1 An Incremental Distance Computation Algorithm

In this chapter we present a simple and efficient method to compute the distance between two convex polyhedra by finding and tracking the closest points. 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 due to its incremental nature. The method works by finding and maintaining a pair of closest features (vertex, edge, or face) on the two polyhedra as the they move. 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 or performed an update to a neighboring feature in expected constant time. Our experiments show that, once initialized, the expected running time of our incremental algorithm is *constant* independent of the complexity of the polyhedra, provided the motion is not abruptly large.

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 neighboring features (boundary and coboundary as defined in Sec. **??**) 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 constant number of neighboring features. This is how we can verify a closest feature pair in expected constant time.

When a pair of features fails the test, the new pair we choose is guaranteed to be closer than the old one. Usually when the objects move and one of the closest features changes, we can 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. It is also clear that in any situation 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 Chap.**??** 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 chapter 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.

In addition, this technique is well adapted for dynamic collision detection. This follows naturally from the fact that two objects collide if and only if the distance between them is less than or equal to zero (plus some user defined tolerance). In fact, our approach provide more geometric information than what is necessary, i.e. the distance information and the closest feature pair may be used to compute inter-object forces.

2 Closest Feature Pair

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 in the convex case. For nonconvex objects, we rely on subdivision into convex pieces, which unfortunately, may take $O((n + r^2)logr)$ time to partition a nondegenerate simple polytope of genus 0, where n is the number of vertices and r is the number of *reflex edges* of the original nonconvex object [23, 2]. In general, a polytope of n vertices can always be partitioned into $O(n^2)$ convex pieces [22].

Given the object representation and data structure for convex polyhedra described in Chapter **??**, here we will define the term *closest feature pair* which we quote frequently in our description of the distance computation algorithm for convex polyhedra.

The closest pair of features between two general convex polyhedra is defined as the pair of features which contain the closest points. Let polytopes A and B denote the convex sets in R^3. Assume A and B are closed and bounded, therefore, compact. The distance between objects A and B is the shortest Euclidean distance d_{AB}:

$$d_{AB} = \inf_{p \in A, q \in B} \mid p - q \mid$$

and let $P_A \in A$, $P_B \in B$ be such that

$$d_{AB} = \mid P_A - P_B \mid$$

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

For each pair of features (f_A and f_B) from objects A and B, first we find the pair of nearest points (say P_A and P_B) between these *two features*. Then, we check whether these points are a pair of closest points *between A and B*. That is, we need to verify that f_B is truly a closest feature on B to P_A and f_A is truly a closest feature on A to P_B This is verification of whether P_A lies inside the Voronoi region of f_B and whether P_B lies inside the Voronoi region of f_A (please see Fig. 1). If either check fails, a new (closer) feature is substituted, and the new pair is checked again. Eventually, we must terminate with a closest pair, since the distance between each candidate feature pair decreases, as the algorithm steps through them.

The test of whether one point lies inside of a Voronoi region of a feature is what we call an "applicability test". In the next section three intuitive geometric applicability tests, which are the essential components of our algorithm, will be described. The overall description of our approach and the completeness proof will be presented in more detail in the following sections.

3 Applicability Criteria

There are three basic applicability criteria which we use throughout our distance computation algorithm. These are (i) point-vertex, (ii) point-edge, and (iii) point-face applicability conditions. Each of these applicability criteria is equivalent to a membership test which verifies whether *the point* lies in the Voronoi region of the *feature*. If the *nearest points* on two features both lie inside the Voronoi region of the other feature, then the two features are a closest feature pair and the points are closest points between the polyhedra.

3.1 Point-Vertex Applicability Criterion

If a vertex V of polyhedron B is truly a closest feature to point P on polyhedron A, then P must lie within the Voronoi region bounded by the constraint planes which are perpendicular to the coboundary of V (the edges touching V). This can be seen from Fig.2. If P lies outside the region defined by the

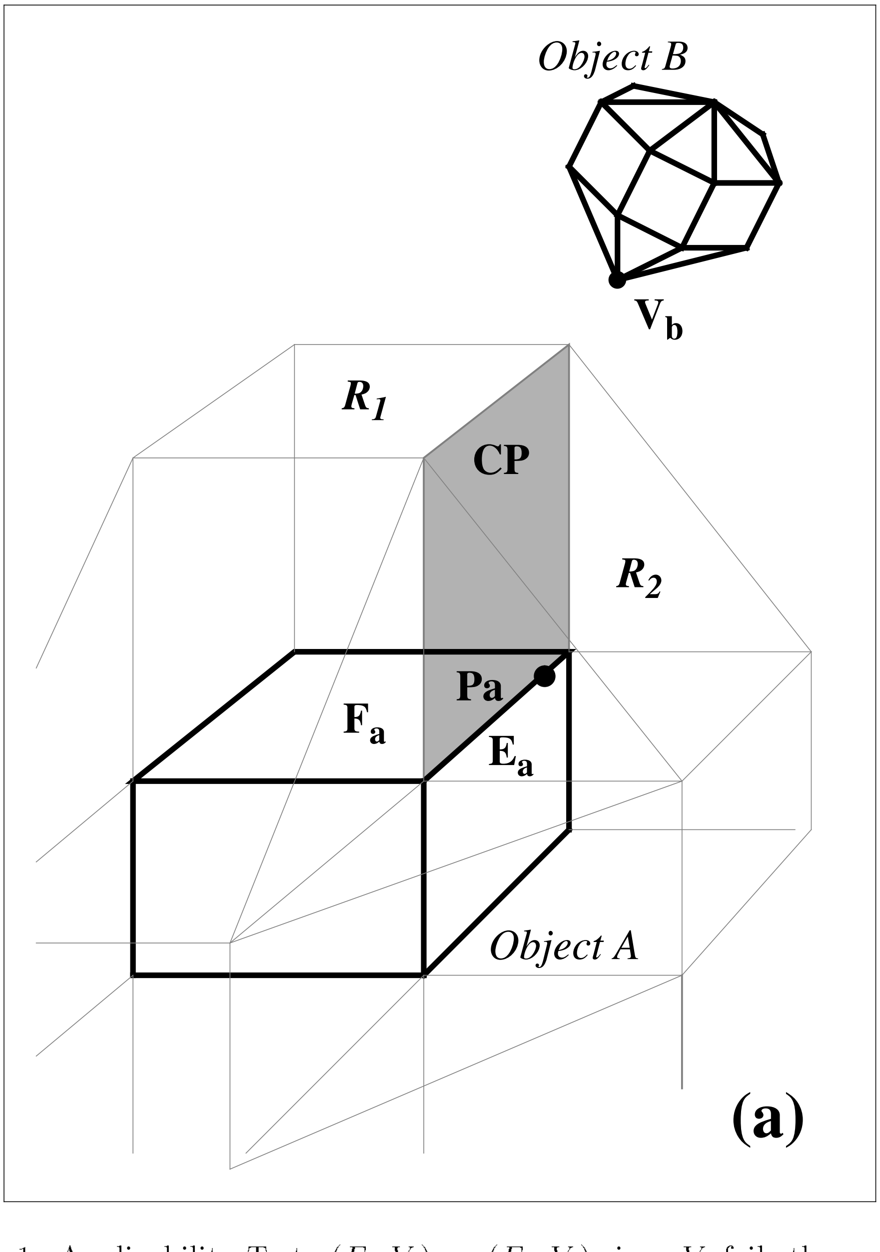


Figure 1: Applicability Test: $(F_a, V_b) \rightarrow (E_a, V_b)$ since V_b fails the applicability test imposed by the constraint plane CP. R_1 and R_2 are the Voronoi regions of F_a and E_a respectively.

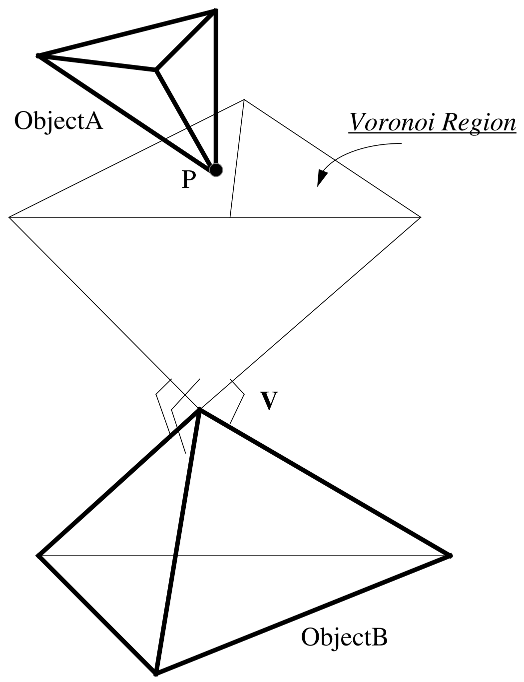


Figure 2: Point-Vertex Applicability Criterion

constraint planes and hence on the other side of one of the constraint planes, then this implies that there is at least one edge of V's coboundary closer to P than V. This edge is normal to the violated constraint. Therefore, the procedure will walk to the corresponding edge and iteratively call the closest feature test to verify whether the *feature containing* P and the *new edge* are the closest features on the two polyhedra.

3.2 Point-Edge Applicability Criterion

As for the point-edge case, if edge E of polyhedron B is really a closest feature to the point P of polyhedron A, then P must lie within the Voronoi region bounded by the four constraint planes of E as shown in Fig.3. Let the head and tail of E be H_E and T_E respectively. Two of these planes are perpendicular to E passing through the head H_E and the tail T_E of E. The other two planes contain E and one of the normals to the coboundaries of E (i.e. the right and the left faces of E). If P lies inside this wedge-shaped

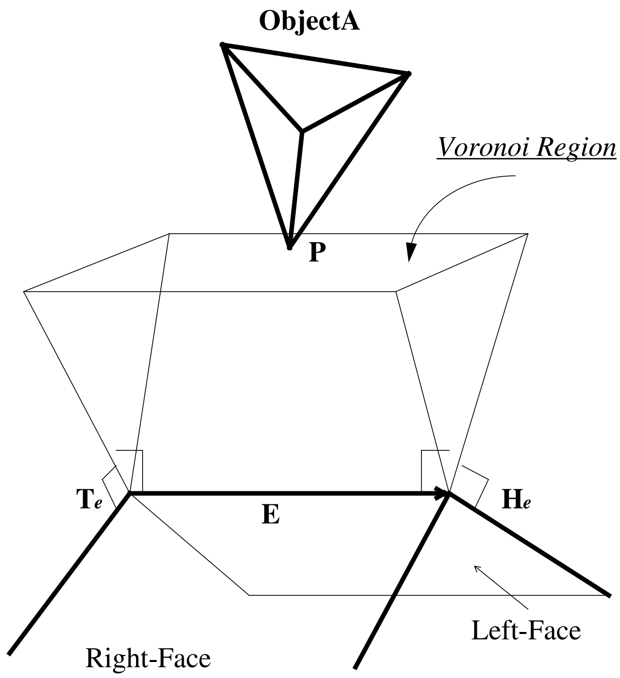


Figure 3: Point-Edge Applicability Criterion

Voronoi region, the applicability test succeeds. If P fails the test imposed by H_E (or T_E), then the procedure will walk to H_E (or T_E) which must be closer to P and recursively call the general algorithm to verify whether the *new vertex* and the *feature containing* P are the two closest features on two polyhedra respectively. If P fails the applicability test imposed by the right (or left) face, then the procedure will walk to the right (or left) face in the coboundary of E and call the general algorithm recursively to verify whether the *new face* (the right or left face of E) and the *feature containing* P are a pair of closest features.

3.3 Point-Face Applicability Criterion

Similarly, if a face F of polyhedron B is actually a closest feature to P on polyhedron A, then P must lie within F's Voronoi region defined by F's prism and above the plane containing F, as shown in Fig.4. F's **prism** is the region bounded by the constraint planes which are perpendicular to F

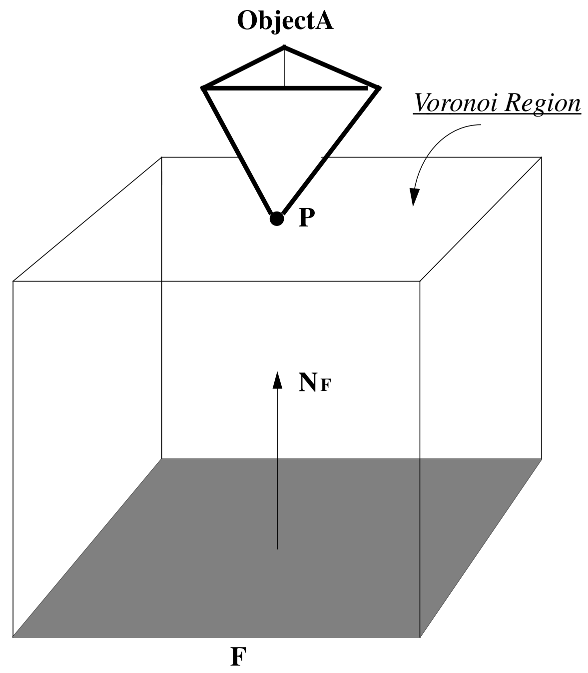


Figure 4: Vertex-Face Applicability Criterion

and contain the edges in the boundary of F.

First of all, the algorithm checks if P passes the applicability constraints imposed by F's edges in its coboundary. If so, the feature containing P and F are a pair of closest features; else, the procedure will once again walk to the edge corresponding to a failed constraint check and call the general algorithm to check whether the *new edge* in F's boundary (i.e. E_F) and the *feature containing* P are a pair of the closest features.

Next, we need to check whether P lies above F to guarantee that P is not inside the polyhedron B. If P lies beneath F, it implies that there is at least one feature on polyhedron B closer to the feature containing P than F or that collision is possible. In such a case, the nearest points on F and the feature containing P may define a local minimum of distance, and stepping to a neighboring feature of F may increase the distance. Therefore, the procedure will call upon a $O(n)$ routine (where n is the number of features of B) to search for the closest feature on the polyhedron B to the feature containing P and proceed with the general algorithm.

3.4 Subdivision Procedure

For vertices of typical convex polyhedra, there are usually three to six edges in the coboundary. Faces of typical polyhedra also have three to six edges in the boundaries. Therefore, frequently the applicability criteria require only a few 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 Voronoi region associated with each feature is preprocessed by subdividing the whole region into smaller cells. That is, we subdivide the prismatic Voronoi region of a face (with more than five edges) into quadrilateral cells and divide the Voronoi region of a vertex (with more than five coboundaries) into pyrmidal cells. After subdivision, each Voronoi cell is defined by 4 or 5 constraint planes in its boundary or coboundary. Fig.5 shows how this can be done on a vertex's Voronoi region with 8 constraint planes and a face's Voronoi region with 8 constraint planes. This subdivision procedure is a simple algorithm which can be done in linear time as part of preprocessing, and it guarantees that when the algorithm starts, each Voronoi region has a constant number of constraint planes. Consequently, the three applicability tests described above run in *constant time*.

3.5 Implementation Issues

In order to minimize online computation time, we do all the subdivision procedures and compute all the Voronoi regions first in a one-time computational effort. Therefore, as the algorithm steps through each iteration, all the applicability tests would look uniformly alike — all as "point-cell applicability test", with a slight difference to the point-face applicability criterion. There will no longer be any distinction among the three basic applicability criteria.

Each feature is an open subset of an affine plane. Therefore, end points are not consider part of an edge. This is done to simplify the proof of convergence. Though we use the winged edge representation for our implementation, all the edges are not deliberately oriented as in the winged edge representation.

We also use hysteresis to avoid cycling in degenerate cases when there is more than one feature pair separated at approximately the same distance. We achieve this by making the Voronoi regions overlap each other with small and "flared" displacements. To illustrate the "flared Voronoi cells", we demonstrate them by an example in Fig. 6. Note that all the constraint planes

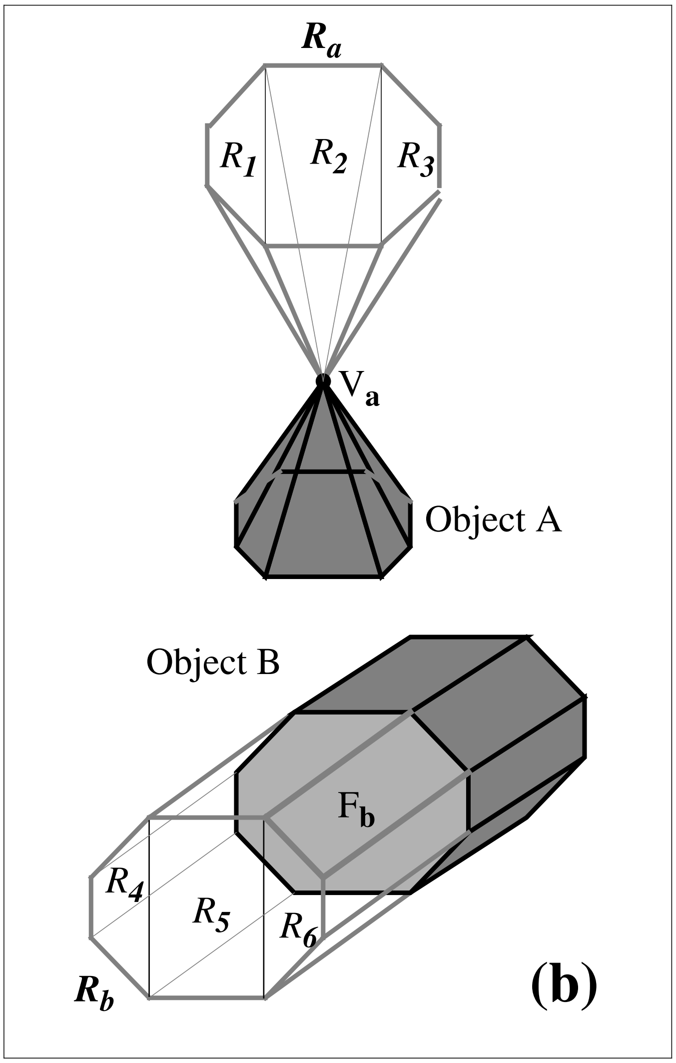


Figure 5: Preprocessing of a vertex's conical applicability region and a face's cylindrical applicability region

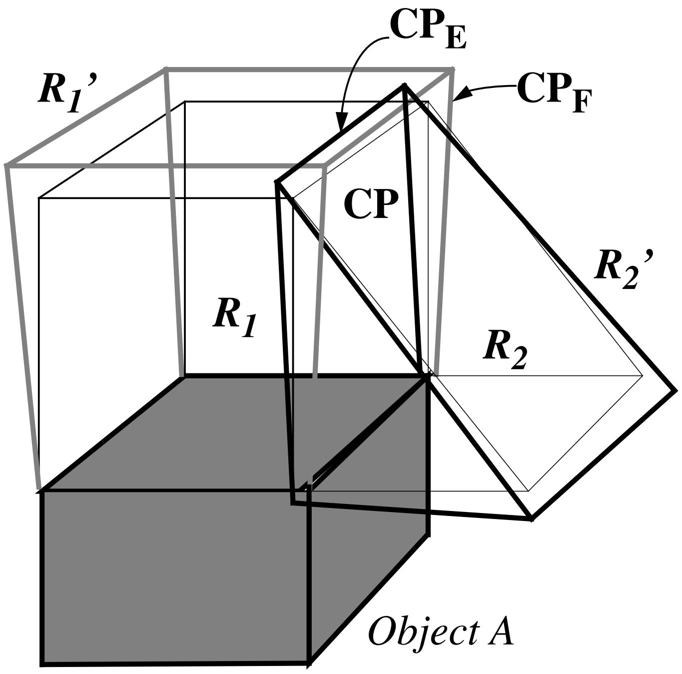


Figure 6: An example of Flared Voronoi Cells: CP_F corresponds to the flared constraint place CP of a face and CP_E corresponds to the flared constraint plane CP of an edge. R_1' and R_2' are the *flared* Voronoi regions of unperturbed R_1 and R_2. Note that they overlap each other.

are tilted outward by a small angular displacement as well as infinitesimal translational displacement. The angular displacement is used to ensure the overlap of adjacent regions. So, when a point falls on the borderline case, it will not flip back and forth between the two adjacent Voronoi regions. The small translational displacement in addition to angular displacement is needed for numerical problem arising from point coordinate transformations. For an example, if we transform a point p by a homogeneous matrix T and transform it back by T^{-1}, we will not get the same point p but p' with small differences in each of its coordinate, i.e. $T^{-1}(Tp)/neqp$.

All faces are convex, *but* because of subdivision they can be coplanar. Where the faces are coplanar, the Voronoi regions of the edges between them is null and face constraints point directly to the next Voronoi region associated with their neighboring coplanar faces.

In addition, the order of the applicability test in each case does not affect the final result of the algorithm. However, to achieve faster convergence, we look for the constraint plane which is violated the most and its associated neighboring feature to continue the verification process in our implementation.

We can also calculate how frequently we need to check for collision by taking into consideration the maximum magnitude of acceleration and velocity and initial separation. We use a priority queue based on the lower bound on time to collision to detect possible intersection at a timely fashion. Here the step size is adaptively set according to the lower bound on time to collision we calculate. For more information on this approach, please see Chapter **??** of this thesis.

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

4 The Algorithm

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 or edge-face rarely happens. However, in applications such as path planning we may end up moving along maximal 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.

4.1 Description of the Overall Approach

Given a pair of features f_A and f_B each from convex polyhedra A and B respectively, except for cases (1), (5) and (6), we begin by computing the nearest points between f_A and f_B. The details for computing these nearest points between f_A and f_B are rather trivial, thus omitted here (please refer to Appendix **??** if necessary). However, we would like to reinforce one point. We do not assume infinitely long edges when we compute these nearest points. The nearest points computed given two features are the actual points on the features, not some virtual points on infinitely extended edges, i.e. nearest points between edges may be their endpoints.

(1) If the features are a pair of vertices, V_A and V_B, then both V_A and V_B have to satisfy the point-vertex applicability conditions imposed by each other, in order for them to be the closest features. If V_A fails the point-vertex applicability test imposed by V_B or vice versa, then the algorithm will return a new pair of features, say V_A and E_B (the edge whose corresponding constraint was violated in the point-vertex applicability test), then continue verifying the new feature pair until it finds a closest pair.

(2) Given a vertex V_A and an edge E_B from A and B respectively, the algorithm will check whether the vertex V_A satisfies the point-edge applicability conditions imposed by the edge E_B *and* whether the nearest point P_E on the edge E_B to V_A satisfies the point-vertex applicability conditions imposed by the vertex V_A. If both checks return "true", then V_A and E_B are the closest features. Otherwise, a corresponding new pair of features (depending on which test failed) will be returned and the algorithm will continue to walk closer and closer toward a pair of closest features.

(3) For the case of a vertex V_A and a face F_B, both of the point-face applicability tests imposed by the face F_B to the vertex V_A *and* the point-vertex applicability criterion by V_A to the nearest point P_F on the face must be satisfied for this pair of features to qualify as a 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 E_A and E_B as inputs, if their nearest points P_A and P_B satisfy the point-edge applicability criterion imposed by E_B and E_A, then they are a pair of closest features between two polyhedra. If not, one of the edges will be changed to a neighboring vertex or a face and the verification process will be done again on the new pair of features.

(5) When a given pair of features is an edge E_A and a face F_B, we first need to decide whether the edge is parallel to the face. If it is not, then either one of two edge's endpoints and the face, *or* the edge and some other edge bounding the face will be the next candidate pair. The former case occurs when the head or tail of the edge satisfies the point-face applicability condition imposed by the face, and when one endpoint of E is closer to the plane containing F than the other endpoint of E. Otherwise, the edge E

and the closest edge E_F on the face's boundary to E will be returned as the next candidate features.

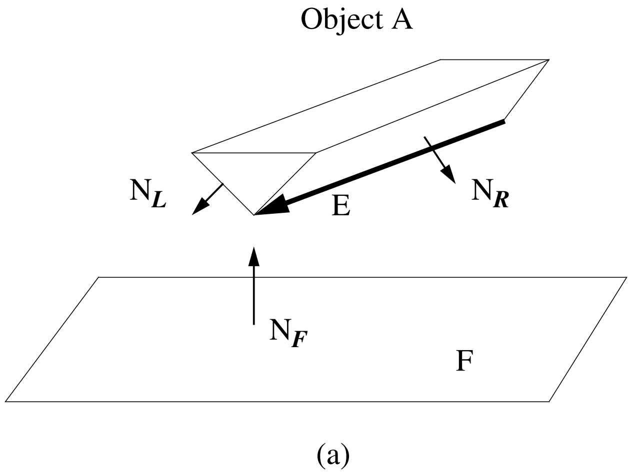
If the edge and the face are parallel, then they are the closest features provided three conditions are met. (i) The edge must cut the "applicability prism" $Prism_F$ which is the region bounded by the constraint planes perpendicular to F_B and passing through F_B's boundary (or the region swept out by F_B along its normal direction), that is $E \cap Prism_F \neq \emptyset$ (ii) the face normal must lie between the face normals of the faces bounding the edge (see Fig. 7, Sec. 5, and the pseudo code in Appendix **??**), (iii) the edge must lie above the face. There are several other situations which may occur, please see Appendix **??** or the proof of completeness in Sec. 5 for a detailed treatment of edge-face case.

(6) In the rare occasion when two faces F_A and F_B are given as inputs, the algorithm first has to decide if they are parallel. If they are, it will invoke an overlap-checking subroutine which runs in linear time in the total number of edges of F_A and F_B. (Note: it actually runs in constant time after we preprocess the polyhedra, since now all the faces have constant size of boundaries.) If they are both parallel, F_A's projection down onto the face F_B overlaps F_B and each face is above the other face relative to their outward normals, then they are in fact the closest features.

But, if they are parallel yet not overlapping, then we use a linear time procedure [71] to find the closest pair of edges, say E_A and E_B between F_A and F_B, then invoke the algorithm again with this new pair of candidates (E_A, E_B).

If F_A and F_B are not parallel, then first we find the closest feature (either a vertex or an edge) f_A of F_A to the plane containing F_B and vice versa. Then, we check if this closest feature f_A satisfy the applicability constraint imposed by F_B. If so, the new candidate pair will be this closest feature f_A and F_B; otherwise, we enumerate over all edge-pairs between two faces to find a closest pair of edges and proceed from there.

If one face lies on the "far side" of the other (where the face is beneath the other face relative to their face outward normals) or vice versa, then the algorithm invokes a linear-time routine to step away from this local minimum of distance, and provides a new pair of features much closer than the previous pair. Please refer to Appendix **??** for a complete description of face-face case.



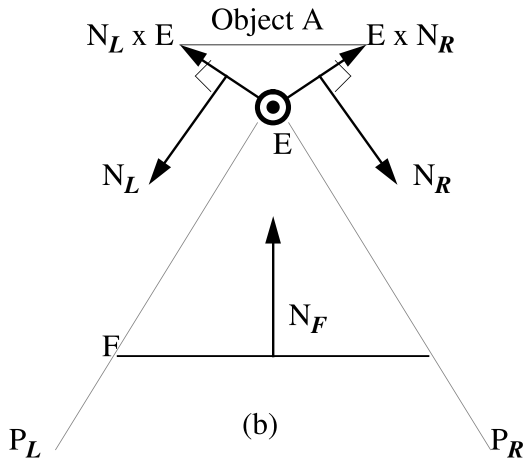


Figure 7: (a) An overhead view of an edge lying above a face (b) A side view of the face outward normal bounded by the two outward normals of the edge's left and right faces.

4.2 Geometric Subroutines

In this section, we will present some algorithms for geometric operations on polytopes. They are used in our implementation of the algorithms described in this thesis.

- **Edge & Face's Prism Intersection:**

To test for the intersection of a *parallel* edge E_A to F_B within a prism $Prism_F$ swept out by a face F_B along its face normal direction N_F, we check whether portion of this edge lies within the interior region of $Prism_F$. This region can be visualized as the space bounded by n planes which are perpendicular to the face F_B and passing through the n edges in its boundary.

Suppose F_i is a face of the prismatic region $Prism_F$ and perpendicular to the face F_B and passing through the edge E_i bounding F_B. To test for E_A's intersection within the prism $Prism_F$, we need to check whether E_A intersects some face F_i of R_F.

We can perform this test with easy and efficient implementation[1] of this intersection test. We first parameterize the edge E_A between 0 and l where l is the length of E_A. The idea is to intersect the intervals of the edge E_A contained in the *exterior*[2] of each half-space with each constraint plane in the prism of a face F_B. First, we decide whether the edge E_A points inward or outward of each hyperplane F_i of the region by taking the inner product of E_A's unit directional vector with F_i's outward normal N_{F_i}.

Then, we can decide which portion of E_A intersect the *exterior* half-space of F_i by taking another inner product of the head of E_A and F_i's outward normal normalized by the inner product of E_A and N_{F_i} to calculate the relative "inclusion" factor of E_A inside of the prism region. The relative (beginning and ending) inclusion factors should be in the range of 0 and l for an indication of an edge-face's prism intersection.

Pseudo code for this approach just described is listed in Appendix **??**. This geometric test is used in our implementation of distance computation al-

[1] suggested by Brian Mirtich

[2] We define the exterior half-space of each constraint plane to be the half-space where all the points satisfy the applicability constraints imposed by F_B's boundary. The "point-cell-checkp" routine in the pseudo code can clarify all the ambiguity. Please refer to Appendix **??** if necessary.

gorithm as a subroutine to check whether a given edge intersects the Voronoi prism of a given face to test their eligibilities be a pair of closest features.

• **Face & Face's Prisim Intersection:**

To check if two convex polygons A and B are overlapping, one of the basic approaches is to verify if there is some vertex of A lying on the "inside" of *all* edges in the boundary of B or vice versa. The "inside" refers to the *interior half-plane* of the edge, not necessary the interior of the polygon. A n-sided convex polygon is formed by the intersection of interior half-planes of n edges. We can use the fact that two polygons do no overlap if there exists a separating line passing through one edge in the boundary of either face. The polygons overlap, if and only if *none* of the edges in the boundary of both polygons forms a separating line.

For our own application, we are interested to verify whether a face A's projection down to the plane containing the face B overlaps with the face B, where A and B are *parallel* and in R^3. To find a separating line between two such polygons in R^3, we check if there exists a supporting plane passing through an edge in the boundary of one polygon such that all vertices of the other polygon lies in the "exterior half-space" of the supporting plane. The supporting plane is perpendicular the polygon and containing an edge in the polygon's boundary. This approach runs in $O(N^2)$ time where N is the number of edges of each polygon.

An elegant approach which runs in $O(M+N)$ can be found in [66, 71], where the two polygons A and B each has M, N vertices respectively. This approach not only determines whether two convex polygon intersect or not, it also finds the intersections between them. The basic approach is to "advance" along the edges of A and B to find the "internal chain" of vertices which form the intersection polygon. Please refer to [66, 71] for all the details.

4.3 Analysis of the Algorithm

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 subdivision preprocessing, all local tests run in constant time since each feature now has constant number of neighboring features. The only exception to this is when f_A is a face, and f_B lies under the plane containing f_A (or vise versa). In this case, we can not use a local feature change, because this may lead to the procedure getting stuck in a loop. Geometrically, we are moving around the local minimum of the distance function, in which we may become trapped. When this situation occurs, we search among all the features of object A to find a closest feature to the f_B[3]. 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 and given a reasonable step size (not more than a few degrees of rotation). So this situation can only occur when the algorithm is first called on an arbitrary pair of features.

The algorithm can take any arbitrary pair of features of two polyhedra and find a 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 expected constant time to update a closest feature 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. A pseudo code is given in Appendix **??**. The completeness proof of this algorithm is presented in Sec. 5 and it should give readers a better insight to the algorithm.

4.4 Expected Running Time

Our approach works well by taking advantage of "continuity of motion" or geometric coherence between discrete time steps. When the motion is abruptly large, the performance doesn't achieve the potential of this algorithm though it still typically finds a pair of closest features in sub-linear time. One of the frequently asked questions is how to choose step size so that we can pre-

[3]We can also use a very expensive *preprocessing* step to find a feature on the opposite side of the polyhedron of the current face. But, this involves searching for all facets whose the outward normals pointing in the opposite direction, i.e. the dot product of the two face outward normals is negative.

serve the geometric coherence to exploit the locality of closest feature pairs in small discrete steps. This question can be addressed indirectly by the following analysis:

For relatively spherical objects, the expected running time t is proportional to $\sqrt{n}$. For cylindrical objects, the expected running time t is proportional to n, where n is the number of faces (or vertices) per polyhedron.

This is derived from the following reasoning: Given a tessellated sphere which has N faces around its equator, when the objects are rotated X degrees, it takes roughly $\frac{k*X}{360/N} = K*N$ steps to update the closest feature pair, where k and K are the adjustment constants. And the total number of features (vertices) for the sphere is $O(N^2)$. Therefore, the number of steps (or accumulated running time) is proportional to $\sqrt{n}$, where $n = O(N^2)$ is the total number of features (vertices or faces). A similar argument holds for the cylindrical objects.

With this observation, we can set the step size to preserve the coherence at each step. However, it is not necessary since the overall runtime is governed by the number of intermediate feature pairs traversed in the process and the algorithm can always find the pair of closest features, given any step size.

5 Proof of Completeness

The algorithm takes any pair of features on two polyhedra and returns a pair of closest features by iteratively checking and changing features. Through each applicability test, the algorithm steps closer and closer to a pair of closest features. That is, after each applicability check, the distance between the updated feature-pair is smaller than that of the previous feature-pair if a switch of feature-pairs is made. To verify if a given pair of features is the closest pair, each step takes running time proportional to the number of neighboring features (boundaries and/or coboundaries of the features). Since each feature has a constant number of neighboring features, each verification step takes only constant time in all cases except when a feature lies beneath a face and inside the face's Voronoi region (this corresponds to a local minimum of distance function). Thus we call this pair of features as an "exceptional" feature pair. This situation only occurs after a large motion in a single step when the actual closest feature of A is on the other (or opposite) side of the last closest feature on A.

Let n be the number of features from each object. If the algorithm never encounters a local minimum, the algorithm will return a pair of closest feature in $O(n^2)$ time since there are at most n^2 feature-pairs to verify using constant-time applicability criteria and other subroutines. If the algorithm runs into an exceptional pair of features, then it is necessary to invoke a linear-time routine to continue the verification process. We will show later that the algorithm calls the linear-time routine at most $2n$ times. Hence, the algorithm converges in $O(n^2)$ time.

Lemma 1 Each applicability test and closest feature-pair verifying subroutine will necessarily return a new pair of features closer in distance than the previous pair when a switch of feature-pairs is made.

Proof:

The algorithm works by finding a pair of closest features. There are two categories: (1) non-degenerate cases – vertex-vertex, vertex-edge, and vertex-face (2) degenerate cases – edge-edge, edge-face, and face-face.

For non-degenerate and edge-edge cases, the algorithm first takes any pair of features f_A and f_B, then find the nearest points P_A and P_B between them. Next, it checks each nearest point against the applicability constraints of the other feature. The distance between two features is the distance between the nearest points of two features, which is the same as the distance between one nearest point to the other feature. Therefore, if we can find another feature f'_A closer to P_B or another feature f'_B closer to P_A, we have shown the distance between each candidate feature pair is less after the switch.

Please note that edge-edge is a special degenerate case, since it can possibly have either one pair or many pairs of nearest points. When a pair of edges has one pair of nearest points, it is considered as all other non-degenerate cases. When both edges are parallel, they can possibly have infinitely many pairs of nearest points. In such a case, we still treat it as the other non-degenerate cases by choosing the nearest points to be the midpoints of the line segments which contains the entire sets of nearest points between two edges. In this manner, the edge-edge degenerate case can be easily treated as in the other non-degenerate cases and the proof of completeness applies here as well.

The other two degenerate cases must be treated differently since they may contain infinitely many closest point pairs. We would like to recall that edges and faces are treated as open subsets of lines and planes respectively. That is, the edge is considered as the set of points between two endpoints of the edge, excluding the head and the tail of the edge; similarly, a face is the set of points interior to its boundary (excluding the edges and vertices in its boundary). This guarantees that when a switch of features is made, the new features are *strictly* closer.

We will first show that each applicability test returns a pair of candidate features closer in distance than the previous pair when a switch of feature pairs is made. Then, we show the closest feature verifying subroutines for both edge-face and face-face cases necessarily return a closer pair of features when a switch of feature pairs is made.

(I) Point-Vertex Applicability Criterion

If vertex V is truly the closest feature to point P, then P must lie within the region bounded by the planes which are perpendicular to the coboundaries of V (the edges touching V). If so, the shortest distance between P and the other object is clearly the distance between P and V (by the definition of Voronoi region) . When P lies outside one of the plane boundaries, say C_{E_1}, the constraint plane of an edge E_1 touching V, then there is at least one point on E_1 closer to P than V itself, i.e. the distance between the edge E_1 (whose constraint is violated) and P is shorter than the distance between V and P. This can be seen from Fig.8. When a point P lies directly on the bisector C_{E_1} between V and E_1, then P is equi-distant from both features; else, P is closer to V if P is inside of V's Voronoi region, and vice versa. Therefore, each Point-Vertex Applicability test is guaranteed to generate a pair of features that is closer in distance than the previous pair (which fails the Point-Vertex Applicability test), when a switch of feature pairs occurs.

(II) Point-Edge Applicability Criterion

If edge E is really the closest feature to point P, then P must lie within the region bounded by the four constraint planes generated by the coboundary (the left and right faces) and boundary (the two end points) of E. This is the Voronoi region of E. If this is the case, the shortest distance between P and the other object is the distance between P and E (see Fig.3 and Sec. 3.2 for the construction of point-edge constraint planes.) When P fails an applicability constraint, say C_F, imposed by the left (or right) face of the

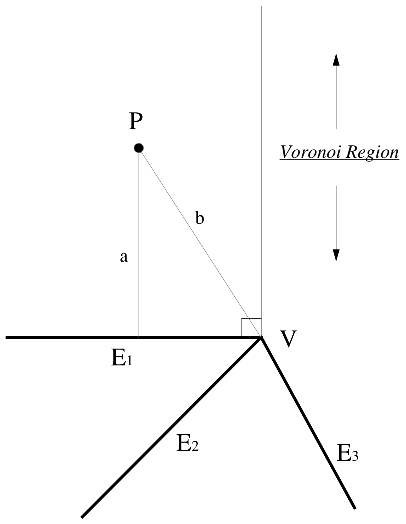


Figure 8: A Side View for Point-Vertex Applicability Criterion Proof

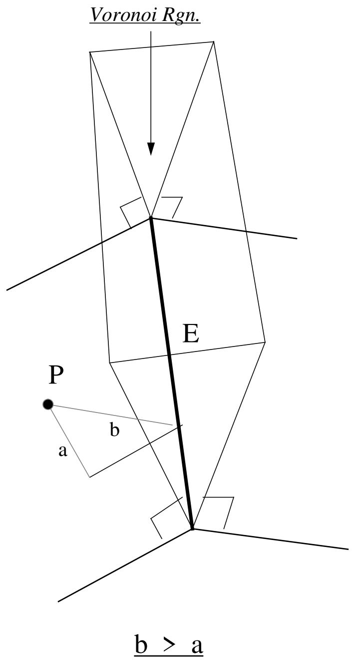


Figure 9: An Overhead View for Point-Edge Applicability Criterion Proof

edge E, there is at least one point on the corresponding face closer to P than E, i.e. the distance between P and the corresponding face is shorter than the distance between P and E (as in Fig.9). If P fails the applicability criterion C_V imposed by the head (or tail) of E, then P is closer to the corresponding endpoint than to E itself (as in Fig.10). Therefore, the distance between the new pair of features is guaranteed to be shorter than that of the previous pair which fails the Point-Edge Applicability Criterion, when a switch of feature pairs is made.

(III) Point-Face Applicability Criterion

If the face F is actually the closest feature to a point P, then P must lie within the prism bounded by the constraint planes which are orthogonal to F and containing the edges in F's boundary and above F (by the definition of Voronoi region and point-face applicability criterion in Sec. 3.3). If P fails one applicability constraint, say C_{E_1} imposed by one of F's edges, say E_1, then E_1 is closer to P than F itself.

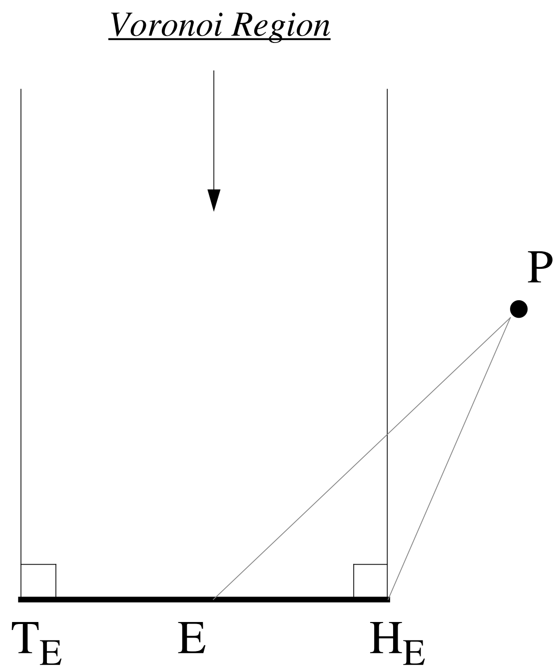


Figure 10: A Side View for Point-Edge Applicability Criterion Proof

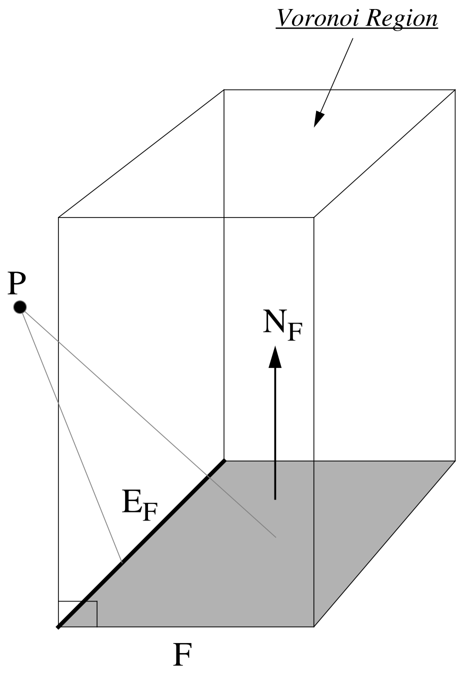


Figure 11: A Side View for Point-Face Applicability Criterion Proof

When a point lies beneath a face F and within the prism bounded by other constraint planes, it is possible that P (and of course, the object A which contains P) lies inside of the object B containing F. But, it is also possible that P lies out side of B and on the other side of B from F. This corresponds to a local minimum of distance function. It requires a linear-time routine to step to the next feature-pair. The linear-time routine enumerates all features on the object B and searches for the closest one to the feature f_A containing P. This necessarily decreases the distance between the closest feature candidates base upon the minimum distance information during the search.

Therefore, we can conclude that the distance decreases when a switch of feature pairs is made as the result of Point-Face Applicability Test.

(IV) Verifying Subroutines in Edge-Face Case

With the exception of edge-edge (as mentioned earlier in the beginning of the proof), only edge-face and face-face may have multiple pairs of closest points and require a more complex treatments. Whereas, the closest feature verification for the other non-degenerate cases and edge-edge solely depends on point-vertex, point-edge, and point-face applicability tests which we have already studied. Now we will examine each of these two cases closely.

• *Edge-Face* (E, F)
The indentation corresponds to the level of nested loops in the pseudo code (Appendix **??**). Please cross-reference if necessary.

Given an edge E and a face F, the algorithm first checks **IF E AND F ARE PARALLEL**, i.e. the two endpoints H_E and T_E are equi-distant from the face F (i.e. their signed distance is the same). The *signed* distance between H_E and F can be computed easily by $H_E \cdot N_F$ where N_F is F's unit outward normal and similarly for the signed distance between F and H_T.

THEN, when the head H_E and tail T_E of the edge E are equi-distant from the face F, the subroutine checks **IF E INTERSECTS F'S PRISM REGION**, (i.e. the region bounded by F's constraint planes constructed from the edges in F's boundary).

> **THEN**, if E intersects the prism region of F, the algorithm checks **IF THE HEAD H_E (OR TAIL T_E) OF E LIES ABOVE THE FACE F.**
>
> > **THEN**, it will check **IF THE FACE OUTWARD NORMAL N_F IS "BOUNDED" BY THE NORMALS OF THE EDGE'S LEFT AND RIGHT FACES**, say N_L and N_R respectively, to verify that the face F lies inside the Voronoi region of the edge E, not in the Voronoi region of E's left or right face (F_L or F_R). This is done by checking if $(N_L \times \vec{E}) \cdot N_F > 0$ and $(\vec{E} \times N_R) \cdot N_F > 0$, where N_F is the outward normal of the face F, $\vec{E}$ is the edge vector (as described in Sec. **??**), and N_R, N_L denote the outward normal of the edge's right and left face respectively.
> >
> > > **THEN**, the edge E and the face F will be returned as the closest feature pair, since E

and F lie within the Voronoi region of each other.

ELSE, it returns the right or left face (F_R or F_L) of E, which ever is closer to F.

ELSE, when the head H_E (implies E also) lies beneath F, then it will invoke a subroutine *e-find-min* which finds the closest feature on the polyhedron B containing F to E by enumerating all the features on B. Then the algorithm will return E and this new feature f_B as the next candidates for the closest feature pair verification.

ELSE, E does not intersect the prism defined by F's boundary edges, then there is at least one edge of F's boundary which is closer to E than the interior of F is to E (by the definition of Voronoi region). So, when the test is failed, it calls a constant time routine (which takes time proportional to the constant number of face's boundary) and returns the edge E and the closest edge or vertex, say f_F, on the F's boundary to E.

ELSE, the algorithm checks **IF ONE END POINT LIES ABOVE THE FACE F AND THE OTHER LIES BENEATH F.**

THEN, if so there is one edge on F's boundary which is closer to E than the interior of F. So, the algorithm will find the closest edge or vertex f_B, which has the minimum distance to E among all edges and vertices in F's boundary, and return (E, f_B) as the next candidate pair.

ELSE, one endpoint H_E (or T_E) of E is closer (the magnitude $\mid V_E \cdot N_F \mid$ instead of the signed distance is used for comparison here) to F than the other endpoint, the algorithm will check **IF THE CLOSER ENDPOINT V_E LIES INSIDE OF F's PRISM REGION** (bounded by constraint planes generated by edges in F's boundary), $Cell_F$, as in the vertex-face case.

THEN, if closer endpoint V_E lies inside of F's prism region, then the algorithm will next verify **IF THIS ENDPOINT LIES ABOVE THE FACE** F (to detect a local minimum of distance function).

THEN, if V_E (the closer endpoint to F) is above F the algorithm next verifies **IF THE CLOSEST POINT P_F ON F TO V_E SATISFIES V_E'S APPLICABILITY CONSTRAINTS**.

THEN, if so the algorithm returns V_E and F as the closest feature pair.

ELSE some constraint plane in V_E's Voronoi cell $Cell_{V_E}$ is violated. This constraint plane is generated by an edge, say E', in V_E's coboundary. This edge E' is closer to F, so the algorithm will return (E', F) as the next candidate pair.

ELSE, if the closer endpoint V_E of E lies beneath F, then the algorithm finds the closest feature f_B on the polyhedron B to V_E The algorithm returns (V_E, f_B) as the next candidate feature pair.

ELSE, if the closer endpoint V_E of E to F lies outside of F's prism region, then there is one edge on F's boundary which is closer to E than the interior of F. So, the algorithm will find the closest edge or vertex f_B, which has the minimum distance to E among all edges and vertices in F's boundary, and return (E, f_B) as the next candidate pair.

(V) Verifying Subroutines in Face-Face Case

• *Face-Face* (F_A, F_B)
The numbering system (n) in this proof is annotated for easy reference within the proof. The indentation corresponds to the level of nested loops in the pseudo code (Appendix **??**).

(0) Given two faces, F_A and F_B, the algorithm first checks **IF THEY ARE PARALLEL**.

(1) **THEN**, if they are parallel, it invokes a subroutines which runs in constant time (proportional to the product of numbers of boundaries between two faces, which is a constant after subdivision) to check **IF THE ORTHOGONAL PROJECTION OF** F_A **ON TO THE PLANE OF** F_B **OVERLAPS** F_B (i.e. F_A intersects the prism region of F_B).

(2) **THEN**, if they overlapped, the algorithm next checks **IF** P_A (implies F_A as well, since two faces are parallel) **LIES ABOVE** F_B, where P_A is one of the points on F_A and its projection down to the F_B is in the overlap region.

(3) **THEN**, if P_A (thus F_A) lies above F_B, the algorithm next checks **IF** P_B **(THUS** F_B**) ALSO LIES ABOVE** F_A, where P_B is one of the points and its projection down to the F_B is in the overlap region.

(4) **THEN**, if P_B (thus F_B) is above F_A and vice versa, we have a pair of closest points (P_A, P_B) satisfy the respective applicability constraints of F_B and F_A. So, P_A and P_B are the closest points on A and B; and the features containing them, F_A and F_B, will be returned as the closest features.

(5) **ELSE**, if P_B (thus F_B) lies beneath F_A, then the algorithm finds the closest feature f_A on polyhedron A containing F_A to F_B by enumerating all features of A to search for the feature which has minimum distance to F_B. So, the new pair of features (f_A, F_B) is guaranteed to be closer than (F_A, F_B).

(6) **ELSE**, if P_A (thus F_A) lies beneath F_B, the algorithm finds the closest feature f_B on polyhedron B containing F_B to F_A and returns (F_A, f_B) as the next candidate feature pair. This pair of new features has to be closer than F_A to F_B since f_B has the shortest distance among all features on B to F_A.

(7) **ELSE**, if F_A and F_B are parallel but the projection of F_A does not overlap F_B, a subroutine enumerates all possible combination of edge-pairs (one from the boundary of each face respectively) and computes the distance between each edge-pair to find a pair of closest edges which has the minimum distance among all. This new pair of edges (E_A, E_B) is guaranteed to be closer than the two faces, since two faces F_A and F_B do not contain their boundaries.

(8) **ELSE**, if F_A and F_B are not parallel, this implies there is at least one vertex or edge on F_A closer to F_B than F_A to F_B *or* vice versa. So, the algorithm first finds the closest vertex or edge on F_A to the plane containing F_B, Φ_B.

(9) **VERTEX:** if there is only a single closest vertex V_A to the plane containing F_B, V_A is closer to F_B than F_A is to F_B since two faces are not parallel and their boundaries must be closer to each other than their interior. Next, the algorithm checks if V_A lies within the Voronoi region of F_B.

(10) **THEN**, if so V_A and F_B will be returned as the next candidate pair.

(11) **ELSE**, the algorithm finds the closest vertex (or edge) on F_B to Φ_A, and proceeds as before.

(12) **VERTEX:** if there is only a single vertex V_B closest to the plane Φ_A containing F_A, the algorithm checks if V_B lies inside of F_A's Voronoi region bounded by the constraint planes generated by the edges in F_A's boundary and above F_A.

(13) **THEN**, if V_B lies inside of F_A's Voronoi region, then F_A and V_B will naturally be the next candidate pair. This new pair of features is closer than the previous pair as already analyzed in (10).

(14) **ELSE**, the algorithm performs a search of all edge pairs from the boundary of F_A and F_B and returns the closest pair (E_A, E_B) as the next candidate feature pair.

(15) **EDGE:**, similarly if the closest feature on F_B to Φ_A is an edge E_B, the algorithm once again checks if E_B cuts F_A's prism to determine whether F_A and E_B are the next candidate features.

(16) **THEN**, the algorithm checks **IF THE END POINT OF E_B LIES ABOVE F_A.**

(17) **THEN**, the new pair of candidate features is F_A and E_B since E_B lies within F_A's Voronoi region.

(18) **ELSE**, the algorithm searches for the closest feature f_A which has the minimum distance among all other features on A to E_B and returns (f_A, E_B) as the next pair of candidate features.

(14) **ELSE**, if not, then this implies that neither face contains the closest points since neither F_A nor F_B's Voronoi regions contain a closest point from the other's boundary. So, the algorithm will enumerate all the edge-pairs in the boundaries of F_A and F_B to find the pair of edges which is closest in distance. This new pair of edges (E_A, E_B) is guaranteed to be closer than the two faces.

(19) **EDGE:** if there are two closest vertices on F_A to Φ_B, then the edge E_A containing the two vertices is closer to Φ_B than F_A

is. Next, the algorithm checks **IF** E_A **INTERSECTS THE PRISM REGION OF** F_B, as in the edge-face case.

(20) **THEN**, the algorithm checks **IF THE END POINT OF** E_A **LIES ABOVE** F_B.

(21) **THEN**, if so the edge E_A and F_B will be returned as the next candidate pair.

(22) **ELSE**, the algorithm searches for the closest feature f_B which has the minimum distance among all other features on B to E_A and returns (E_A, f_B) as the next pair of candidate features.

(23) **ELSE**, if this is not the case, then the algorithm finds the closest vertex (or edge) on F_B to the plane containing F_A, say Φ_A and proceeds as before, from block (11) to block (18).

In either degenerate case (edge-face or face-face), a pair of features closer in distance is always provided when a switch of feature pairs is made. All the verifying subroutines run in *constant* time (proportional to the number of boundary or coboundary of features *or* the product of numbers of boundaries between two features), except where an edge lies beneath a face or when a face lies beneath a face. In such local cases, a linear-time routine is invoked.

To sum up, all the applicability tests and closest feature-pair verifying subroutines generate a pair of new features closer in distance than the previous pair when a switch of feature-pairs is made. □

Corollary The algorithm takes $O(n^3)$ time to find the closest feature pair, since each pair of features is visited at most once, and each step takes at most linear time.

$O(n^3)$ is a rather loose upper bound. We can find a tighter bound on the running time if we study the local minimum case in greater details. As a result of a new analysis, we claim in Lemma 2 that the algorithm should converge in $O(n^2)$ time. The proof is stated below.

Lemma 2 The algorithm will converge in $O(n^2)$ time.

Proof:

Except for the case of a local minimum which corresponds to the situation where one feature lies under the plane containing the other face, all the applicability tests and subroutines run in constant time. Thus the overall running time is $O(n^2)$ in this case, since the algorithm can only exhaustively visit $O(n^2)$ pairs of features and no looping can possibly exist if it doesn't encounter local minima (by Lemma 1). As mentioned earlier, when a local minimum is encountered, a linear-time routine is invoked. But it is not necessary to call this linear-time routine more than $2n$ times to find the closest feature-pair. The reasons are the following:

We claim that we will not revisit the same feature twice in a linear time check. We will prove by contradiction. Assume that the linear time routine is called to find a closest feature in B to f_A. The routine will never again be called with (f_A, B) as argument. Suppose it was, i.e. suppose that f_B was the result of the call to the linear time routine with input (f_A, B) and after several iterations from $(f_A, f_B) \rightarrow \ldots \rightarrow (f'_A, f'_B)$ we come to a feature pair (f_A, f'_B) and that the linear routine is invoked to find the closest feature on B to f_A. Once again, f_B must be returned as the closest feature to f_A. But, this violates Lemma 1 since the distance must strictly decrease when changes of features are made, which is not true of the regular sequence $(f_A, f_B) \rightarrow \ldots \rightarrow (f_A, f'_B) \rightarrow \ldots \rightarrow (f_A, f_B)$.

Similarly, the routine will not be called more than once with (f_B, A) as arguments. Thus, the linear-time routine can be called at most $2n$ times, where n is the total number of features of object B (or A).

Therefore, the closest-feature algorithm will converge in $O(n^2)$ time, even in the worst case where local minima are encountered. □

6 Numerical Experiments

The algorithm for convex polyhedral objects described in this chapter has been implemented in both Sun Common Lisp and ANSI C for general convex objects. (The extension of this algorithm to non-convex objects is presented in the next chapter.) The input data are arbitrary pairs of features from two given polyhedral objects in three dimensional space. The routine outputs a

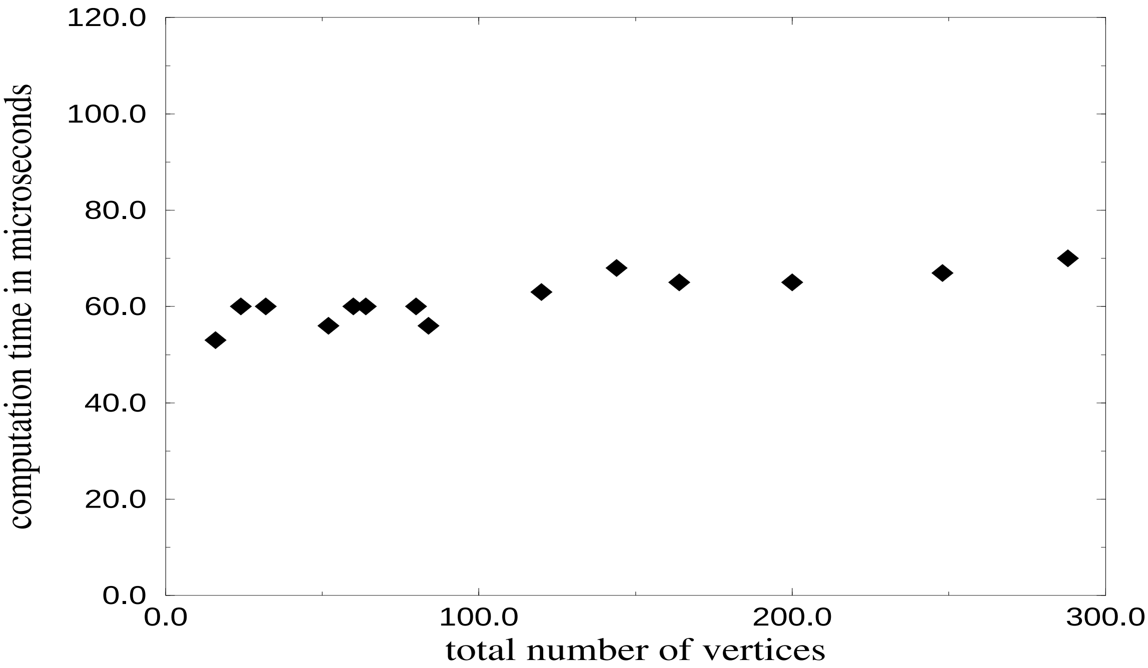


Figure 12: Computation time vs. total no. of vertices

pair of the closest features (and a pair of nearest points) for these two objects and the Euclidean distance between them.

Numerous examples in three dimensional space have been studied and tested. Our implementation in ANSI C gives us an expected constant time performance of 50-70 usec per object pair of arbitrary complexity on SGI Indigo2 XZ (200-300 usec per object pair on a 12.5 Mips 1.4 Mega flops Sun4 Sparc Station) for relative small rotational displacement. Fig.12 shows the expected constant time performance for polygonized spherical and cylindrical objects of various resolutions. Each data point is taken as the average value of 1000 trials with the relative rotation of 1 degree per step. (The closest features rarely change if two objects are translated along the line connecting the closest feature pair.)

We would like to mention that the accumulated runtime for 1000 trials shows slightly increasing trend for total vertices over 1000, this is due to the fact that the number of feature pair changes for the same amount of rotation increases if the resolution of objects increases. For spherical and most objects, the runtime is roughly $O(\sqrt{n})$ where n is number of vertices for each object with a extremely small constant. For the cylindrical objects where the total number of facets to approximate the quadratic surface is linear with the number of vertices, the total accumulated runtime is expectedly sub-linear with the total number of vertices.

For two convex hulls of teapots (each has 1047 vertices, 3135 edges, and 2090 faces), the algorithm takes roughly 50, 61, 100, 150 usec for 1, 5, 10, 20 degrees of relative rotation and displacements. Note that when the relative movement between 2 objects becomes large, the runtime seems to increase as well, this is due to the fact that the geometric coherence doesn't hold as well in the abrupt motions. Nevertheless, the algorithm continues to perform well compared to other algorithms.

The examples we used in our simulation include a wide variety of polytopes: cubes, rectangular boxes, cylinders, cones, frustrums, and a Puma link of different sizes as shown in Fig.13. In particular, the number of facets or the resolution for cylinders, cones and frustrums have been varied from 12, 20, 24, 48, up to 120 in order to generate a richer set of polytopes for testing purpose. Due to the programming language's nature, Lisp code usually gives a longer run time (roughly 10 times slower than the implementation written in ANSI C).

To give readers a feeling of speed difference the expected run time obtained from our Lisp code is 3.5 msec which is roughly 10 times slower than our C implementation (from the same Sun4 SPARC station described above), independent of object complexity. The experiment results obtained from our Lisp code are briefly summarized in Table 1. M1 and M2 stand for the number of vertices for each object respectively; and N stands for the total number of vertices between the two objects. The average CPU time is taken over 50 trials of experimental runs. (Each trial picks all 18 arbitrary permuted combination of feature pairs to initialize the computation and calculates the average run time based upon these 18 runs.) With initialization to the previous closest feature, the routine can almost always keep track of the closest features of two given polytopes in expected constant time (about 3 to 4 msec).

Without initialization (i.e. no previous closest feature pair is given), 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,

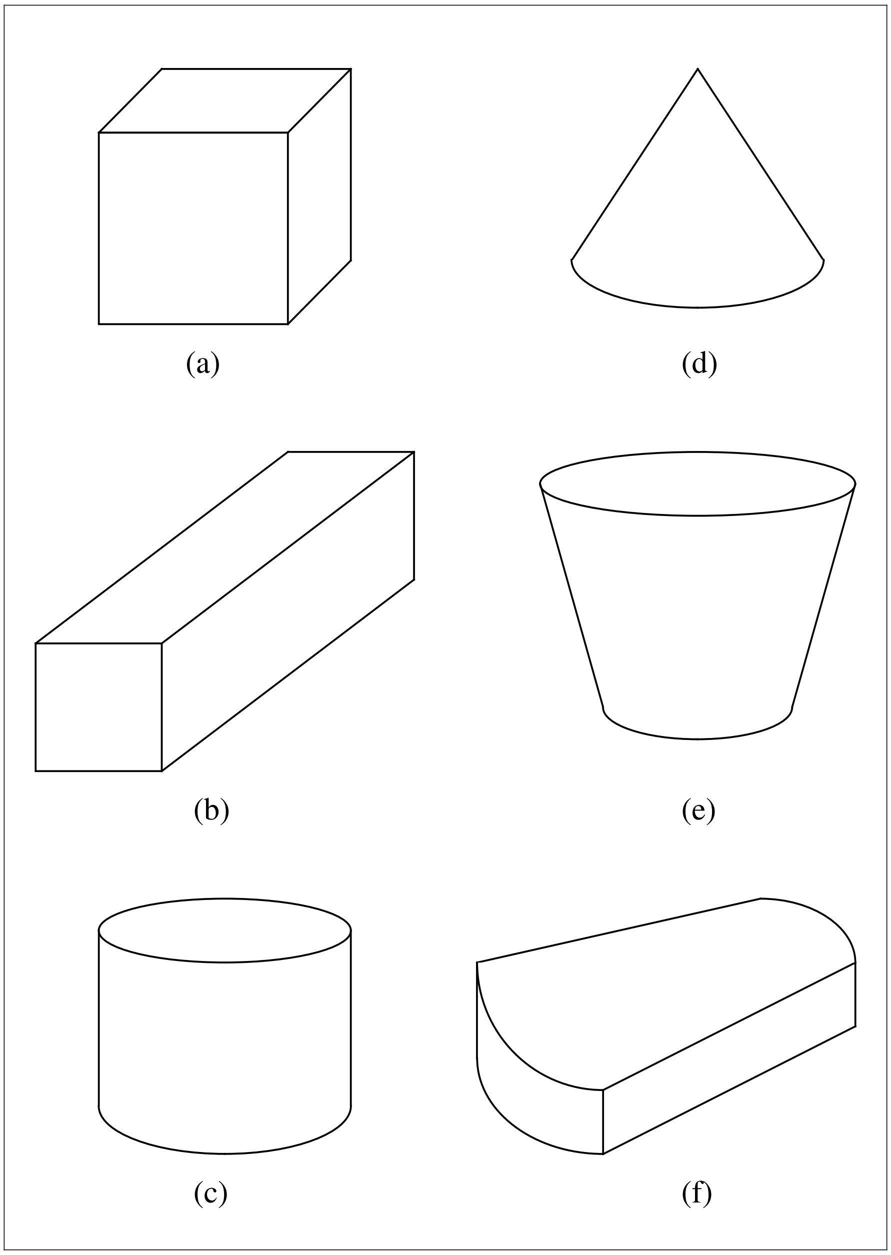


Figure 13: Polytopes Used in Example Computations

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

we believe our algorithm compares very favorably with other algorithms designed for distance computations or collision detection. Since the nature of problem is not the same, we cannot provide exact comparisons to some of the previous algorithms. But, we can briefly comment on the number of operations involved in a typical calculation. For an average computation to track or update a pair of closest features, each run takes roughly 50-70 usec on a SGI Indigo2 XZ or 92-131 arithmetic operations (roughly 12-51 operations to find the nearest points, 2 x 16 operations on 2 point transformations, 2 x 4 x 6 operations for two applicability tests and each involves 4 inner product calculations) independent of the machines. By comparing the number of arithmetic operations in the previous implemented algorithms, we believe that our implementation gives better performance and probably is the fastest implemented collision detection algorithm. (Please see [65], [13], [17], [40], [39], [38], [72], and [90].)

This is especially true in a dynamic environment where the trajectories are not known (nor are they in closed form even between impacts, but are given by elliptic integrals). The *expected constant time* performance comes from the fact that *each update* of the closest feature pair involves only the constant number of neighboring features (after the initialization and preprocessing procedures). The speed of algorithm is an attribute from the *incremental* nature of our approach and the geometric coherence between successive, discrete movements.

7 Dynamic Collision Detection for Convex Polyhedra

The distance computation algorithm can be easily used for detecting collision while the objects are moving. This is done by iteratively checking the distance between any pair of moving objects. If the distance is less than or equal to zero (plus ϵ - a small safety margin defined by the user), then a collision is declared. We can set the step size of the algorithm once we obtain the initial position, distance, velocity and acceleration among all object pairs. We also use a simple queuing scheme to reduce the frequency of collision checks by relying on the fact that only the object pairs which have a small (Chapter **??**).

Furthermore, we will not need to transform all the Voronoi regions and all features of polyhedra but only the closest points and the new candidate features (if necessary), since local applicability constraints are all we need for tracking a closest pair of closest features. The transformation is done by taking the relative transformation between two objects. For example, given two objects moving in space, their motions with respect to the origin of the world frame can be characterized by the transformation matrices T_A and T_B respectively. Then, their relative motion can be represented by the homogeneous relative transformation $T_{AB} = T_B^{-1} T_A$.

References

[1] L. E. Andersson and N. F. Steward. Maximal distance for robotic simulation: the convex case. *Journal of Optimization Theory and Applications*, 57(No. 2):215–222, 1988.

[2] C. Bajaj and T. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. of Comput.*, (No. 2):339–364, 1992.

[3] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[4] D. Baraff. Issues on computing contact forces for non-penetrating rigid bodies. *Algorithmica*, Vol. 10(2-4):292–352, Aug/Sept/Oct 1993.

[5] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, 26(2):303–308, July 1992.

[6] J. Barraquand, B. Langlois, and J-C. Latombe. Robot motion planning with many degrees of freedom and dynamic constraints. In *Proceedings 5th Int. Symp Robotics Research*, Tokyo, Japan, 1989.

[7] R. Barzel and A. Barr. A modeling system based on dynamic constraints. *ACM Computer Graphics*, 22(4):31–39, 1988.

[8] D. N. Bernstein. The number of roots of a system of equations. *Funktsional'nyi Analiz i Ego Prilozheniya*, 9(3):1–4, Jul-Sep 1975.

[9] J. E. Bobrow. A direct minimization approach for obtaining distance between convex polyhedra. *International Journal of Robotics Research*, 8(No. 3):65–67, 1989.

[10] W. Bouma and Jr. G. Vanecek. Collision detection and analysis in a physically based simulation. In *Proceeding of the Seconde Eurographics Workshop on Animation and Simulation*, 1992. Vienna Austria.

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

[12] S. A. Cameron. A study of the clash detection problem in robotics. *Proc. IEEE ICRA*, pages pp. 488–493, 1985.

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

[14] J. Canny. Computing roadmaps of general semi-algebraic sets. In *AAECC-91*, pages 94–107, 1991.

[15] J. Canny and B. Donald. Simplified voronoi diagrams. *Discret Comput. Geometry*, pages 219–236, 1988.

[16] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. 10th Intern. Symp. on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 89–104. Springer Verlag, May 1993. Lect. Notes in Comp. Science 263.

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

[18] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.

[19] J. F. Canny. Constructing roadmaps of semi-algebraic sets I: Completeness. *Artificial Intelligence*, 37:203–222, 1988.

[20] J. F. Canny and M. C. Lin. An opportunistic global path planner. *Algorithmica, Special Issue on Computational Robotics*, Vol. 10(2-4):102–120, Aug/Sept/Oct 1993.

[21] J.F. Canny. Generalized characteristic polynomials. *Journal of Symbolic Computation*, 9(3):241–250, 1990.

[22] B. Chazelle. Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[23] B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete & Comput. Geom.*, (5):505–526, 1990.

[24] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. Exact collision detection for interactive, large-scaled environments. Tech Report #TR94-005, 1994. University of North Carolina, Chapel Hill.

[25] James F. Cremer and A. James Stewart. The architecture of newton, a general-purpose dynamics simulator. In *IEEE Int. Conf. on Robotics and Automation*, pages 1806–1811, May 1989.

[26] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6(3):pp. 381–392, 1985.

[27] D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. In *Proc. 17th Internat. Colloq. Automata Lang. Program*, Lecture Notes in Computer Science 443, pages 400–413. Springer-Verlag, 1990.

[28] B. R. Donald. Motion planning with six degrees of freedom. Master's thesis, MIT Artificial Intelligence Lab., 1984. AI-TR-791.

[29] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *ACM Computer Graphics*, 26(2):131–139, 1992.

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

[31] H. Edelsbrunner. A new approach to rectangle intersections, part i&ii. *Intern. J. Computer Math.*, 13:pp. 209–229, 1983.

[32] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1988.

[33] I. Emiris and J. Canny. A general approach to removing degeneracies. In *IEEE FOCS*, pages 405–413, 1991.

[34] S. F. Fahlman. A planning system for robot construction tasks. *Artifical Intellignece*, 5:pp. 1–49, 1974.

[35] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press Inc., 1990.

[36] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3:295–311, 1986.

[37] Jr. G. Vanecek. Brep-index: A multi-dimensional space partitioning tree. *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD Applications*, pages 35–44, 1991. Austin Texas.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] G.H. Golub and C.F. Van Loan. *Matrix Computations*. John Hopkins Press, Baltimore, 1989.

[43] J. K. Hahn. Realistic animation of rigid bodies. *ACM Computer Graphics*, 22(4):pp. 299–308, 1988.

[44] Mark Hall and Joe Warren. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–42, November 1990.

[45] B. V. Herzen, A. H. Barr, and H. R. Zatz. Geometric collisions for time-dependent parametric surfaces. *ACM Computer Graphics*, 24(4), August 1990.

[46] C. Hoffmann. *Geometric & Solid Modeling*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1989.

[47] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *The International Journal of Robotics Research*, 2(4):77–80, 1983.

[48] Kass, Witkin, Baraff, and Barr. An introduction to physically based modeling. Course Notes 60, 1993.

[49] J. B. Keller. Impact with friction. *Journal of Applied Mathematics*, Vol. 53, March 1986.

[50] O. Khatib. Real-time obstable avoidance for manipulators and mobile robots. *IJRR*, 5(1):90–98, 1986.

[51] J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[52] A. Lingas. The power of non-rectilinear holes. In *Proc. 9th Internat. Colloq. Automata Lang. Program.*, volume 140 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, 1982.

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

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

[55] D. Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics*. PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[56] D. Manocha. Solving polynomial systems for curve, surface and solid modeling. In *ACM/SIGGRAPH Symposium on Solid Modeling*, pages 169–178, 1993. Revised version to appear in IEEE Computer Graphics and Applications.

[57] D. Manocha and J. F. Canny. Efficient teniques for multipolynomial resultant algorithms. *Proceedings of ISSAC'91*, 1991. Bonn, Germany.

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

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

[60] N. Megiddo and A. Tamir. Linear time algorithms for some separable quadratic programming problems. *Operations Research letters*, 13(4):203–211, 1993.

[61] J. Milnor. On the betti numbers of real varieties. *Proc. Amer. Math. Soc.*, 15:275–280, 1964.

[62] B. Mirtich and J. Canny. Impusle-based, real time dynamic simulation. Submitted to ACM SIGGRAPH, 1994. University of California, Berkeley.

[63] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *ACM Computer Graphics*, 22(4):pp. 289–298, 1988.

[64] A. P. Morgan. Polynomial continuation and its relationship to the symbolic reduction of polynomial systems. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 23–45, 1992.

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

[66] J. O'Rourke, C.-B Chien, T. Olson, and D. Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19:384–391, 1982.

[67] J. O'Rourke and K. J. Supowit. Some NP-hard polygon decomposition problems. *IEEE Trans. Inform. Theory*, IT-30:181–190, 1983.

[68] M. H. Overmars. Point location in fat subdivisions. *Inform. Proc. Lett.*, 44:261–265, 1992.

[69] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

[70] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *Computer Graphics*, 23(3):185–192, 1990.

[71] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

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

[73] J. Reif. *Complexity of the Mover's Problem and Generalizations*, chapter 11, pages pp. 267–281. Ablex publishing corp., New Jersey, 1987. edited by J.T. Schwartz and M. Sharir and J. Hopcroft.

[74] D. F. Rogers. *Procedural Elements for Computer Graphics*. McGraw-Hill Book Company, 1985.

[75] Edward J. Routh. *Elementary Rigid Dynamics*. 1905.

[76] J. Ruppert and R. Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. In *Proc. of the Fifth Annual Symposium on Computational Geometry*, pages 380–392. ACM, 1989.

[77] N. Sancheti and S. Keerthi. Computation of certain measures of proximity between convex polytopes: A complexity viewpoint. *Proceedings of IEEE ICRA'92*, 3:2508–2513, May 1992.

[78] J.T. Schwartz and M. Sharir. *On the 'Piano Movers' Problem, II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds*, chapter 5, pages 154–186. Ablex publishing corp., New Jersey, 1987.

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

[80] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Trans. on Computers*, C-31(No. 3), March 1982.

[81] J. Snyder, A. Woodbury, K Fleischer, B. Currin, and A. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. *Computer Graphics, Proceedings of ACM SIGGRAPH'93*, pages 321–334, August 1993.

[82] D. Sturman. A discussion on the development of motion control systems. In *SigGraph Course Notes: Computer Animation: 3-D Motion Specification and Control*, number 10, 1987.

[83] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics – SIGGRAPH'87*, (4), 1987.

[84] R. Thom. Sur l'homologie des varietes algebriques reelles. *Differential and Combinatorial Topology*, pages 255–265, 1965.

[85] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, Computer Science Department, University of North Carolina at Chapel Hill, 1989.

[86] Yu Wang and Matthew T. Mason. Modeling impact dynamics for robotic operations. In *IEEE Int. Conf. on Robotics and Automation*, pages 678–685, May 1987.

[87] C. G. Gibson K. Wirthmuller and A. A. du Plessis E. J. N. Looijenga. *Topological Stability of Smooth Mappings*. Springer-Verlag, Berlin . Heidelberg . New York, 1976.

[88] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–22, March 1990.

[89] Andrew Witkin and William Welch. Fast animation and control of nonrigid structures. *Computer Graphics*, 24(4):243–252, August 1990.

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

[91] K. Zikan and W. D. Curtis. Intersection and separations of polytopes. *Boeing Computer Services Technology*, BCSTECH-93-031, 1993. A note on collision and interference detection.