Proceedings of the 1991 IEEE
International Conference on Robotics and Automation
Sacramento, California - April 1991

A Fast Algorithm for Incremental Distance Calculation

Ming C. Lin and John F. Canny
University of California, Berkeley
Berkeley, CA 94720

Abstract

A simple and efficient algorithm for finding the closest points between two convex polyhedra is described here. Data from numerous experiments tested on a broad set of convex polyhedra on $\Re^3$ show that the running time is roughly *constant* for finding closest points when nearest points are approximately known and is linear in total number of vertices if no special initialization is done. This algorithm can be used for collision detection, computation of the distance between two polyhedra in three-dimensional space, and other robotics problems. It forms the heart of the motion planning algorithm of [1].

1 Introduction

In this paper we present a simple method for finding and tracking the closest points on a pair of convex polyhedra. The method is generally applicable, but is especially well suited to repetitive distance calculation as the objects move in a sequence of small, discrete steps. The method works by finding and maintaining the pair of closest features (vertex, edge, or face) on the two polyhedra. We take advantage of the fact that the closest features change only infrequently as the objects move along finely discretized paths. By preprocessing the polyhedra, we can verify that the closest features have not changed in constant time. Our experiments show that, once initialized, the expected running time of our algorithm is constant, independent of the complexity of the polyhedra.

Our method is very straightforward in its conception. We start with a candidate pair of features, one from each polyhedron, and check whether the closest points lie on these features. Since the objects are convex, this is a local test, involving only the boundary and coboundary of the candidate features. If the features fail the test, we step to a neighboring feature of one or both candidates, and try again. With some simple preprocessing, we can guarantee that every feature has a boundary and coboundary of constant size. This is how we can verify the closest feature pair in constant time.

When a pair of features fail the test, the new pair we choose is guaranteed to be closer than the old one. So when the objects move and one of the closest features changes, we usually find it after a single iteration. Even if the closest features are changing rapidly, say once per step along the path, our algorithm will take only slightly longer. Its also clear that the algorithm must terminate, in a number of steps at most equal to the number of feature pairs.

This algorithm is a key part of our general planning algorithm, described in [1]. That algorithm creates a one-dimensional roadmap of the free space of a robot by tracing out curves of maximal clearance from obstacles. We use the algorithm in this paper to compute distances and closest points. From there we can easily compute gradients of the distance function in configuration space, and thereby find the direction of the maximal clearance curves.

2 Related Work

Collision detection has a long history. It has been considered in both static and dynamic (moving objects) versions in [2], [3], [4], [5], [6], [7] and [8]. Our work shares with [6], [7], and [8] the calculation and maintenance of closest points during incremental motion. But whereas [6], [7], and [8] require linear time to verify the closest points, we use the properties of convex sets to reduce this check to constant time. In this aspect, it recalls work in [9] and [10] where local applicability constraints are used to check when two features can come into contact.

A fact that has often been overlooked is that collision *detection* for convex polyhedra can be done in

CH2969-4/91/0000/1008$01.00 © 1991 IEEE

linear time in the worst case. The proof is by reduction to linear programming. If two point sets have disjoint convex hulls, then there is a plane which separates the two sets. Letting the four variables that define the plane be variables, add a linear inequality for each vertex of polyhedron A that specifies that the point is on one side of the plane, and an inequality for each vertex of polyhedron B that specifies that it is on the other side. Megiddo and Dyers work [11], [12], [13] showed that linear programming is solvable in linear time for any fixed number of variables. More recent work [14] has shown that linear time linear programming algorithms are quite practical for a small number of variables. The algorithm of [14] has been implemented, and seems fast in practice.

3 Object Representations and Basic Definitions

Each object is represented as a convex polyhedron, or a union of convex polyhedra. Many real-world objects that have curved surfaces are represented by polyhedral approximations. The accuracy of the approximations can be improved by increasing the resolution or the number of vertices. With our method, there is little or no degradation in performance when the resolution is increased. For nonconvex objects, we rely on subdivision into convex pieces, which unfortunately, may take quadratic time.

Each polyhedron has a field for its faces, edges, vertices, position, and orientation. Each face is parameterized by its outward normal and its distance from the origin. Its data structure also includes a list of vertices which lie on its boundaries, a list of edges which bound the face, and its coboundary – the polyhedron itself. Each edge is described by its head, tail, right face, and left face. Each vertex is characterized by its x, y, z-coordinates, and its coboundary which is the set of edges intersecting at the vertex.

The closest pair of features between two general convex polyhedra is defined as the pair of features which contain the closest points. Let A and B denotes the sets of points defining objects A and B in $\Re^3$. The distance between objects A and B is the shortest Euclidean distance d_{AB}:

$$d_{AB} = \min_{p \in A, q \in B} | p - q |$$

and let $P_A \in S_A$, $P_B \in S_B$ be such that

$$d_{AB} = | P_A - P_B |$$

where P_A and P_B are a pair of closest points between objects A and B.

4 Preliminaries

Given a pair of features, there are altogether 6 possible cases that we need to consider: (1) a pair of vertices, (2) a vertex and an edge, (3) a vertex and a face, (4) a pair of edges, (5) an edge and a face, and (6) two faces.

In general, the case of two faces rarely happens. However, in our particular application to path planning we may end up moving along maximum clearance paths which keep two faces parallel, or an edge parallel to a face. It is important to be able to detect when we have such a degenerate case.

For each pair of features from objects A and B, first, we need to find a pair of closest points between these two features. Then, we need to verify that $point_A$ is truly the closest point of A to $feature_B$ and $point_B$ is truly the closest point of B to $feature_A$. If either check fails, a new (closer) feature is substituted, and the new pair is checked. Eventually, we must terminate with the closest pair, since we are moving closer to the closest pair of features through each iteration.

In the next section three intuitive geometric applicability tests, which are the essential components of our algorithm, will be described. The overall descriptions of our approach and the algorithm itself will be presented in more detail in the following sections.

5 Applicability Criteria

There are three basic applicability criteria that each feature-pair has to satisfy to be the closest features. These are (i) point-vertex, (ii) point-edge, and (iii) point-face applicability conditions. Here the implementation details will be briefly described.

5.1 Point-Vertex Applicability Criterion

If P is truly the closest point to V, then P must lie within the region bounded by the planes which are perpendicular to the coboundary of V, which are the edges touching V. This can be easily seen from the geometry of two vertices as shown in Fig.1. If P lies outside one of the plane boundary, then this implies that there is at least one edge of V's coboundary closer to P than V. Therefore, the procedure will "walk"

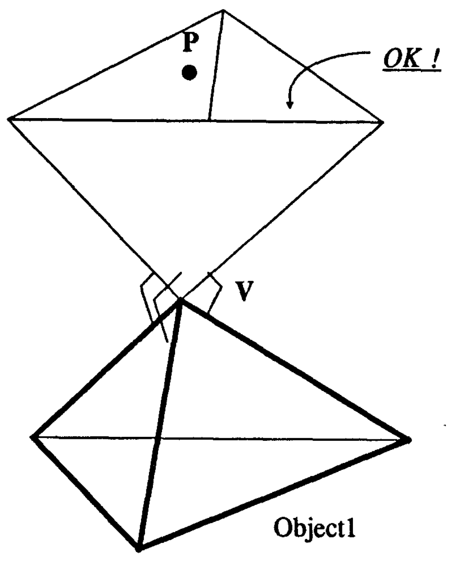

Figure 1: Point-Vertex Applicability Criterion

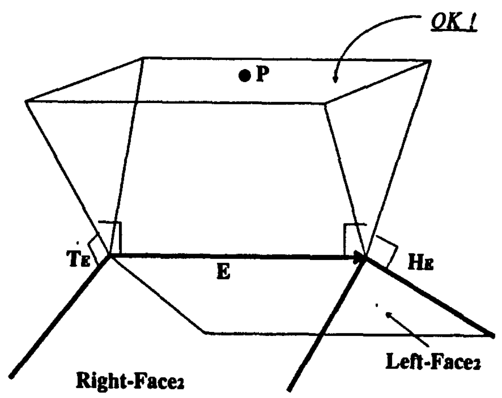

Figure 2: Point-Edge Applicability Criterion

to the edge that fails the applicability test and will iteratively call the feature test to verify whether P and the new edge are the closest features on the two objects.

5.2 Point-Edge Applicability Criterion

As for the point-vertex case, if P is really the closest point to E, then P must lie within the region bounded by the four planes which are superposed by the coboundaries of E, as shown in Fig.2. Two of these planes are perpendicular to E passing through the head and the tail of E, respectively. The other two planes are perpendicular to the coboundaries of E or the right and the left faces of E. If P satisfies all the applicability conditions, then the procedure will return P and E as a pair of the closest features. If P fails the applicability test of H_E or T_E, then the procedure will "walk" to the appropriate end of edge E and recursively call the general algorithm to verify whether the new vertex and P are the two closest features on two objects respectively. If P fails the applicability test of the right or the left face, then the procedure will "walk" to the corresponding face (coboundary of E) and call the general algorithm recursively to verify whether the new feature (the right or left face of E) and P are pair of the closest features.

5.3 Point-Face Applicability Criterion

Similarly, if P is actually the closest point to F, then P must lie within the region bounded by the planes which are perpendicular to F and containing the edges in the boundary of F, as shown in Fig.3. If P fails one applicability test from one of F's edges, the procedure will, once again, "walk" to the corresponding edge and call the general algorithm to check whether the new feature (in this case, boundary of F — E_F) and P are a pair of the closest features. In addition, we need to check whether P lies above F to guarantee that P is not inside the second polyhedron. If P lies beneath F, it implies that there is at least one feature on the given object closer to P than F or that collision is possible. Then, the procedure will return the closest feature of the given object to P and proceed with the usual checking procedures.

5.4 Preprocessing Procedure

For vertices of typical convex polyhedra, there are usually three or four edges in the coboundary. The faces of polyhedra also have four or five edges typically. Therefore, frequently the applicability criteria require only three to five quick tests for each round. When a face has more than five edges in its boundary or when a vertex has more than five edges in its coboundary, the polyhedron is preprocessed by subdividing the whole volume into smaller cells. That

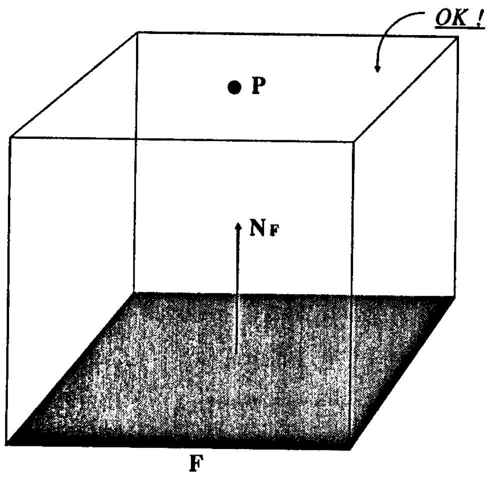

Figure 3: Vertex-Face Applicability Criterion

is, we divide the original polyhedron by inserting several virtual planes and edges. After preprocessing, each vertex of the new cell has only 4 or 5 coboundaries (edges) and each face has only 4 or 5 boundaries (edges). Fig.4 shows how this can be done on a cone with 8 boundaries (edges) on its bottom face and 8 coboundaries (edges) on its apex and on a cylinder with 8 edges on its top and bottom faces. This preprocessing procedure is a simple calculation, and it guarantees that when the algorithm starts, every feature has a constant size boundary and coboundary. Consequently, the three applicability tests described above run in *constant time*.

In the next section, we will show how these applicability conditions are used to update the pair of closest features between two convex polyhedra approximately in *constant time*.

6 General Description of the Approach

Given a pair of features of two polyhedra, we apply the appropriate applicability check from the last section.

Except for case (1) - a pair of vertices, case (5) - an edge and a face, and case (6) - two faces, we need to compute the nearest points between two features, before we can apply the applicability tests described in the previous section. The details for computing

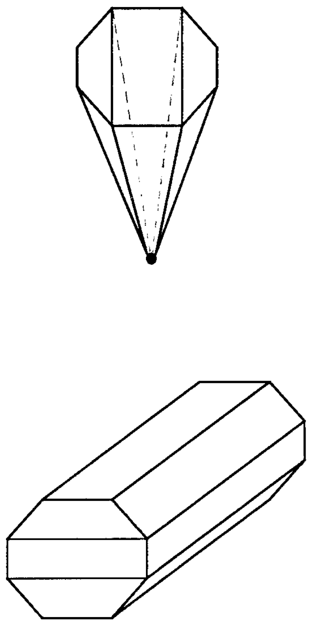

Figure 4: Preprocessing of a Cone and a Cylindar

these nearest points are rather trivial, thus omitted here. (Please refer to [15], if necessary.)

(1) If the features are a pair of vertices, then they both have to satisfy the applicability conditions imposed by each other, in order for them to be the closest features. If either one of the vertices fails the applicability test imposed by the other, the algorithm will return a new pair of features - one of the two vertices and the edge for which the test failed, then continue checking the new features until it finds the closest pair.

(2) Given a vertex and an edge, the algorithm will check whether the vertex satisfies the applicability conditions imposed by the edge *and* whether the nearest point on the edge to the vertex satisfies the applicability conditions imposed by the vertex. If both verifications return value "true", then they are the closest features. Otherwise, a corresponding new pair of features (depending on which test failed) will be returned and the algorithm will proceed until it finds

the pair of closest features.

(3) For the case of a vertex and a face, both of the applicability tests imposed by the face to the vertex *and* from vertex to the nearest point on the face must be satisfied for this pair to qualify as the "closest-feature pair". Otherwise, a new pair of features will be returned and the algorithm will be called again until the closest-feature pair is found.

(4) Similarly, given a pair of edges as inputs, if their nearest points satisfy the applicability conditions imposed by the each other, then they are the closest features between two polyhedra. If not, one of the edges will be changed to a neighboring vertex or a face and the check will be done again on the new pair of features.

(5) When a given pair of features is an edge and a face, we first need to decide whether the edge is parallel to the face. If it isnt, then the actual closest features will be either one of the vertices of the edge and the face, or the edge and some other edge bounding the face. The former case occurs when this vertex satisfies the vertex-face applicability condition, and when the edge is pointing "into" the face in the direction of this vertex. Otherwise the latter case applies. The edge (bounding the face) to be chosen is the edge which is closest to the original edge. If the edge and the face are parallel, then they are the closest features provided two conditions are met. (i) The edge must cut the "applicability prism" figure 3 of the face, and (ii) the face normal must lie "between" the face normals of the faces bounding the edge.

(6) In the rare occasion when two faces are given as inputs, the algorithm has to decide if they are parallel. If they are, it will evoke an overlap-checking subroutine which runs roughly at linear time in the total number of edges of the two faces. If they are both parallel and overlapping, then they are in fact the closest features. However, if they are not parallel *or* parallel yet not overlapping, then the first face and the nearest edge of the second face to the first face will be returned as a pair of new features, and the algorithm will process them as the case of an edge and a face.

A careful study of all of the above checks shows that they all take time in proportion to the size of the boundary and coboundary of each feature. Therefore, after preprocessing, all checks run in constant time. The only exception to this is when $feature_A$ is a face, and $feature_B$ lies under the plane of $feature_A$. In this case, we cant use a local feature change, because this may lead to the procedure getting stuck in a loop. The distance between the closest pair of points corresponds to distance between the closest point to the origin of the Minkowski sum and the origin itself. Geometrically, we are moving around on the "far side" of the Minkowski sum of the polyhedra, and the distance function has many local minimum, in which we may become trapped. When this situation occurs, we instead search among all the features of object A to find a closest feature to the $feature_B$. This is not a constant time step, but note that it is impossible for the algorithm to move to such an opposing face once it is initialized. So this situation can only occur when the algorithm is first called on an arbitrary pair of features.

The algorithm can take any random pair of features of two polyhedra and find the true pair of closest features by iteratively checking and changing features. In this case, the running time is proportional to the number of feature pairs traversed in this process. It is not more than the product of the numbers of features of the two polyhedra, because the distance between feature pairs must always decrease when a switch is made, which makes cycling impossible. Empirically, it seems to be not worse than linear when started from an arbitrary pair of features. However, once it finds the closest pair of features *or* a pair in their vicinity, it only takes constant time to keep track of the closest pair as the two objects translate and rotate in three space. The overall computational time is shorter in comparison with other algorithms available at the present time.

If the two objects are just touching or intersecting, it gives an error message to indicate collision and terminates the procedure with the contacting-feature pair as returned values. The proof of algorithm's completeness can be found in [15].

7 Numerical Experiments

The algorithm described in this paper has been implemented in Lucid Common Lisp. The input data are a random pair of features from two given objects in three dimensional space. The subroutine outputs are a pair of the closest features of the two polyhedra, as well as a pair of nearest points and the Euclidean distance between them.

Numerous examples in three dimensional space have been applied to test the subroutine. The examples include a wide variety of polytopes: cubes,

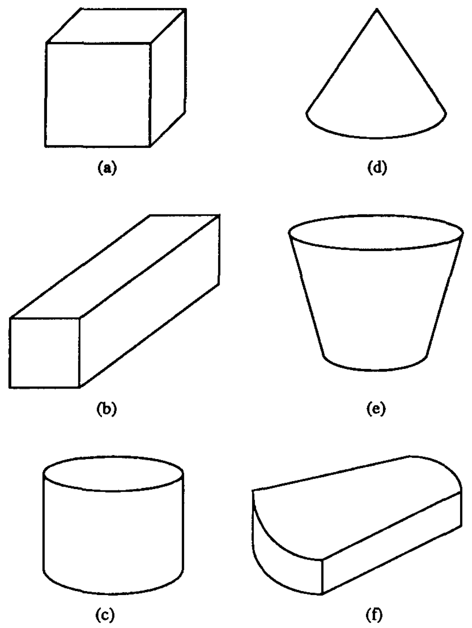

Figure 5: Polytopes Used in Example Computations

rectangular boxes, cylinders, cones, frustrums, and a Puma link of different sizes as shown in Fig.5. In particular, the number of facets (thus the number of vertices) or the resolution for cylinders, cones and frustrums have been varied from 12, 20, 24, up to 48 in order to generate a richer set of polytopes for testing purpose.

For each pair of polytopes (placed randomly by translations and rotations), at least 18 pairs of features are selected to test the subroutine. The examples were run on a Sun4 SPARC station which is a 12.5 Mips 1.4 Mega flops machine.

The experiment results are briefly summarized in Table 1. A more detailed table of running time with comparison to the other algorithm available now [6] is present in [15]. With initialization to the previous closest feature, the subroutine can almost always keep track of the closest features of two given polytopes at constant time (about 3 to 4 msec). Without initialization, the algorithm runs in average time not worse than linear in the total number of vertices. This is what we would expect, since it seems unlikely that the algorithm would need to visit a given feature more than once. In practice, we believe our algorithm compares very favorably with other algorithms designed for distance computations or collision detection. (Please see [3], [4], [5], [6], [7], [16], [17], [18], and [19].)

Objects	M1 + M2 = N	w/o Init	w/ Init
(a),(b)	8 + 8 = 16	21	3.2
(a),(f)	8 + 16 = 24	23	3.3
(a),(c)	8 + 24 = 32	26	3.5
(d),(e)	48 + 21 = 69	41	3.5
(c),(c)	96 + 48 = 144	67	3.0

Table 1: Average CPU Time in Milliseconds

8 Conclusion

A new algorithm for computing the Euclidean distance between two polyhedra has been presented here. It utilizes the geometry of polyhedra to establish three important applicability criteria for detecting collisions. With preprocessing to reduce the size of coboundary when appropriate, it runs almost always in *constant time* if the previous closest features have been provided and (on average) linear in the total number of vertices if no special initialization is done. Beside its efficiency and simplicity, it is also complete — it is guaranteed to find the closest feature or point pair if the objects are separated; it gives an error message to indicate collision and returns the contacting pair if they are just touching or intersecting.

The methodology described here can be used in distance calculations, collision detection, motion planning, and other robotics problem. Our application is to plan obstacle-avoidance paths. By tracking the closest feature pair of two convex polyhedra incrementally, the algorithm traces out the skeleton curves which are loci of the maxima for a distance function.

With slight modification, this algorithm can be easily extended for nonconvex objects. Since it runs in constant time once initialized, the algorithm is extremely useful in reducing the error by increasing the resolution of polytope approximation, when the objects have smooth curved surfaces. Refining the approximation to reduce error will no longer have detrimental "side effect" in running time.

Acknowledgements

This work is supported by the David and Lucile Packard Foundation, by NSF Presidential Young Investigator Grant number IRI-8958577, and by Darpa Durip Contract #N00039-88-C0292 Task4.

References

[1] J. F. Canny and M. C. Lin. An opportunistic global path planner. *Proc. IEEE ICRA*, pages pp. 1554–1559, 1990.

[2] J. W. Boyse. Interference detection among solids and surfaces. *Comm ACM*, 22(1):3–9, 1979.

[3] M. Orlowski. The computation of the distance between polyhedra in 3-space. Presented SIAM Conf. on Geometric Modeling and Robotics, 1985. Albany, NY.

[4] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. *Proc. IEEE ICRA*, pages pp. 591–596, 1986.

[5] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. PAMI*, 8:pp. 200–209, 1986.

[6] E. G. Gilbert and D. W. Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *IEEE J. Robotics Automat.*, RA-1:pp. 21–30, 1985.

[7] E. G. Gilbert and S. M. Hong. A new algorithm for detecting the collision of moving objects. *Proc. IEEE ICRA*, pages pp. 8–14, 1989.

[8] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:pp. 193–203, 1988.

[9] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):pp. 560–570, 1979.

[10] B. R. Donald. *Motion Planning with Six Degrees of Freedom.* PhD thesis, MIT Artificial Intelligence Lab., 1984.

[11] N. Megiddo. Linear-time algorithms for linear programming in r^3 and related problems. *SIAM J. Computing*, 12:pp. 759–776, 1983.

[12] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Jour. ACM*, 31:pp. 114–127, 1984.

[13] M. E. Dyer. Linear algorithms for two and three-variable linear programs. *SIAM J. on Computing*, 13:pp. 31–45, 1984.

[14] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[15] J. F. Canny and M. C. Lin. Local methods for fast computation of distance functions. In Preparation, 1990. U. C. Berkeley.

[16] E. G. Gilbert and C. P. Foo. Computing the distance between general convex objects in three dimensional space. *IEEE Trans. Robotics Automat.*, 6(1), 1990.

[17] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6:pp. 381–392, 1985.

[18] W. E. Red. Minimum distances for robot task simulation. *Robotics*, 1:pp. 231–238, 1983.

[19] P. Wolfe. Finding the nearest points in a polytope. *Math. Programming*, 11:pp. 128–149, 1976.