On Enhancing GJK Algorithm for Distance Computation Between Convex Polyhedra: Comparison of Improvements

Shen-Po Shiang, Yu-Ren Chien, and Jing-Sin Liu*
Institute of Information Science
Academia Sinica
Nankang, Taipei, Taiwan 11529
R.O.C.

Abstract

The computation of Euclidean distance between two convex polyhedra is an important problem in robotics, computer graphics and animation. By geometric reasoning, we present an improvement of the well-known distance computation algorithm made by Gilbert, Johnson, and Keerthi (GJK). Some comparative simulations 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.

I Introduction

In robotics, computer graphics and animation, the Euclidean distance between detected object and obstacles around is an indispensable information when moving the object by manipulation robots or for realistic, 3D environment modeling. It is found applicable in many problems, such as intersection detection (14), collision avoidance (Khatib,1986), (Gilbert and Johnson, 1985), path planning (Bobrow, 1988), and path modification (Quinlan, 1994), where knowledge of the distance between the robot and its environment (or in general two moving objects) is crucial. In general, the problem of finding the distance between convex bodies (Gilbert et al. ,1988), (Bobrow, 1989), (Zeghloul et al., 1992) can be

* E-mail : liu@iis.sinica.edu.tw **Acknowledgment** This work was supported by National Science Council of R.O.C. under contract NSC 88-2212-E-001-001.

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 (Rimon and Boyd, 1997) used the L-J ellipsoid of the convex or non-convex objects to form a convex optimization problem for distance computation. The computation of the distance estimate (within a user's specified error) is shown to be an eigenvalue problem. Lin and Canny (Lin and Canny, Apr. 1991) provide 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 (Johnson and Cohen, 1998) for minimum distance computation also gives an efficient solution for objects described by different surface representations. Gilbert, Johnson, and Keerthi (GJK) presented a popular computation algorithm (Gilbert et al., 1988) in which the convex set is represented in terms of their support properties. For polytopes which is defined as th convex hull of a finite set of vertices, in particular, the properties can be easily obtained from their vertices. In order to find out the distance of two ploy hedra named K_1 and K_2, suppose the vertices of K_1 are $s_1, s_2, \ldots, s_n$ (n points), and the vertices of K_2 are $t_1, t_2, \ldots, t_p$ (p points), respectively. Then construct the set $K = \{ s_i - t_j, s_i \in K_1, t_j \in K_2\}$(also a polyhedron), where the elements are the relative translations of K_1 and K_2, in translational configuration (TC) space (Cameron, Dec 1997). There should be n*p points 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, called TCSO (translational C-space obstacle) (Gilbert et al.,1988), (Cameron, Dec 1997), (Cameron, Apr. 1997). The TCSO is also often called the Minkowski difference of the two polyhedra. The algorithm is extended in (Gilbert et al.,1988), (Lin and Canny, Apr. 1991), (Chung Tat Leung, 1996) and to more general convex objects (Gilbert and Foo, 1990) and is organized in (Cameron, Dec 1997), (Cameron, Apr. 1997). With some modification by Cameron (Cameron, Dec 1997), this method can also provide constant time updates for slowly moving polyhedra. (Chung Tat Leung, 1996) presented an efficient means of updating the Minkowski difference to create a collision detection method for convex polyhedra.

Section 2 is a summary of GJK algorithm. In Section 3, we present some modifications after carefully examining the steps of GJK algorithm. The changes we made would give more readability of the output yet, keep the results that confirm an almost-linear time complexity. In Section 4, the Cameron's enhancement (Cameron, Dec 1997) in the computation of support function is introduced for comparison study. A convenient method is developed to find out the neighbor points, so-called vicinity matrix, which is the information required in the "hill climbing" (Cameron, Apr. 1997), (Sato et al., Apr. 1996) 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

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(Y_s) ;
$(S_i, H_i) \leftarrow$ support_functions($-\nu_i$) ;
$V_{i+1} \leftarrow V_i \cup \{S_i\}$;
$i \leftarrow i + 1$;
} until ($\nu_i \bullet \nu_i + H_i = 0$)

Fig. 1 The GJK algorithm

It is essential to describe the GJK algorithm first before we introduce the algorithmic modifications that we make. The GJK algorithm is shown in Fig.1.

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 (Gilbert et al.,1988) $H_X : R^m \rightarrow R$ for a polyhedron X is the evaluation of inner products of a fixed vector with all vertices of polyhedron and looking for the maximum:

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

where $\eta \in R^m$ is a given vector, is the inner product. Define the support mapping $S_X : R^m \rightarrow X$ 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 and taking a new point S_i into consideration. The goal is achieved by checking if the points A, B in Fig. 2 satisfy

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

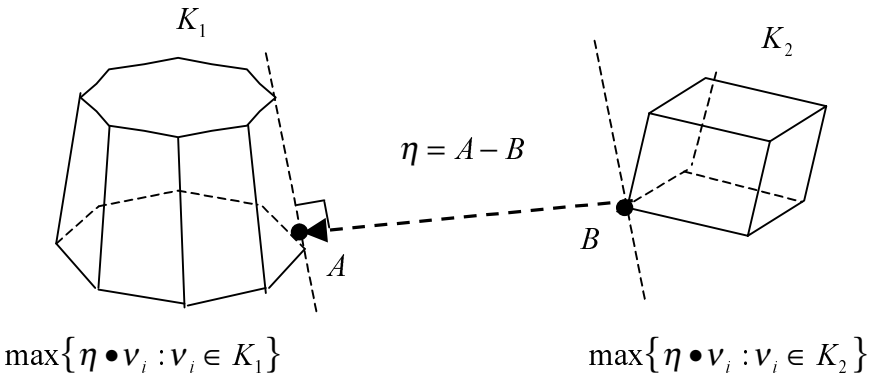

Fig. 2. Illustration of support function evaluation.

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 (Chung Tat Leung, 1996).

Definition (Rockfellar, 1970). A set of $m + 1$ points $\{b_0, b_1, \ldots, b_m\}$ is said to be affinely independent if

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

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 $\text{aff}\{b_0, b_1, \ldots, b_m\}$ 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 never needs to explicitly construct the TCSO. The key element of the approach is the algorithm for computing the distance between 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 (Rockfellar, 1970). 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 (Gilbert et al.,1988), (Sato et al., Apr. 1996) 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 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

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

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

(I_s' = The complement of I_s)

$$\Delta(Y_s) = \sum_i \Delta_i(Y_s) \quad \text{for all } i \in I_s.$$

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_1 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 et al.,1988)) 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 $j \in I_s$'. Because of the constraint of the numbers of elements in each set, there won't be more than 15 subsets that need to verify. Usually, Y_s is uniquely determined.

Ⅲ.The Modification

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. 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):

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(). (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 ν_i at step i (Gilbert et al.,1988), (Ong and Gilbert, 1997), (Gilbert and Foo, 1990)). 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 (Gilbert et al.,1988). 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 loop 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 ← initial_set(Z_1, Z_2) ;
 i ← 0 ;
Repeat{
    ν_i ← nu_compute(V_i) ;
   (S_i, H_i) ← support_functions(- ν_i) ;
   V_{i+1} ← refine_set(V_i, S_i) ;
   i ← i + 1 ;
} until ( ν_i • ν_i + H_i = 0)
```

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

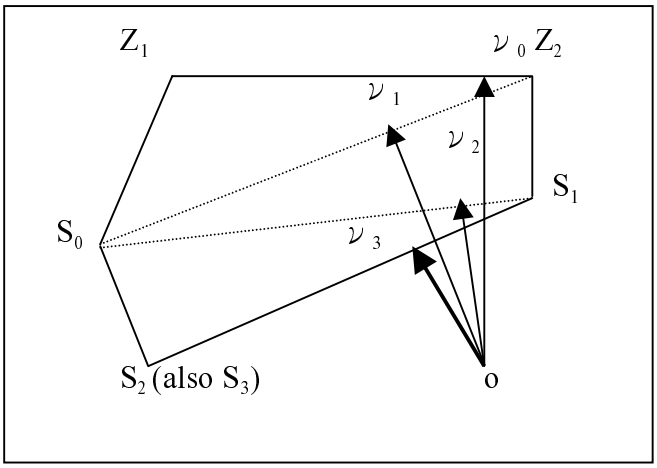

Fig.4. An illustration of GJK.

(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 (Gilbert et al.,1988), (Bobrow, 1989), (Zeghloul et al., 1992). 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 start partially in some particular points, the other point is distinct but arbitrary so that it can effectively lead us to two new points far apart after the calculation of support function. This is a good initial choice for efficient computation.

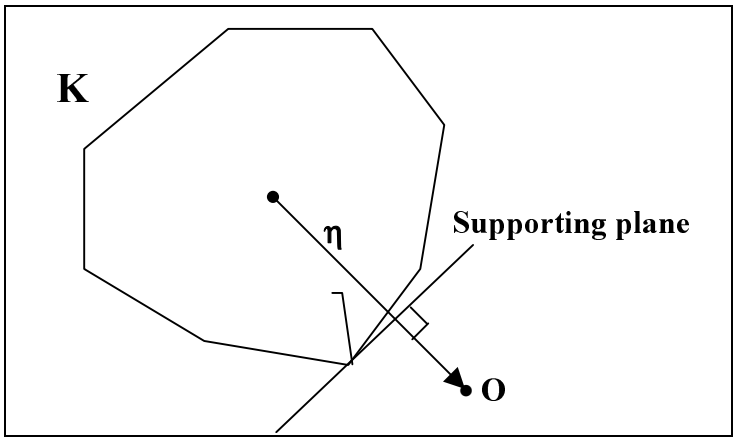

Fig. 5 The choice of initial point.

(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 to the origin of the TC-space (Gilbert et al.,1988), (Cameron, Dec 1997). It is also a vector that gives us the

direction from the origin to the nearest point on the TCSO. The idea of this function is mainly the same as that used in (Gilbert et al.,1988), 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 contained in the set V_i.

Given two points X_1, X_2, let

$$\lambda_1 = X_2 \bullet X_2 - X_2 \bullet X_1 ,$$
$$\lambda_2 = X_1 \bullet X_1 - X_1 \bullet X_2$$

Conveniently, set the notation

$$\lambda_1 = \text{lambda\_1}(X_1, X_2);$$
$$\lambda_2 = \text{lambda\_2}(X_1, X_2);$$

then express the nearest point by a convex combination of X_1, X_2

$$\upsilon = \frac{\lambda_1}{\lambda_1 + \lambda_2} X_1 + \frac{\lambda_2}{\lambda_1 + \lambda_2} X_2 \qquad (4)$$

In (4), 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 in two-dimensional space.

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

The refinement method used by Gilbert, Johnson, and Keerthi 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 the article (Gilbert et al.,1988). Eventually, the unique affinely independent subset will be found. However, the process is very complicated. Instead, our modification is a discard-and-add process and makes the refinement much simpler and, furthermore, geometrically clear. 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 which 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 repesents) v_i .

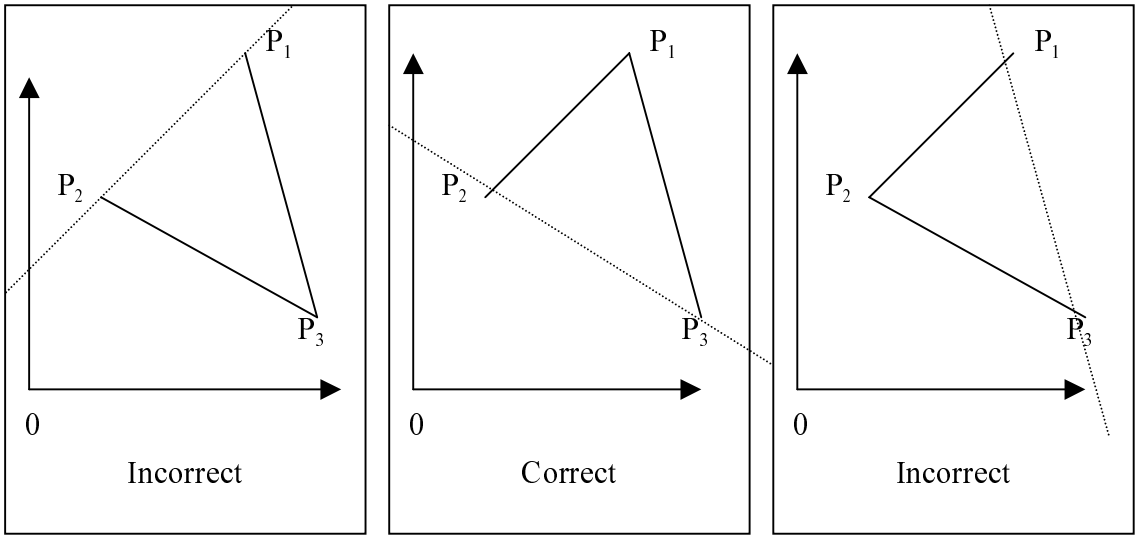

Fig. 5 Decide which point should be discarded.

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 (Gilbert et al.,1988). 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.

3.2 Three-dimensional case (Fig. 6)

(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. A tetrahedron has three triangles at each vertex, and three vertices per face. 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 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.

```
The Algorithm in 3D:
Vo ← initial_set(Z1, Z2, Z3) ;
i ← 0 ;
Repeat{
    ν i ← nu_compute(Vi) ;
   (Si, Hi) ← support_functions(- ν i) ;
   Vi+1 ← refine_set(Vi, Si) ;
   i ← i + 1 ;
} until ( ν i • ν i + Hi = 0)
```

Fig.6 GJK algorithm with our modification in 3D case.

Given the vertices of the tetrahedron P_1, P_2, P_3, P_4, let the remaining point be P_4. The plane formed by the three points P_1, P_2, P_3 is given by

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

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

$$K= -- (A\ P_1[0] + B\ P_1 \text{ (Gilbert et al.,1988)} + C\ P_1\text{(Cameron, Dec 1997))}.$$

Define

$$N = (A\ P_4[0] + BP_4 \text{ (Gilbert et al.,1988)} + C\ P_4\text{(Cameron, Dec 1997))} + K, \quad (3)$$

where $P_i = (P_i[0], P_i$ (Gilbert et al.,1988), P_i(Cameron, Dec 1997)).

Then the separation condition will be satisfied if $Z \bullet N < 0$. Discard the subsets containing the farthest point when more than one subset saisfying 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 et al.,1988)), we have:

$$\begin{aligned}
\lambda_1 &= \text{lambda_1}(P_2, P_3)(P_2 \bullet P_2 - P_2 \bullet P_1) + \text{lambda_2}(P_2, P_3)(P_3 \bullet P_2 - P_3 \bullet P_1);\\
\lambda_2 &= \text{lambda_1}(P_1, P_3)(P_1 \bullet P_1 - P_1 \bullet P_2) + \text{lambda_2}(P_1, P_3)(P_3 \bullet P_1 - P_3 \bullet P_2);\\
\lambda_3 &= \text{lambda_1}(P_1, P_2)(P_1 \bullet P_1 - P_1 \bullet P_3) + \text{lambda_2}(P_1, P_2)(P_2 \bullet P_1 - P_2 \bullet P_3);
\end{aligned} \quad (4)$$

λ will be negative if ν is on the edge of the tetrahedron. There are seven kinds of variations in the sign of λ. 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
(C)	One of λ is positive, the others are negative	3 cases	ν On Vertex
(D)	All λ are negative	None	Not Exist

Table 1

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

In situation (B), suppose the ν is on the edge composed by the points P_x and P_y. As two-

dimensional case,

$$\text{the only negative } \lambda \leftarrow 0;$$
$$\text{Compute the other two } \lambda_x,\ \lambda_y \text{ with}$$
$$\lambda_x = \text{lambda\_1}(P_x,P_y);$$
$$\lambda_y = \text{lambda\_2}(P_x,P_y); \qquad x,y \in 1,2,3$$

In situation (C):

$$\lambda_x \leftarrow 0 \text{ , if } \lambda_x < 0;$$
$$\text{the only positive } \lambda \leftarrow 1;$$

Finally, the nearest point ν is computed by:

$$\lambda = \lambda_1 + \lambda_2 + \lambda_3 ;$$

$$\upsilon = \frac{\lambda_1}{\lambda} P_1 + \frac{\lambda_2}{\lambda} P_2 + \frac{\lambda_3}{\lambda} P_3$$

IV. Comparison

In this section, simulations are performed to compare the improvements made by us and by Cameron (Cameron, Dec 1997). The input data are the vertices of two polyhedra.

4.1 Cameron's Enhancement (Cameron, Dec 1997)

From our experience in implementing the GJK algorithm shown in Fig. 1, the computation time is largely contributed by the evaluation of the support function. Therefore, the key to faster computation of distances between two convex bodies is to improve the computation speed of the support function. Stephen Cameron (Cameron, Dec 1997) 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 stopped 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 searching sequence from such a procedure form a path of edges originating from the initial vertex, going toward the vector with respect to which we evaluate the support function, and stopping at the supporting vertex. In addition, the vectors with respect to which we evaluate the support function, i.e., $-\nu_i$, roughly point from the polyhedron to the origin. Therefore, all supporting vertices lie on the sides of the polyhedron closer to the origin (as Fig. 8 shows). 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.

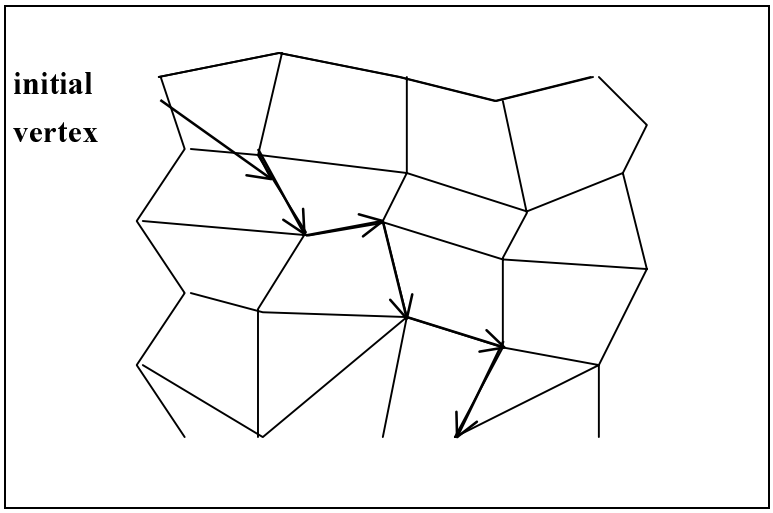

Fig. 8. The path of searching supporting vertex.

4.2 The Edge Connection Data

In implementing Cameron's improvement, 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. To obtain the faces data, we first test all the combinations of three vertices to see which ones determine the hyper-planes 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 extremely 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.

c).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

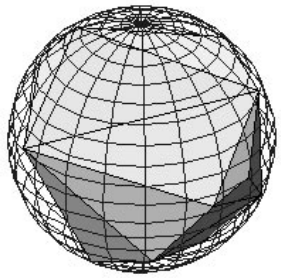

Fig.9. The construction of a polyhedron from a unit ball

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 is systematically given, each vertex of which is generated from a set of random points on a unit ball (see Fig. 7) 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.8. In Table 2, the most efficient results are marked by a *. It seems that when the total number of vertices increases, the combined algorithm (ours plus Cameron's modification) becomes more efficient. The results obtained give us ideas about how to compute the distance between the polyhedra efficiently. On one hand, Cameron's enhancement, which doesn't cause the computation time linear in the total number of hull points if the adjacency information of vertices is available, is significant especially when the total number of vertices is large.

On the other hand, our modification provides an alternative approach to distance calculation. By examining the performance of the algorithm with and without our modification for a specific problem,

as Table 2 shows, a more efficient combined algorithm can be adopted. Moreover, our modification has mathematical intuition and thus facilitates comprehension and implementation of the algorithm.

(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.

V. Conclusion

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 $\Delta(Y_s)$ of each subsets, 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. These can be used to compute the gradient of the distance with respect to the configuration variables, which is useful in robot path planning.

Finally, we have demonstrated experimentally that our modifications and the GJK algorithm both have an almost linear time complexity, and we have also presented an approach to calculate the vicinity matrix, which must be provided in updating the support function. After combining with Cameron's modifications, the comparison has also been shown. The reason why our changes do not dominate timings as much as Cameron did can be easily shown by the simulation, which means that computation of support function is the most time-consuming part in GJK algorithm (Cameron, Dec 1997), (Sato et al., Apr. 1996), (Ong and Gilbert, Apr. 1997).

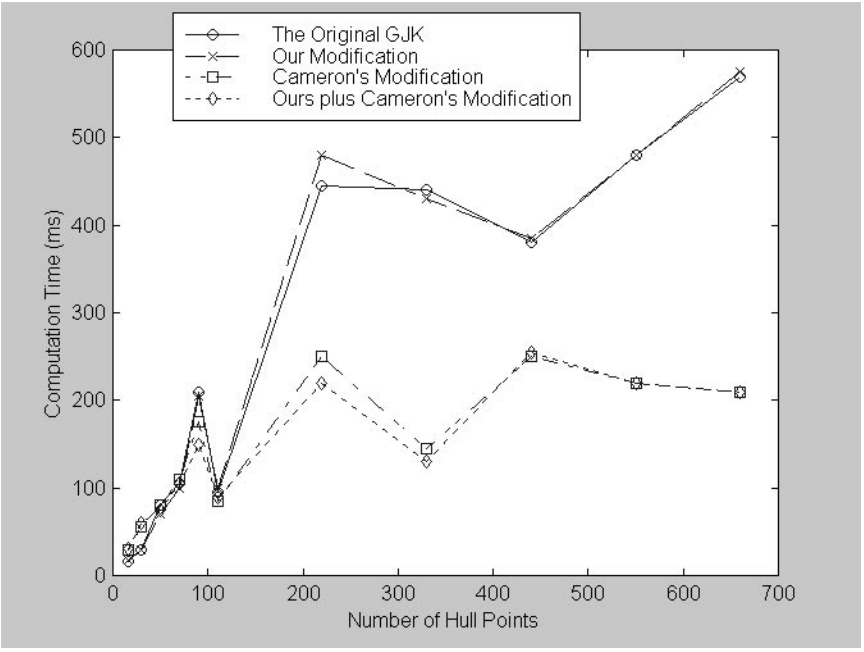

Fig. 10. The simulation result

Ⅵ. References

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.

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.

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.

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

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.

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

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

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

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*, pp. 771-778, Minneapolis, April 1996.

Elon Rimon, Stephon P. Boyd, "Obstacle Collision Detection Using Best Ellipsoid Fit," *Journal of Intelligent and Robotic Systems*, vol. 18, pp 105-126. 1997.

David E. Johnson, Elaine Cohen, "A Framework For Efficient Minimum Distance Computations," in *IEEE Int. Conf. Robot. Automat.*, pp. 3678-3684, May 1998.

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

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.

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, 1997.

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.

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.

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.

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.

APPENDIX

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

Given the vertices of a convex polyhedron, the information about which vertices determine a face or an edge is usually required, especially when the polyhedron is visualized in a computer graphics environment, or when the adjacency relationships of the vertices are needed in some geometric algorithms. 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 (Rockfellar, 1970) 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 some of the resulting 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. 9). 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 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.

```
/*
 * An Algorithm 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.9 Pseudo code of the face determination

```
for (each face){
   x = the vector pointing from the center of the face to a certain vertex,
       say, v1;
   for (each of the other vertices, vi)
      calculate the angle from x to the vector pointing from the center
      of the face to vi;
      sort the vertices, starting from v1, 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. 10 Pseudo code of edge determination