space. m = 3 is a special case of this algorithm, and the convex sets are polytopes defined by their vertices. The sets defined in the approach, without loss of generality, always contain not larger than four elements because of the Caratheodory theorem[12]. Arbitrary set of one to four points, each is the difference of two vertices (one from K_1 and one from K_2), can be chosen as the initial set in GJK algorithm. One better choice[1, 9] is to find the direction of the vector defined by the difference of the centers of two objects, and then compute a most appropriate point on the TCSO by the use of support function. It is because the closest points of two convex polyhedra are usually in the direction we just mentioned.

In GJK algorithm, the step of determining affinely independent set $Y_s = \{ y_i \in K: i \in I_s \}$ from V_i and the minimum distance point ν_i of the polyhedron formed by Y_s is the Distance Subalgorithm. the step of finding the minimum distance point ν_i of the polyhedron formed by V_i and determining affinely independent set $Y_s = \{ y_i \in K: i \in I_s \}$ which contains ν_i from V_i is the Distance Subalgorithm. The Distance Subalgorithm in its mathematical form is presented in the following:

Take a subset Y_s from V_i. Define the real number $\Delta_i(Y_s)$, $\Delta(Y_s)$ by

$\Delta_i(\{y_i\}) = 1, \quad i \in I_s \quad$ (I_s = The numbers of elements in Y_s)

$\Delta_j(Y_s \cup \{y_j\}) = \Sigma\, \Delta_i(Y_s)(y_i \cdot y_k - y_i \cdot y_j) \quad$ for all $i \in I_s$, $\quad k \in I_s$, $j \in I_s'$

(I_s' = The complement of I_s)

$\Delta(Y_s) = \Sigma_i \Delta_i(Y_s) \quad$ for all $i \in I_s$.

Then the output of the Distance Subalgorithm consists of positive real numbers λ_i and set Y_s :

$$\lambda_i = \Delta_i(Y_s)/\Delta(Y_s); \quad \Sigma_i \lambda_i = 1, \quad \lambda_i > 0 \qquad (2)$$

The closest point ν_i computed can be expressed as

$$\nu_i = \Sigma_i (\lambda_i y_i) \quad \text{for all } i \in I_s \qquad (3)$$

This completes the GJK algorithm for computing the closest point to the origin in TC-space. By finding the initial set of next loop, the ν is computed by a recursive formula for finding out each $\Delta_i(Y_s)$ and is used in later application of support function. We can calculate ν_i in the form of (3) if we know the affinely independent set Y_s and λ_i in (2) is the solution of (3). The approach used in GJK algorithm (Theorem 3 in Gilbert, Johnson, and Keerthi[1]) is to check all the subsets of every initial set in each loop of the algorithm whether the following three conditions are satisfied:

(a) $\Delta(Y_s) > 0$, (b) $\Delta_i(Y_s) > 0$ for each $i \in I_s$, (c) $\Delta_j(Y_s \cup \{y_i\}) \leq 0$ for each $j \in I_s'$. (4)

Usually, Y_s is uniquely determined.

Ⅲ.Geometric Interpretation and The Modification of GJK Algorithm

The algorithm in our improvement is based on three functions nu_compute(), support_functions(), and refine_set(), which make the concept more obvious and the implementation easier. In addition, the feature of the closest point (i.e. vertex, edge or face) can be found.

The modification is applicable to two- or three-dimensional space. However, the situations are more complicated in three-dimensional space. In what follows, we will discuss two-dimensional space first, then the situations in three- dimensional space follows.

3.1 Two-dimensional case

(a) Description of the algorithm (Fig. 3):

In the 2D TC-space, our modification in an iteration is: choose a triangle, discard a vertex of it, and then add another vertex to construct a new triangle.

First, take two vertices Z_1 and Z_2 on the TCSO as the initial set V_0. Then compute the nearest point ν_i using the function nu_compute(). It is known that ν_i is the nearest point on the object formed by the points of set V_i to the origin of the TC-space and is an approximation to the nearest point at *ith* iteration[1, 14, 16]. After ν_i is determined, the algorithm then obtains a new point in K from support_functions(). In the algorithm, the new point is named $S_i = S_K(-\nu_i)$, which is farthest toward the origin away from ν_i. The new point thus found will give us the optimized path to the final answer[1]. S_i and two other initial points form next set V_{i+1}.

Geometrically clear, the ν_i of this set is on one edge of the triangle which is made by S_i and the initial points. Thus, the function of refine_set() is to refine the new initial set V_i of next iteration with the two points that compose this edge. The procedure continues until the termination condition $\nu_i \bullet \nu_i + H_i = 0$ is satisfied. In other words, the iteration in Fig. 3 will terminate when the points of V_i can form the nearest edge of the TCSO to the origin. The ν_i we get at this time is what we look for. Fig. 4 shows an illustration of basic cycle of the process.

```
The Algorithm in 2D:
$V_0 \leftarrow$ initial_set($Z_1$, $Z_2$) ;
 $i \leftarrow 0$ ;
Repeat{
    $\nu_i \leftarrow$ nu_compute($V_i$) ;
    ($S_i$, $H_i$) $\leftarrow$ support_functions($-\nu_i$) ;
    $V_{i+1} \leftarrow$ refine_set($V_i$, $S_i$) ;
    $i \leftarrow i + 1$ ;
} until ( $\nu_i \bullet \nu_i + H_i = 0$)
```

Fig.3 GJK algorithm with our modification in 2D case.

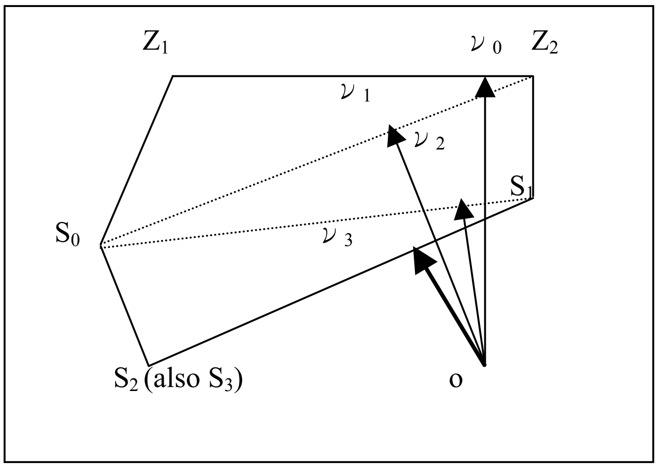

Fig.4. An illustration of GJK.

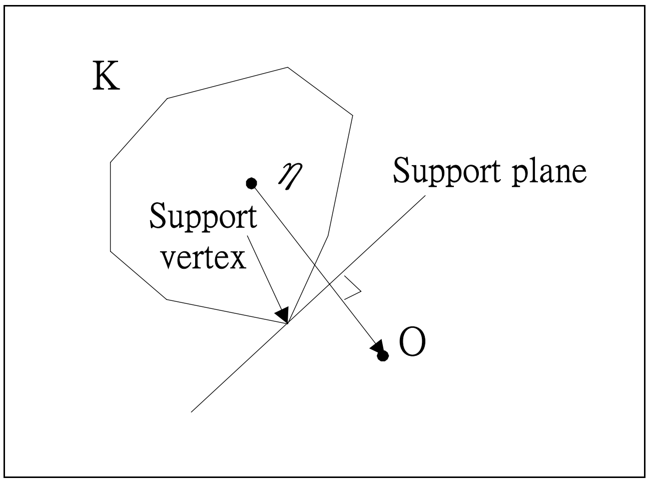

Fig5. The choice of initial point

(b). The choice of initial points:

One of the most important ways to reduce the computation time is to make a good choice of initial points[1, 13, 17]. Because the available points are on the boundary of the TCSO, we must take advantage of the feature of support function. One is chosen to be the supporting vertex with respect to the vector pointing from the centroid to the origin in the TC space, i.e. $S_K(\eta)$ where η is centroid of K_1 minus centroid of K_2, as illustrated in Fig. 5. In order not to initialize the algorithm in some particular points, the other initial point is distinct but arbitrary so that it can effectively produce two new points far apart after the calculation of support function. This is a good initial choice for efficient computation.

(c). The computation of ν (about nu_compute()) :

Each ν is computed by the function nu_compute. It represents the nearest point on the TCSO in the polytope in each step to the origin of the TC-space[1, 2]. It is also a vector that gives us the direction from the origin to the nearest point on the TCSO in the polytope. The idea of this function is mainly the same as that used in Gilbert, Johnson, and Keerthi[1], but the sets we use are sometimes different. As will be seen later, λ might be negative in the computations. Because we won't consider the affinely independent set, there are always two points (say X_1, X_2) contained in the original set V_i. The nearest point of the original triangle edge can be calculated as follows. Given two points X_1, X_2, let

$$\Delta_{1,X_1X_2} = X_2 \cdot X_2 - X_2 \cdot X_1,$$

$$\Delta_{2,X_1X_2} = X_1 \cdot X_1 - X_2 \cdot X_1,$$

$$\Delta_{X_1X_2} = \Delta_{1,X_1X_2} + \Delta_{2,X_1X_2}$$

$$\lambda_{i,X_1X_2} = \frac{\Delta_{i,X_1X_2}}{\Delta_{X_1X_2}}.$$

Then express the nearest point by a convex combination of $X_1$and X_2

$$\upsilon = \lambda_{1,X_1X_2} X_1 + \lambda_{2,X_1X_2} X_2 \tag{5}$$

In (5), the meaning of ν is the nearest point on the line segments connecting X_1, X_2. Because the correct point ν is between X_1 and X_2, ν should be the point X_1 (or X_2) if λ_1(or λ_2) is negative. λ_1 and λ_2 cannot be negative at the same time, so there are only three variations of sign of λ_i in two-dimensional space.

(d). How to refine the set V_i (about refine_set()) :

The refinement method used by Gilbert, Johnson, and Keerthi[1] is to investigate every subsets of V_i , or the set $V_i \cup \{S_i\}$ in our modified algorithm Fig.3, and then verify them with the three conditions in (5). Eventually, the unique affinely independent subset will be found. However, the process is very complicated. Instead, our modification presented below is a simpler and geometrically clear discard-and-add process.

Firstly, the process starts from two initial points and adds a third point S_i in the way that can speed up computation. Then keep the two points that form the proper edge to refine the set V_{i+1}. In brief, we regularly choose a triangle and then discard a vertex of the triangle and add another point to construct a new triangle.

Now, which point is to be discarded? The point that has maximum distance is not the correct one. The method used here is quite similar to that of determining an affinely independent set. Refer to Fig. 5 for illustration. Select two points from the set $V_i \cup \{S_i\}$ which contains three points. Its three subsets are three edges of the triangle formed by the set $V_i \cup \{S_i\}$. A line-by-line testing can decide which line of edge can separate the origin of the TC-space and the remaining point, then discard the remainder. The separating edge is precisely the edge of the triangle formed by V_i which contains (or represents) ν_i.
There might be more than one subset corresponding to the separation condition we just mentioned. The simplest way is to discard the farthest point once the non-unique situation happens.
Remark: The reason why three points suffice in our modification is obviously revealed by the Caratheodory theorem[1]. Suppose X belongs to the translate of a linear space . Without loss of generality, assume X contains no more than (dimX +1) points. Thus, the set V_i in our algorithm will contain two points.

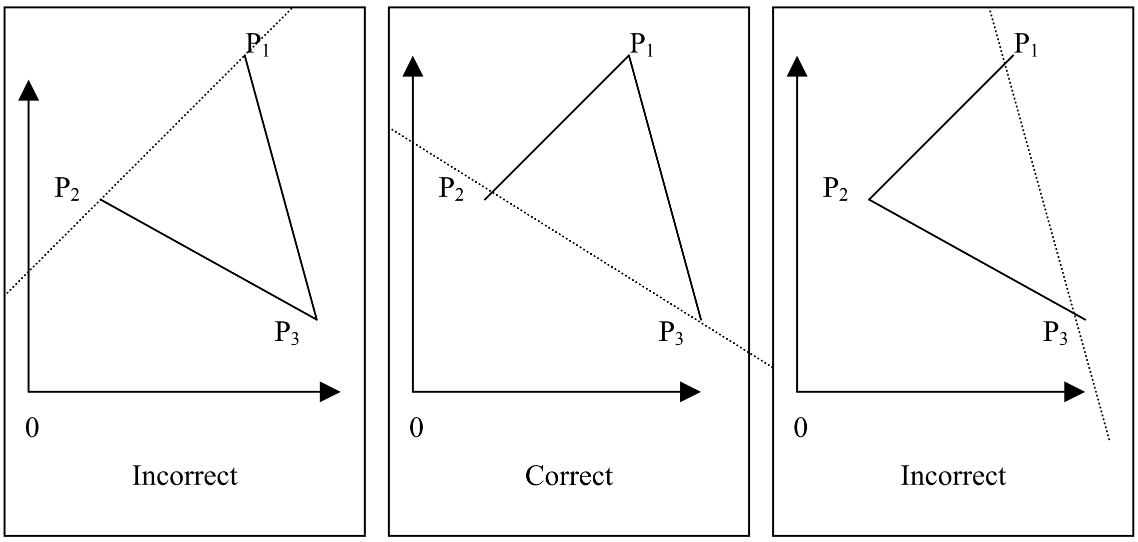

Fig. 6 Decide the vertex to be discarded.

3.2 Three-dimensional case (Fig. 7)

(a). Main ideas:

The main idea for three-dimensional case is the same as two-dimensional case, but there are some important changes should be made. From Caratheodory theorem, the triangles we update in two-dimensional space now change as tetrahedrons in 3D space. For a tetrahedron, three faces (triangles) intersect at one vertex, and a face has three vertices. Thus, there should be three points in the initial set. The way to find ν in function nu_compute() is changed from seeking an edge of a triangle to finding a face of a tetrahedron. Moreover, where ν is either on a face, an edge, or a vertex is examined after we determine the subsets of $V_i \cup \{S_i\}$.

(b). refine_set() in three-dimensional space:

The equation used to refine the set V_i should be replaced by one representing a plane in three-dimensional space. Four non-coplanar points form a tetrahedron. Three points

```
The Algorithm in 3D:
$V_0 \leftarrow$ initial_set($Z_1, Z_2, Z_3$) ;
$i \leftarrow 0$ ;
Repeat{
    $\nu_i \leftarrow$ nu_compute($V_i$) ;
    $(S_i, H_i) \leftarrow$ support_functions($-\nu_i$) ;
    $V_{i+1} \leftarrow$ refine_set($V_i, S_i$) ;
    $i \leftarrow i + 1$ ;
} until ( $\nu_i \bullet \nu_i + H_i = 0$ )
```

Fig.7 GJK algorithm with our modification in 3D case.

determine a face of the tetrahedron. In 3D case, we compute the normal vector of the plane formed by the three points, then find out which plane can separate the origin and the remaining (the fourth) point.

Let the four vertices of the tetrahedron be P_1, P_2, P_3, P_4. A plane that separates the origin and the vertex, say P_4, can be found by the following way. The plane formed by the three points P_1, P_2, P_3, is given by the equation

$$f(x, y, z)=Ax+By+Cz+K=0$$

where the normal vector $(A, B, C) = (P_2 - P_1) \times (P_3 - P_1)$,

$$K = -(A P_{11} + B P_{12} + C P_{13}).$$

Define

$$N = (A\,P_{41} + BP_{42} + C\,P_{43}) + K, \qquad (3)$$

where $P_i = (P_{i1}, P_{i2}, P_{i3})$.

Then the plane offers quick testing of separation if $Z \bullet N < 0$. Discard the subsets containing the farthest point when more than one subset satisfying the separation condition.

(c). nu_compute() in three-dimensional space:

The goal of the function is to find out the nearest point ν on the plane, which is determined by the three points in V_i, to the origin of the TC-space. From (2) (or Theorem 3 in Gilbert, Johnson, and Keerthi[1]), we have:

$$\Delta_{1,P_1P_2P_3} = \Delta_{1,P_2P_3} \cdot (P_2 \cdot P_2 - P_2 \cdot P_1) + \Delta_{2,P_2P_3} \cdot (P_3 \cdot P_2 - P_3 \cdot P_1),$$

$$\Delta_{2,P_1P_2P_3} = \Delta_{1,P_1P_3} \cdot (P_1 \cdot P_1 - P_2 \cdot P_1) + \Delta_{2,P_1P_3} \cdot (P_3 \cdot P_1 - P_3 \cdot P_2),$$

$$\Delta_{3,P_1P_2P_3} = \Delta_{1,P_1P_2} \cdot (P_1 \cdot P_1 - P_1 \cdot P_3) + \Delta_{2,P_1P_2} \cdot (P_2 \cdot P_1 - P_3 \cdot P_2),$$

$$\Delta_{P_1P_2P_3} = \Delta_{1,P_1P_2P_3} + \Delta_{2,P_1P_2P_3} + \Delta_{3,P_1P_2P_3},$$

$$\lambda_{i,P_1P_2P_3} = \frac{\Delta_{i,P_1P_2P_3}}{\Delta_{P_1P_2P_3}} \qquad (4)$$

$\lambda_{i,P_1P_2P_3}$ will be negative if ν is on the edge of the tetrahedron. There are seven kinds of variations in the sign of $\lambda_{i,P_1P_2P_3}$ which affect the geometric location of ν in the plane $P_1P_2P_3$. This is shown in Table 1.

(A)	All λ are positive	1 case	ν On Face
(B)	One of λ is negative, the others are positive	3 cases	ν On Edge or Vertex
(C)	One of λ is positive, the others are negative	3 cases	ν On Edge or Vertex
(D)	All λ are negative	None	Not Exist

Table 1

In Situation (A), no more modification is needed.

Situation (B) is shown in Fig. 8(a). Each case may have three sub-cases in this situation. Assuming $\lambda_{3,P_1P_2P_3}$ <0 with respect to the vertices P_1, P_2, and P_3, there are three cases:

(i)Origin in region(i): $\lambda_{2,P_1P_2} < 0$. Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (1, 0, 0)$

(ii)Origin in region(ii): λ_{1,P_1P_2}<0. Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (0, 1, 0)$

(iii)Origin in region (iii):

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (\lambda_{1,P_1P_2}, \lambda_{2,P_1P_2}, 0)$

If $\angle P_3P_1P_2 > 90°$, region (i) doesn't exist.
If $\angle P_3P_2P_1 > 90°$, region (ii) doesn't exist.

Now we investigate Situation (C) in Table 1. Situation (C) implies that at least one vertex is irrelevant to the determination of the nearest point. Each case in this situation has five sub-cases, as indicated by Fig.8(b). Assuming $\lambda_{3,P_1P_2P_3} > 0$ with respect to the vertices P_1, P_2, and P_3:

(i)Origin in region (i): $\lambda_{2,P_1P_3} < 0$.Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (1, 0, 0)$

(ii)Origin in region (ii): $\lambda_{1,P_1P_3} > 0$ and $\lambda_{2,P_1P_3} > 0$. Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (\lambda_{1,P_1P_3}, 0, \lambda_{2,P_1P_3})$

(iii)Origin in region (iii): $\lambda_{1,P_1P_3} < 0$ and $\lambda_{1,P_2P_3} < 0$. Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (0, 0, 1)$

(iv)Origin in region (iv): $\lambda_{1,P_2P_3} > 0$ and $\lambda_{2,P_2P_3} > 0$. Then

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (0, \lambda_{1,P_2P_3}, \lambda_{2,P_2P_3})$

(v)Origin in region (v): $\lambda_{2,P_2P_3} < 0$.

$(\lambda_{1,P_1P_2P_3}, \lambda_{2,P_1P_2P_3}, \lambda_{3,P_1P_2P_3}) \leftarrow (0, 1, 0)$

When $\angle P_1P_3P_2 < 90°$, only sub-case (iii) is possible.

Finally, in all situations the nearest point ν is computed by the unified equation:

$$\upsilon = \lambda_{1,P_1P_2P_3} P_1 + \lambda_{2,P_1P_2P_3} P_2 + \lambda_{3,P_1P_2P_3} P_3$$

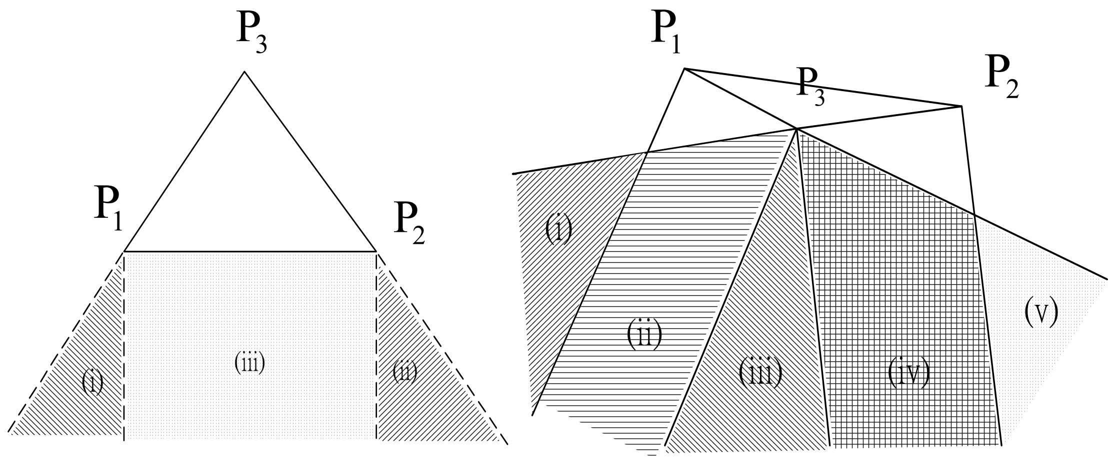

Fig.8(a) Fig.8(b)

Fig.8 Possible geometric relations between the origin and triangle $P_1P_2P_3$ in situations (B) and (C). The filled regions represent various positions of the projection of the origin on plane $P_1P_2P_3$.

IV. Implementation and Comparisons

In this section, for various shapes of objects simulations are performed to compare the improvements made by us and by Cameron[2], in static and tracking cases. In particular, efficiency comparisons of collision-free motion planning of redundant robots are also made, which is important for on-line application of distance computation algorithms. The input data are the vertices of two polyhedra.

4.1 Cameron's Enhancement[2]

From our experience in implementing the GJK algorithm shown in Fig. 1, the computation simulation of Fig. 9. It shows that the computation of support function consumes a big percentage of overall computation time. Therefore, the key to faster computation of distances between two convex bodies is to improve the computation speed of the support function. Stephen Cameron[2] achieved the improvement. The enhancement exploits the adjacency feature of vertices (two vertices are adjacent if they are connected by an edge) and is now described as follows. Given a convex polyhedron, an initial vertex of it, and the edge connection data of its vertices, the goal is to find a "supporting vertex". The procedure starts by finding the one with the largest inner product from the initial vertex and its vicinities. If the search result happens to be the initial one, the search is terminated and the answer is obtained owing to convexity property. Otherwise, the obtained vertex is set as a new initial and the process is repeated. In a finite number of iterations, due to convexity, we'll find the supporting vertex without the need of evaluating inner product for every vertex of the polyhedron.

The supporting vertex searching sequences form a succession of edges of TCSO:

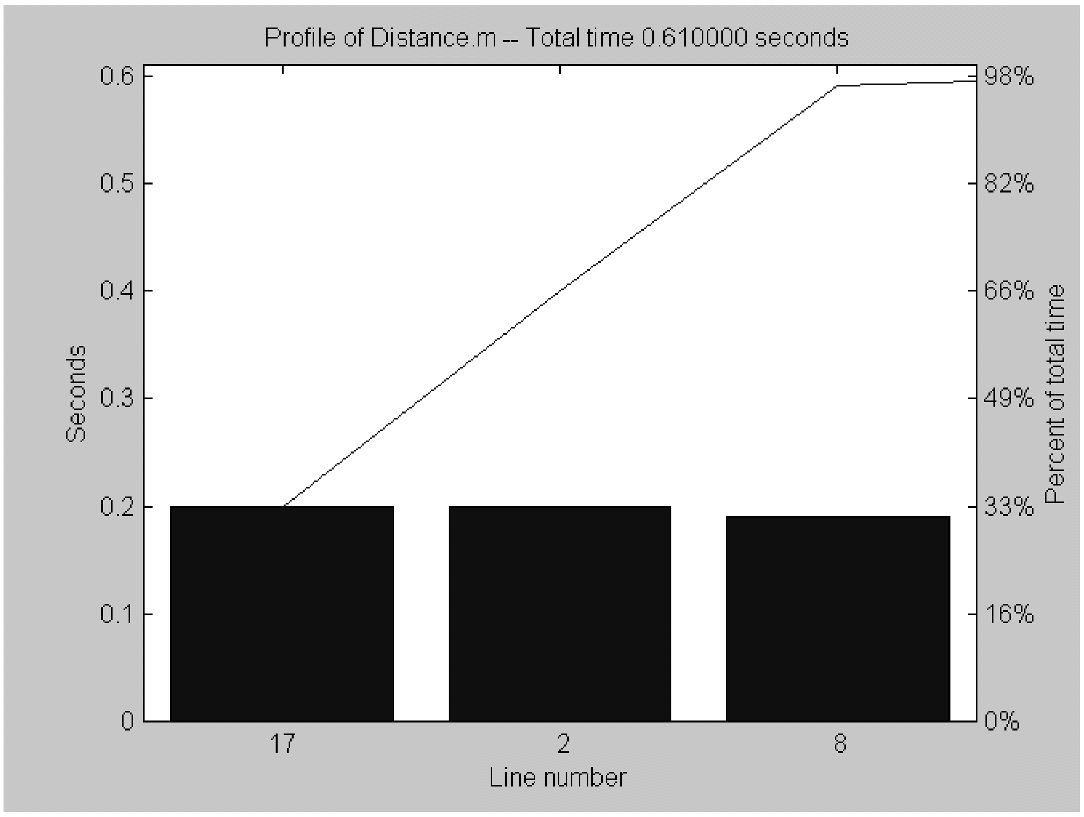

Fig. 9: The lines 17, 2, and 8 in program evaluate the support function.

originating from the initial vertex, going along the edge which corresponds to the direction vector which evaluates the support function, and stopping at the supporting vertex (as Fig. 10 shows). In addition, the vectors with respect to which we evaluate the support function, i.e., -ν_i, roughly point from the polyhedron to the origin. Therefore, all supporting vertices lie on the sides of the polyhedron closer to the origin. Thus, in most cases, only a few inner product evaluations are needed and this greatly reduces the computation time by using the previous evaluation result as the new initial vertex for next search. As a whole, the procedure achieves the O(1) improvement of GJK algorithm.

4.2 The Edge Connection Data

In implementing Cameron's improvement, a geometric characterization of a convex polyhedron by its faces and edges, in addition to vertices, are needed. For this, a method is developed to find the edge connection data of the vertices for convex polyhedron (see Appendix), without the use of the so-called edges graph. The edges data are found from the faces data, which specify the sets of vertices that form a face (polygon). To obtain the faces data, we first test all the combinations of three vertices to see which ones determine the

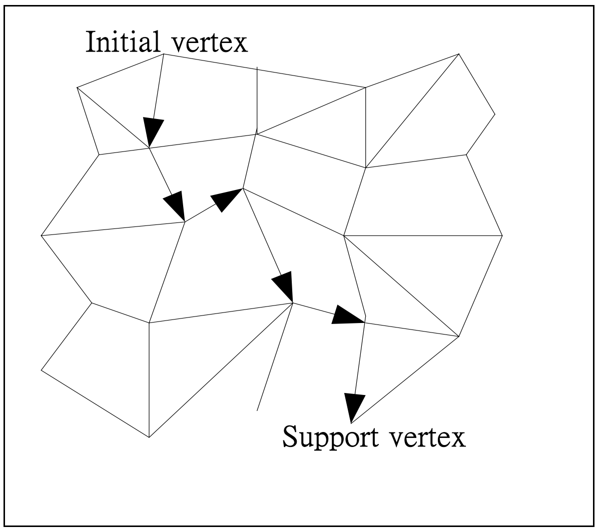

Fig. 10 Searching path for the supporting vertex

hyperplanes with the property that all the other vertices and the centroid of the polyhedron are in the same half-space, i.e., support the polyhedron. Then, we consider combinations that determine the same hyper-planes and integrate them with sets of more than three vertices, which then uniquely and completely determine the faces of the polyhedron. The edges of a polyhedron are, by definition, the union set of the edges of each face; therefore, the edges data can be constructed by tracing along the boundary of every face. As a matter of fact, this method is quite time-consuming. It is found that to compute the edges data for a polyhedron with hundreds of vertices on a 400-MHz Pentium II PC, a few hours to half-day of computation time is required. However, it is quite useful for visualization of a convex polyhedron, or, as we have seen, for faster distances computation. Though time-consuming, the computation of edges data from a set of vertices can be taken as *a priori* information (i.e. set-up time) for the problem.

4.3 Computing Distances between Objects in Motion

When a pair of objects between which the distance is computed is moving, it is a tracking problem. As is usually the case, in tracking problems the motion is so slow that the relative position does not change significantly within one time-step of computation and the new supporting vertex can be the same as or quite close to the last one. As a result, if the supporting vertices are cached, the search for the supporting vertex can be finished along few edges by using the previously cached one as an initial vertex. It makes the speed superiority of Cameron's enhancement over the original GJK more remarkable.

For continuous motions of moving objects or for small motions, the Euclidean distance (or the relative position) and thus the closest features do not vary a lot between two time steps. The subset of vertices that determined the nearest point of the TCSO in last iteration can be used to initialize the current distance calculation loop. We store the supporting vertex and its neighbors that determine the minimum distance in the previous time step and use these vertices to re-compute the distance of two objects at the new time step. In this way, the searching of supporting vertices from time to time can speed up by using last supporting vertices and its neighbors as new initials, in case the objects are moving slowly, without seeking from scratch.

4.4 Comparisons

For comparison study, consider the following four algorithms for computing the distances between two static convex bodies approximating the unit balls in 3-D space: the original GJK, GJK with our modification, GJK with Cameron's, and GJK with ours plus Cameron's modifications. All of them are implemented in MATLAB 5.2 on a 266-MHz Pentium II PC under Windows98. A pair of polyhedra of varying complexity for distance computation study is systematically produced,

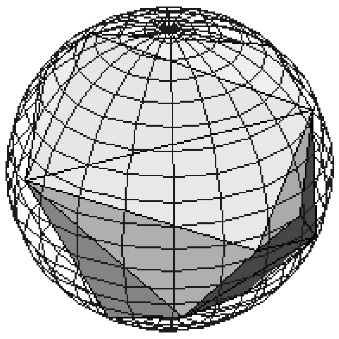

Fig. 11. The construction of a polyhedron from a unit ball

each vertex of which is generated from a set of random points on a unit ball (see Fig. 11) with translation. With increasing number of points, the polyhedron approximates the unit ball more accurately. The experimental results are shown in Table 2 and collectively in Fig. 12. In Table 2, the most efficient results are marked by a *. It reveals that when the total number of vertices increases, the combined algorithm (ours plus Cameron's modification) becomes more efficient.

Next consider the tracking case where there is relative motion between the two polyhedra. To describe relative motion, it is sufficient to consider one polyhedron to be static and the other polyhedron to be moving. The polyhedra are randomly generated as static cases; the unit balls from which two polyhedra are generated are $5\sqrt{3}$ units of length apart in simulation. The relative configuration of two polyhedra, which consists of a relative position and a relative orientation, is as follows. The relative position between two polyhedra varies slowly along a path composed of 10 randomly generated positions. Specifically, at each time-step, the translation of the moving body is a translation by fixed short distance (1 unit of length) along a random direction. The relative orientation is a rotation of a fixed small angle (10 degrees in simulation) about its center around a random direction (clockwise or counterclockwise). This is shown in Fig. 13, where the distances are deliberately shortened so that the incremental change in the relative position can be observed. The comparisons of the combined algorithm and the original GJK algorithm for tracking case are shown in Table 3. It can be seen that with the supporting vertex and its neighbors in the last run stored, the efficiency of distance computation improves significantly.

(Unit: ms)

Number of Vertices (Object1 /Object2)	8/8	10/20	20/30	30/40	40/50
The Original GJK	16*	30*	80	105	210
Our Modification	17	30*	70*	100*	205
Cameron's Modification	30	55	80	110	175
Ours plus Cameron's Modification	31	60	75	105	150*

Number of Vertices (Object1 /Object2)	50/60	100/120	150/180	200/240	250/300
The Original GJK	95	445	440	380	480
Our Modification	100	480	430	385	480
Cameron's Modification	85*	250	145	250*	220*
Ours plus Cameron's Modification	90	220*	130*	255	220*

Table 2

The above computation results suggest how to compute the distance between the polyhedra efficiently. Cameron's enhancement, which doesn't cause the computation time grow linearly with the total number of hull points if the adjacency information of vertices is available, is significant, especially in cases where the total number of vertices is large. The results also demonstrate our modification provides an alternative and competitive approach to distance calculation. By examining the performance of the algorithm with and without our modification for a specific problem, a more efficient combined algorithm can be adopted. Moreover, clear geometric meaning facilitates the understanding and implementation of the algorithm.

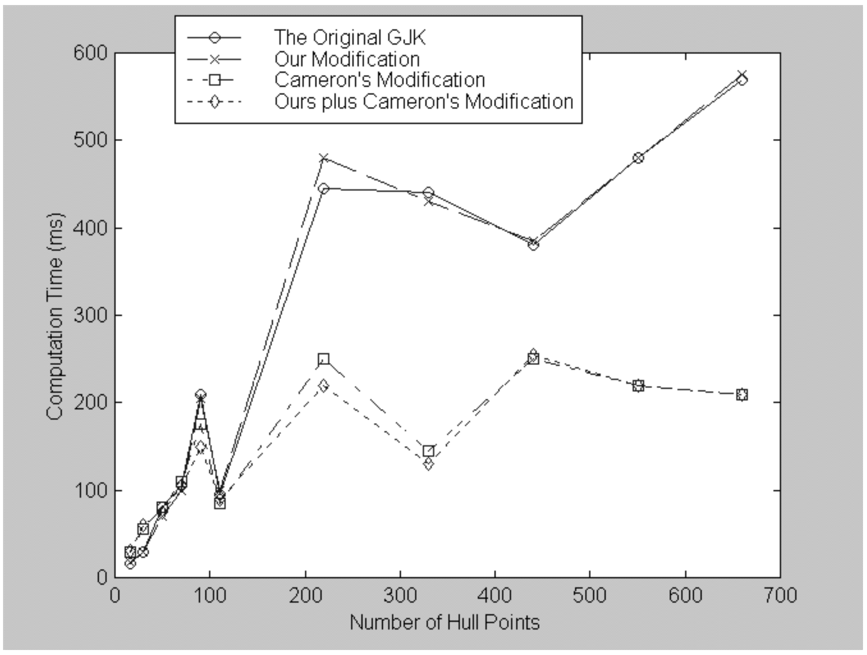

Fig. 12: The simulation result

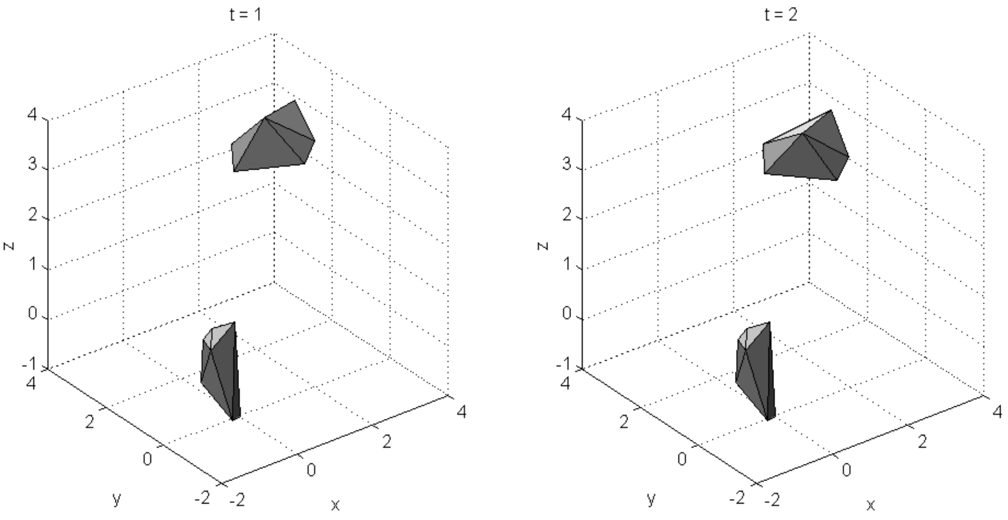

Fig. 13 Small motion is introduced between consecutive time steps.

(Unit: ms)

Number of Vertices (Object1 /Object2)	10/20	20/30	30/40	40/50
The Original GJK	666	1726	946	2096
Ours plus Cameron's Modification	358	424	462	522

Number of Vertices (Object1 /Object2)	50/60	100/120	150/180	200/240
Ours plus Cameron's Modification	1250	2650	6074	5250
The Enhanced Version	458	582	668	800

Table 3

4.4 Application to collision-free trajectory planning of redundant robots

Given the trajectory $\mathbf{x}_e(t)$ of the end-effector of a redundant robot prescribed by the operator, the trajectory planning problem of the robot is to plan the various joints trajectory $\mathbf{q}(t)$ of robots that achieve the desired end-effector motion and avoids collisions with obstacles. The inverse kinematics of redundant robots is described by velocity inversion equation of the following form[21]:

$$\dot{\mathbf{q}} = \mathbf{J}_e(\mathbf{q})^+ \dot{\mathbf{x}}_e(t) + \mathrm{k}(\mathbf{I} - \mathbf{J}_e(\mathbf{q})^+ \mathbf{J}_e(\mathbf{q}))\nabla \mathrm{H}(\mathbf{q}) \qquad (6),$$

where $\mathbf{q}$ is the vector of joint angles, $\mathbf{J}_e$ and $\mathbf{x}_e$ are the Jacobian matrix and the position of the end-effector, respectively; k is a negative constant, and H is an objective function to be minimized. For the purpose of collision avoidance, H is chosen to be the sum of all the artificial potential fields (U_{ij}) built by each obstacles (numbered by i) and each links (numbered by j)

$$\mathrm{H}(\mathbf{q}) = \sum_{i,j} \mathrm{U}_{ij}(\mathbf{q})$$

These fields are assigned non-negatively such that their values remain zero if the links are at a distance away from the obstacles and approach infinity as the links approach the obstacles[5]:

$$\mathrm{U}_{ij}(\mathbf{q}) = \begin{cases} \frac{1}{2}\left(\frac{1}{\mathrm{d}_{ij}(\mathbf{q})} - \frac{1}{\mathrm{d}_0}\right)^2, & \mathrm{d}_{ij}(\mathbf{q}) \le \mathrm{d}_0 \\ 0, & \mathrm{d}_{ij}(\mathbf{q}) > \mathrm{d}_0 \end{cases}$$

where d_{ij} , is the distance between obstacle i and link j, determines the obstacle-link pair's contribution to potential field, d_0 represents the limit distance of the potential influence.

The gradient of the artificial potential field H can be derived as[21],

$$\nabla \mathrm{H}(\mathbf{q}) = \sum_{i,j} \nabla \mathrm{U}_{ij}(\mathbf{q}),$$

$$\nabla \mathrm{U}_{ij}(\mathbf{q}) = \begin{cases} \left(\frac{1}{\mathrm{d}_0} - \frac{1}{\mathrm{d}_{ij}}\right)\frac{1}{\mathrm{d}_{ij}^2}\nabla \mathrm{d}_{ij}, & \mathrm{d}_{ij}(\mathbf{q}) \le \mathrm{d}_0 \\ 0, & \mathrm{d}_{ij}(\mathbf{q}) > \mathrm{d}_0 \end{cases},$$

$$\frac{\partial \mathrm{d}_{ij}(\mathbf{q})}{\partial q_k} = \frac{(\mathbf{v}_i - \mathbf{v}_j)^{\mathrm{T}}}{\left\|\mathbf{v}_i - \mathbf{v}_j\right\|}[\mathbf{A}_1^0 \ldots \mathbf{A}_{k-1}^{k-2}][\mathbf{Q}_k][\mathbf{A}_{k-2}^{k-1} \ldots \mathbf{A}_0^1]\mathbf{v}_i \qquad (7)$$

(the kth component of the gradient ∇d_{ij}),

where for every pair of obstacle-link, there corresponds a closest point ν_i on the obstacle and a closest point on the link ν_j, and the distance d_{ij} between those two points; $\mathbf{A}_k^{k-1}(q_k)$

is the Denavit-Hartenberg homogeneous transformation matrix from link $k-1$ to link k, and

$$\mathbf{Q}_k = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

for robots with rotary joints.

For computation of the distance and closest points information required in (7), suppose all links and obstacles can be properly modeled by convex polyhedra. Thus, the distance of pairwise link and obstacle can be readily calculated with the enhanced GJK algorithm. As for the nearest points, they can be computed by recording vertex of the TCSO is composed of which two vertices of the polyhedra. The TC-space nearest point found by GJK algorithm is a convex combination of TC-space vertices which can be transformed back to the corresponding vertices (or nearest point) of polyhedra.

By solving inverse kinematics (6) and plotting the robot and the obstacles at selected instants with MATLAB 5.3, some collision-free trajectories have been successfully planned by the distance based planning method to guide the robot to achieve its goal. To know its efficiency, a typical result is presented as follows. Consider a 3-D workspace contains a 4-*dof* articulated robot with link lengths 2, 1, 0.6, and 0.4 and all rotary joints; and two stationary pyramid-shaped obstacles located in the workspace. A hexadecagonal cylinder is used to model each link and joint. Since the capability of collision avoidance is provided by redundancy or, mathematically, by the augmentation of the second term in the velocity equation (6), a larger k magnitude tends to render the collision avoidance ability of robot more visible. For simulation study, let the initial pose (joint angles: 0, 90, -150, and -30) of the robot arm, the start position (0, -0.7877, 3.1918) and destination position (0, -0.0804, 2.3) of the end-effector, d_0 (0.5), and the value of k (-1) be properly specified. The velocity of the end-effector was fixed at 0.1 in magnitude; it remained in the -y direction until t = 6.7, when it began to move in the z direction instead before the stop at t = 16.7. A simulation with real-time visualization is performed and several frames of the motion are shown in Fig.14. In addition, the run-time profile of the simulations (Fig.15) shows that the distance-computing subroutine is the most time-consuming part (about 70% of total execution time) in planning a collision-free trajectory. It also shows that combination of Cameron's enhancement on the evaluation of support function and our enhancement on the initialization of GJK in the tracking case achieves the best efficiency of GJK algorithm. The computational cost depends on the number of obstacles and links and the proximity of obstacles and links. It must be fast enough to be useful in on-line applications.

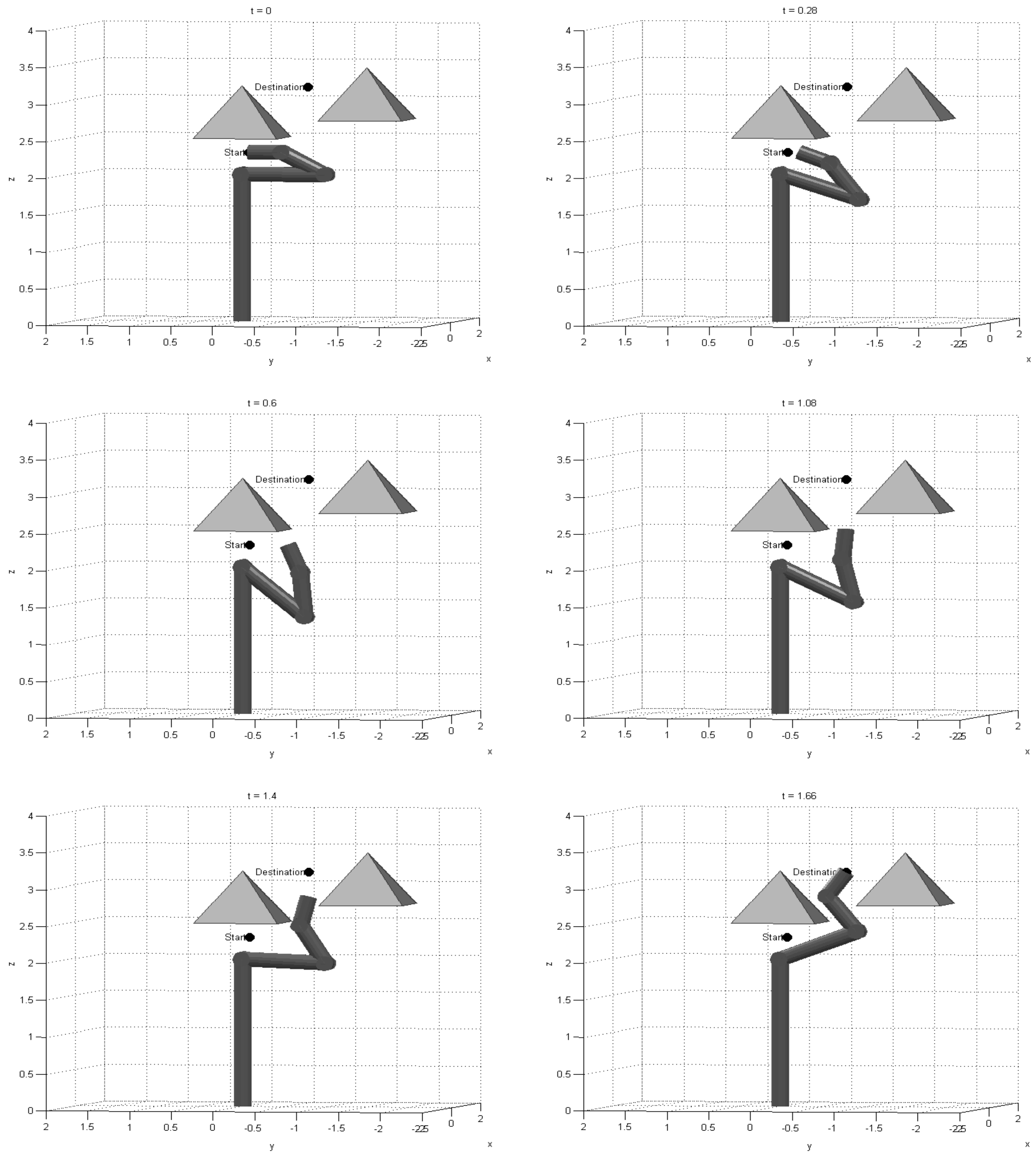

Fig. 14. Collision-free motion of 4-link redundant robot

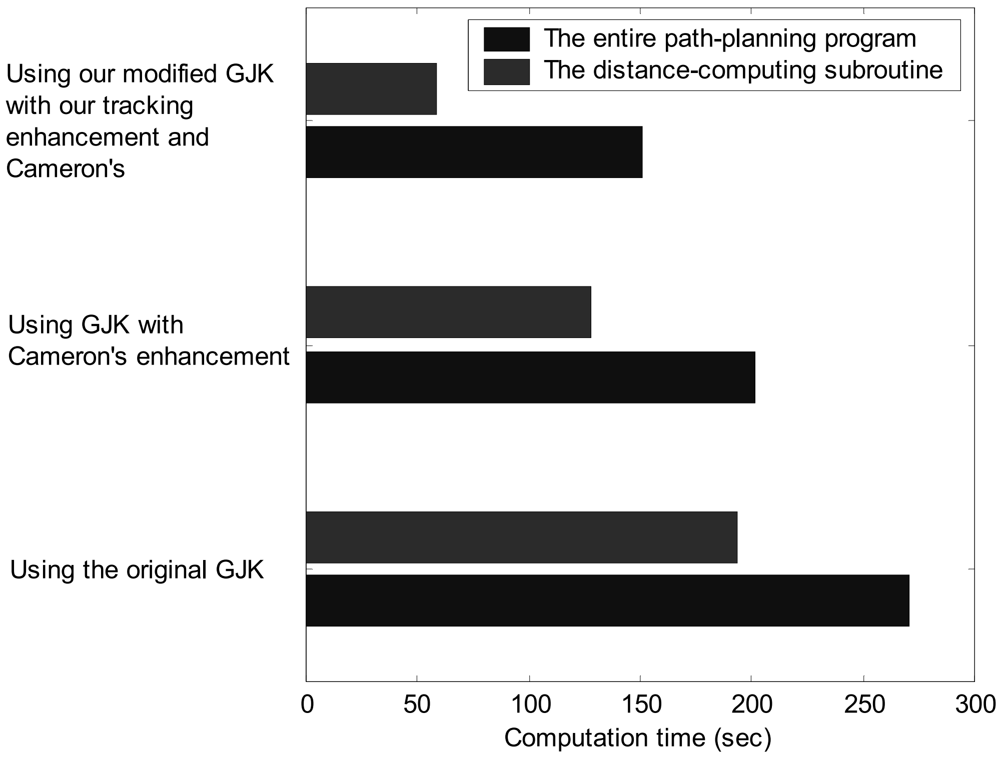

Fig. 15. Run-time profile comparisons of path-planning.

V. Conclusion

By examining the steps of Distance Subalgorithm in GJK algorithm from a geometric point of view, we have described possible modifications of the steps in the GJK algorithm that make the computation of Euclidean distance between convex polyhedra easily realized. The improvement proceeds by updating the coefficient λ, instead of verifying $\Delta(Y_s)$ for each subset in Distance Subalgorithm, and these induce an explicit triangulation of the TCSO boundary. By the numbers of elements in the final set in our modified algorithm, the nearest point, together with the information about the feature of the nearest point is on a vertex, an edge or a face of TCSO can be provided.

A typical simulation (e.g., Fig. 9) indicates that computation of support function is the most time-consuming part in GJK algorithm[2, 9, 15]. However, we have demonstrated experimentally that our modifications still preserve the almost linear time complexity of GJK algorithm. In combination with Cameron's modifications for speedup of support function computation, the favorable comparisons have also been shown. Since in applications the distance between two objects needs to be updated from time to time, every possible enhancement of distance computation procedure can speed up the repetitive process as the time goes on. This is shown more clearly by the involved a couple of pairwise distance computations in collision-free motion planning of redundant robots in a typical simulation of 4-link robot in 3D space in Sec. 4.4. This means that our contribution to GJK algorithm is not only on long-time computational efficiency in cases distance computation is repeated but also on a more clear understanding of its steps by revealing their geometric meanings.

Acknowledgment

This work was supported by National Science Council of R.O.C. under contract NSC 88-2212-E-001-001.

Ⅵ. References

1. E. G. Gilbert, D. W. Johnson, and S.S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Trans. Robot. Automation*, vol. 4, pp. 193-203, Apr.1988.

2. Stephen Cameron, "A comparison of two fast algorithms for computing the distance between convex polyhedra," *IEEE Trans. Robot. Automation,* vol. 13, no 6, pp. 915-920, Dec.1997.

3. Bobrow, J.E., "Optimal robot path planning using the minimal-time criterion," *IEEE Journal of Robotics and Automation*, vol.4, no. 4, pp.443-450, Aug.1988.

4. Quinlan, Sean. "The Real-Time Modification of Collision-Free Path," Ph.D. Thesis, Stanford University, 1994.

5.Khatib, O., "Real-time obstacle avoidance for manipulators and mobile robots", *The International Journal of Robotics Research*, vol. 5, no.1, pp.90-98, Spring 1986.

6. M. Lin and J. Canny, "A fast algorithm for incremental distance calculation," in *IEEE Int. Conf. Robot. Automat.*, Sacramento, CA, pp 1008-1014, April 1991.

7. Chung Tat Leung, Kelvin. "An efficient collision detection algorithm for polytopes in virtual environments," M. Phil. Thesis, The University of Hong Kong, 1996.

8. S. Cameron, "Enhancing GJK: computing minimum and penetration distances between convex polyhedra," in *IEEE Int. Conf. Robot. Automat.* , Albuquerque, NM, April. 1997, pp 3112-3117.

9.Y. Sato, M. Hirata, T. Maruyama, and Y. Arita. "Efficient collision detection using fast distance-calculation algorithms for convex and non-convex objects," in *IEEE Int. Conf. Robot. Automat*, Minneapolis, Minnesota, pp. 771-778, April 1996.

10. Elon Rimon, Stephon P. Boyd, "Obstacle collision detection using best ellipsoid fit," *Journal of Intelligent and Robotic Systems*, vol. 18, pp 105-126, 1997.

11. David E. Johnson, Elaine Cohen, "A framework for efficient minimum distance computations," in *IEEE Int. Conf. Robot. Automat.*, pp. 3678-3684, May 1998.

12. R. T. Rockfellar, *Convex Analysis*. Princeton, NJ: Princeton Univ. Press, 1970.

13. J.E. Bobrow, "A direct minimization approach for obtaining the distance between convex polyhedra," *The International Journal of Robotics Research*, vol.8, pp.65-76, 1989.

14. C. J. Ong and E. G. Gilbert, "Growth Distances: New Measures for Object Separation and Penetration," *IEEE Trans. Robot. Automat.,* vol. 12, No 6, pp. 888-903, 1996.

15. C. J. Ong and E. G. Gilbert, "The Gilbert-Johnson-Keerthi distance algorithm: a fast version for incremental motions," in *IEEE Int. Conf. Robot. Automat.*, pp. 1183-1189, Albuquerque, New Mexico, April 1997.

16. E. G. Gilbert and C.-P. Foo, "Computing the distance between general convex objects in three-dimensional space," *IEEE Trans. Robot. Automat.,* vol.6, No 1, pp. 53-61, 1990.

17. S. Zeghloul, P. Rambeaud and J.P. Lallemand," A fast distance calculation between convex objects by optimization approach," in *IEEE Int. Conf. Robot. Automat.*, pp. 2520-2525, Nice, France, May 1992.

18. E. G. Gilbert and D.W. Johnson, "Distance functions and their application to robot path planning in the presence of obstacles," *IEEE J. Robot. Automat.,* vol.1, No 1, pp. 21-30, 1985.

19.G. Hurteau and N. F. Stewart, "Distance calculation for imminent collision indication in robot system simulation," *Robotica,* vol. 6*,* pp. 47-51, 1988.

20. C. J. Ong and E. G. Gilbert, "Robot path planning with penetration growth distance," *J. Robotic Systems*, vol. 15, no. 2, pp.57-74, 1998.

21. C. Y. Chung, B. H. Lee, and J. H. Lee, "Obstacle avoidance for kinematically redundant robots using distance algorithm," *Proc. IROS' 97*, pp. 1787-1793, 1997.

22. H. Edelsbrunner, "On computing the extreme distances between two convex polygons," *J. Algorithms*, vol.6, pp.515-542, 1985.

23. D.P.Dobkin and D.G. Kirkpatrick, "A linear algorithm for determining the separation of convex polyhedra," *J. Algorithms*, vol.6, pp.381-392, 1985.

24.F.Chin and C.A. Wang, "Optimal algorithms for the intersection and minimum distance problems between planar polygons," *IEEE Trans. Computer*, vol.C-32, pp.1203-1207, 1983.

APPENDIX

A Method to Construct Face and Edge Data from a Vertex Set of Convex Polyhedron

A face of a polyheron has associated with it a plane in which the face lies and a set of vertices. In practice, we are given the vertices of a convex polyhedron, the information about which vertices determine a face is usually required, for example when the polyhedron is visualized in a computer graphics environment. In this appendix, a method to acquire such information, which can be implemented in the form of a function block and applied to individual cases, is developed.

By definition, a face of a convex polyhedron is a polygon located on a unique support plane[12] of the polyhedron and whose vertices are all those of the polyhedron on the support plane. Since each support plane contains at least three vertices of the polyhedron, such planes can be found by enumerating all possible triples of vertices, each triple for one plane, and discarding those bearing no supporting property. Note that by this enumeration more than one triples may determine the same support plane, for one face may have more than three vertices. Thus, identifying each set of triples lying on the same plane and taking the union set of those triples gives a list of vertices determining each face.

To test whether a triple of three vertices determine a support plane or not, a simple criterion, which follows from the definition of a support plane, is provided here (Fig. 15). A plane supports the polyhedron if and only if there is no vertex such that the centroid lies on its opposite side. Thus, it suffices to check each of the remaining vertices and the centroid of the polyhedron lie on which side of the plane determined by the three vertices under test.

The edges of a polyhedron are the union set of the edges of all its faces. An edge is described by two neighboring vertices and connects two adjacent faces, therefore the edge can be constructed by tracing along the boundary of every face. Of course, finding the edges of a face or convex polygon is the 2-D equivalent to finding the faces of a convex polyhedron and can be done using similar procedures to the above; however, due to the geometric simplicity of this case, a more direct method is provided here. First, obtain the vector u_{ref} pointing from the center of the polygon to an arbitrary vertex. Then, sort in a counterclockwise (or clockwise) all the vertices according to the angle between u_{ref} and each of the vectors pointing from the center to all the vertices. Finally, two adjacent vertices (in the circular order) comprise the endpoints of an edge of the polygon. The method is summarized in Fig. 17.

```
/*
 * An Algorithm for Finding the Sets of Vertices Determining the
 * Faces of a Given Convex Polyhedron
  */
```

Function prototype(s):

```
boolean IsFace(a triple of vertices);
```

Definition of variable(s):

`F,` the output of algorithm, is the set of the triples, or quads, etc., of vertices determining the faces.

```
F = ∅;
for (each triple of vertices)
   if (IsFace(the triple))
      F = F ∪ the triple;
for (each element of F, say, f1)
for (each of the other elements of F, say, f2)
      if (p ∈ the plane determined by f1, ∀ p ∈ f2)
         modify F by changing f1 to f1 ∪ f2 and excluding f2;

boolean IsFace(a triple of vertices) {
   return (whether the centroid of the polyhedron does not lie
          on the plane determined by the triple
          and all the other vertices lie on the same side of
          the plane as the centroid does);
}
```

Fig. 16. Pseudo code of the face determination

```
for (each face){
   x = the vector pointing from the center of the face to a certain vertex,
        say, $v_1$;
   for (each of the other vertices, $v_i$)
      calculate the angle from x to the vector pointing from the center
      of the face to $v_i$;
      sort the vertices, starting from $v_1$, in the ascending
      order of those angles calculated for them;
   for (each of the sorted vertices)
      if (this is the last one)
         add the line segment connecting this vertex and the first one
         to the list of edges of the polytope;
      else
         add the line segment connecting this vertex and the next to
         the list of edges of the polytope;
}
```

Fig. 17. Pseudo code of the edge determination