Proceedings of the 1997 IEEE
International Conference on Robotics and Automation
Albuquerque, New Mexico - April 1997

# Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra

Stephen Cameron
Oxford University Computing Laboratory
Parks Road, Oxford OX1 3QD, UK
stephen.cameron@comlab.ox.ac.uk

**Abstract**

*The problem of tracking the distance between two convex polyhedra is finding applications in many areas of robotics, including intersection detection, collision detection, and path planning. We present new results that confirm an almost-constant time complexity for an enhanced version of Gilbert, Johnson and Keerthi's algorithm, and also describe modifications to the algorithm to compute measures of penetration distance.*

## 1 Introduction

In robotics we are finally moving towards a situation whereby it is possible to apply some non-trivial algorithms for analysis, simulation and planning to realistic models of robots and environments. Examples of these algorithms include intersection detection (i.e., would two static objects occupy some common region of space?), collision detection (i.e., will an interference occur between two moving objects?), and path planning. A very useful and basic function between objects is their distance apart.

In this paper we will focus on the problem of finding the distance between known pairs of convex polyhedra. Variations have also appeared, fuelled by the fact that when objects are relatively far apart we rarely care about an exact answer, or by the fact that (in robotics) we are usually keeping track of the distance between objects as (simulated) time is stepped forward. For this *tracking problem* Lin and Canny [8] claim an 'almost constant' time complexity, and we have recently shown [4] that a minor modification to the algorithm of Gilbert, Johnson and Keerthi [6] also gives that algorithm the same theoretical time complexity. The first part of this paper then will concentrate on demonstrating this behaviour experimentally for our enhanced version of the Gilbert, Johnson and Keerthi algorithm (GJK), and in explaining some parts of our implementation that differ from that described in [6]. Both GJK and the Lin-Canny algorithm (LC) have been extended in various ways, including treating objects as collections of convex pieces, and the inclusion of non-polyhedral objects [5, 11, 10, 12]. However in the extensions the efficient solution of the basic problem is still a key issue, and we demonstrate here that the convex polyhedra case can be solved fast enough to make its special treatment worthwhile.

When two (simulated) objects interpenetrate then the distance between them decreases to zero. For collision detection, say, it is often quite acceptable to accept zero as an answer, but this solution gives no hints as to how to extricate the system from this condition. Thus *measures of interpenetration* have been defined, which can provide a numerical clue as to how to remove one object (say, a robot arm) from another. So we will also here briefly consider the computation of *MTD* (mininum translational distance), which is the size of the smallest translation required so that two objects come into contact (only) [3]. For non-penetrating objects *MTD* returns the usual Euclidean distance, and for penetrating objects it is defined to return a negative number that gives a measure of how hard it is to remove one object from the other. [2] defined a measure that gives the same answer as *MTD* over their common range of definition, and more recent definitions of penetration measures include [7, 1].

## 2 Using Configuration Space

In order to understand and compare the action of the algorithm it is helpful to recast the problem using configuration space. *MTD* is defined in terms of the auxiliary function $MTD^+$, where

$$MTD^+(A,B) = \inf_{\mathbf{t}} \{\, |\mathbf{t}| \ : \ A+\mathbf{t} \text{ is in contact with } B \,\},$$

$\mathbf{t}$ ranges over all possible translation vectors, and $A+\mathbf{t}$ indicates the object $A$ after having been translated by $\mathbf{t}$. Fig. 1 illustrates the definition, with the arrows showing the minimum translation vectors (dashed arrowed lines) to move A to place it in contact with B, or to move C to place it in contact with B. Both vectors have the same length, and so $MTD^+(A,B) =$

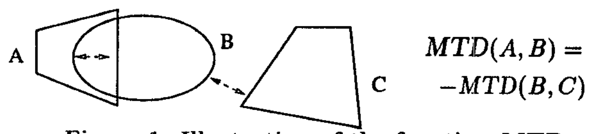

Figure 1: Illustration of the function *MTD*.

$MTD^+(B,C)$, but we distinguish the cases of overlap and non-overlap by defining *MTD* to be the value of $MTD^+$ if the objects do not overlap, and minus the value of $MTD^+$ otherwise.

This definition can be seen to give a 'reasonable' value, and the sign of *MTD* gives an easy answer to the interference detection problem, but the definition does not tell us how to compute $MTD^+$. However if we consider the *translational configuration space* (TC-space) of the objects, then we can define their translational C-space obstacle (TCSO) as

$$\{\, \mathbf{b} - \mathbf{a} \,|\, \mathbf{a} \in A \text{ and } \mathbf{b} \in B \,\}.$$

This may also be recognised as the Minkowski difference operation as used in path planning [9]. Then a result proved in [3] is that the minimum distance between two objects is the same as the minimum distance between the origin of TC-space and the TCSO: that is,

$$MTD(A,B) = MTD(\mathbf{O}, TCSO(A,B))$$

where $\mathbf{O}$ is the set containing the origin only, and so the problem of finding the distance between two $n$-dimensional point sets has been transformed into that of finding the distance between a point (the origin) and another $n$-dimensional point set.

If we take the TCSO of two convex polyhedra $P$ and $Q$, then the TCSO is another convex polyhedra, and each vertex of the TCSO correspond to the vector difference of a vertex from $P$ and a vertex from $Q$. To compute the minimum distance between $P$ and $Q$ both LC and GJK use a loop in which two surface points—one from $P$ and one from $Q$—are tracked. We call these points—which are not necessarily unique—*witness points* as when the minimum distance has been found then these points are that distance apart.

For the tracking problem we will normally be calling a distance algorithm many (i.e., hundreds or more) times, so that each call corresponds to the next increment of simulated time. When either LC or GJK called to compute the distance at some time step then it normally makes most sense if the witness points for the algorithm are initialised to be same[1] as the witness

[1]By 'same', we mean the same points relative to the objects on which they lie. These objects will in general be moving, and so the coordinates of the witness points may well have altered.

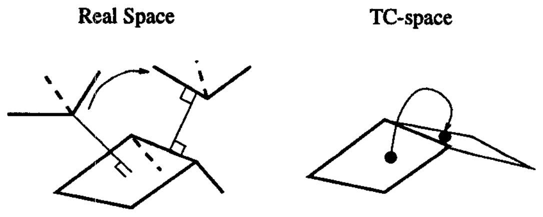

Figure 2: Examples of witness-point transitions as the objects move

points that were found at the last time-step. In this case one of the assumptions used to obtain the 'almost constant' time complexity for both algorithms is that the witness points that realise the minimum distance for the new time step will be close the seed values of the witness points.

Given a pair of witness points $(\mathbf{p},\mathbf{q})$ then each is a surface point on $P$ and $Q$, and their difference $\mathbf{q}-\mathbf{p}$ is a surface point on the TCSO, which we call a TC-witness point. Then as the polyhedra move and the witness points move from feature to (neighbouring) feature on the objects, then the witness point in TC-space moves between neighbouring features on the TCSO. This process is illustrated in Fig. 2, which shows witness points moving from a vertex-face feature pair to a nearby edge-edge pair, and thus between two neighbouring faces on the TCSO.

## 3 Overview of the Gilbert, Johnson and Keerthi Algorithm

Several items of information cached as part of the tracking process in GJK, from which the witness points can be easily computed as required. The key piece of information is a set of 1–4 pairs of vertices (one each from $P$ and $Q$) that define simplices (i.e., a point, bounded line, triangle or tetrahedron) in each object, and a corresponding simplex in the TCSO. At initialisation we can set this simplex randomly, say by picking a single vertex from $P$ and $Q$. The idea is that the simplices will contain the witness points (and TC-witness point) on their boundaries; to make this true, each call of GJK can be expressed as a very simple loop structure involving two key sub-routines:

```
while not best_simplex( $S$) do
    $S$ ← refine_features( $S$);
endwhile
```

which checks whether the simplex does contain the witness points using `best_simplex`, and if not tries to find a neighbouring simplex that does using `refine_simplex`. The details of these sub-routines will be explained below, but note that in practice the loop body is typically only executed once or twice per call to GJK when tracking, and on average 3–6 times per call at initialisation, even for polyhedra with hundreds of vertices.

### 3.1 `best_simplex`

For **`best_simplex`** we can use the fact that if the witness point is the closest to the origin then all of the points of the TCSO must lie to the far side (i.e., furthest from the origin) of a plane passing through the witness point and perpendicular to the line from the witness point to the origin. That is, if $\mathbf{x}$ is the current witness point then for all vertices $\mathbf{v}_i$ of the TCSO we have $g_{\mathbf{x}}(\mathbf{v}_i) \leq 0$, where $g_{\mathbf{x}}(\mathbf{v}) = \mathbf{x}\cdot\mathbf{x} - \mathbf{x}\cdot\mathbf{v}$. Conversely, if there exists a vertex $\mathbf{v}_j$ with $g_{\mathbf{x}}(\mathbf{v}_j) > 0$ then there is some point in the TCSO that is closer to the origin than $\mathbf{x}$ is. In particular we can look for a $\mathbf{v}_j$ which maximises $(-\mathbf{x})\cdot\mathbf{v}_i$, which is called a *support vertex* for the given direction $(-\mathbf{x})$. This can be done without explicit reference to the TCSO, as the support vertex on the TCSO is given by the vector difference of support vertices for $P$ and $Q$ (relative to the directions $\mathbf{x}$ and $-\mathbf{x}$). Thus the finding of support vertices becomes a critical part of the algorithm; in [6] this was done by looking at each vertex of both polyhedra, giving rise to a linear time complexity.

### 3.2 Enhancing GJK

An alternative to computing the support vertex function by looking at all the vertices is to use *hill climbing*, in which we start each time from the previous vertex $\mathbf{v}_{old}$ that was returned by the support vertex function (at the last time-step). Then given a new support direction we set $\mathbf{v} = \mathbf{v}_{old}$ and compare $\mathbf{x}\cdot\mathbf{v}$ with $\mathbf{x}\cdot\mathbf{v}_j$ for each neighbouring vertex $\mathbf{v}_j$ of $\mathbf{v}$ (i.e., for every vertex that is connected to $\mathbf{v}$ by an edge of the polyhedron). If each $\mathbf{x}\cdot\mathbf{v}_j \geq \mathbf{x}\cdot\mathbf{v}$ then we stop and return $\mathbf{v}$, otherwise we set $\mathbf{v}$ to a neighbouring vertex that gave a smaller value and recurse. By the convexity of the polyhedra this process will terminate at a support vertex. When tracking we only expect this process to traverse 0 or 1 edges most times, giving rise to a time complexity bounded by the degree of the vertices (normally about 3).

Hill climbing was also used in [12], and we suspect that it may have been added by other users of GJK as well. For larger convex hulls it gives a dramatic improvement in computation times over the original GJK algorithm—see §4.

### 3.3 Solving Each Simplex

We need to find the nearest point to the TC-space origin on each vertex, edge, and face (as appropriate) of the current simplex. The method used by GJK does precisely this, but is expressed in linear algebraic terms. For a simplex in TC-space defined by a set $S' = \{\mathbf{x}_i\}$ of up to 4 linearly-independent points, then the TC-witness point $\mathbf{p}_{S'}$ for the sub-space defined by $S'$ (i.e., a point, line, plane, or the whole of TC-space) can be written uniquely as $\mathbf{p}_{S'} = \sum_i \lambda_i \mathbf{x}_i$ with $\sum_i \lambda_i = 1$, and this point will lie within the simplex if all the $\lambda_i$'s are non-negative. Now note that $\mathbf{p}_{S'}$ must have the property that it is perpendicular to all lines contained in the sub-space, and in particular $(\mathbf{x}_i - \mathbf{x}_1)\cdot\mathbf{p}_{S'} = 0$ for each $i$. This, together with the constraint on the sum of the $\lambda_i$'s, gives us a system of $l$ linear equations in $l$ unknowns,

$$\begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_l \end{bmatrix} M_{S'} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}$$

where the $l \times l$ matrix $M_{S'}$ is given by

$$\begin{bmatrix} 1 & (\mathbf{x}_2-\mathbf{x}_1)\cdot\mathbf{x}_1 & \dots & (\mathbf{x}_l-\mathbf{x}_1)\cdot\mathbf{x}_1 \\ 1 & (\mathbf{x}_2-\mathbf{x}_1)\cdot\mathbf{x}_2 & \dots & (\mathbf{x}_l-\mathbf{x}_1)\cdot\mathbf{x}_2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & (\mathbf{x}_2-\mathbf{x}_1)\cdot\mathbf{x}_l & \dots & (\mathbf{x}_l-\mathbf{x}_1)\cdot\mathbf{x}_l \end{bmatrix}$$

Then this equation has a unique solution if the determinant of $M_{S'}$ is non-zero.

Given the current simplex defined by points $S$ we do not know *a priori* whether the closest point is to an edge (say) of the simplex or to some other feature. However by looking at the cofactors of $M_S$ GJK systematically finds the smallest $S' \subseteq S$ which contains the solution. Rephrasing [6, eqn. 30] slightly, we obtain the recursion

$$\begin{aligned} \Delta_{\mathbf{x}}(\{\mathbf{x}\}) &= 1 \\ \Delta_{\mathbf{x}}(Y+\mathbf{x}) &= \sum_{\mathbf{y}\in Y} \Delta_{\mathbf{y}}(Y) \times (\mathbf{y}\cdot\mathbf{z}_Y - \mathbf{y}\cdot\mathbf{x}) \end{aligned}$$

where '+' denotes the addition of a point to a set which did not previously contain it, and $\mathbf{z}_Y$ is an arbitrary (but fixed) element of $Y$. A simple technique gives us an easy way of ordering these terms[2]: from an arbitrary ordering of the simplex vertices $\mathbf{x}_i$ we index the subsets by saying that the subset numbered $m$ contains $\mathbf{x}_i$ if and only if the $i$'th bit in the binary representation of $m$ is a 1. We also set $\mathbf{z}_Y$ to be the $\mathbf{x}_i$ of smallest $i$ in $Y$. Then by examining our table of $\Delta_{\mathbf{x}}(S')$ values the subset yielding the shortest distance can be found by inspection—it is the subset $S'$ for which $\mathbf{x}\in S' \Leftrightarrow \Delta_{\mathbf{x}}(S'\cup\{\mathbf{x}\}) > 0$. The only way that this test can fail to identify $S'$ is if $\Delta_{\mathbf{x}}(S) \leq 0$ for each $\mathbf{x}\in S$, which means that the origin is within the simplex and we have interference between the polyhedra.

[2]If $S$ contains $n$ points there are $n2^{n-1}$ values in total requiring $n(n-1)(2^{n-2}-1)$ non-trivial multiplies and $\frac{1}{2}n(n+1)$ dot-products,

The corresponding $\lambda_i$'s can be obtained with just another $l$ divisions, and if we want to obtain explicitly the witness points for each polyhedra then as each $\mathbf{x} \in S'$ can be expressed as a vector difference of some $\mathbf{x}_P \in P$ and a $\mathbf{x}_Q \in Q$ so the individual witness points are $\sum \lambda_{\mathbf{x}} \mathbf{x}_P$ and $\sum \lambda_{\mathbf{x}} \mathbf{x}_Q$.

### 3.4 `refine_simplex`

Ignoring for now the interference case (§5), when we compute the nearest point to the origin on the simplex in TC-space then that must be to either a point, an edge or a face. Thus this point can be expressed as a linear combination of 1–3 TCSO vertices. Furthermore, if `best_simplex` returned false then the support vertices that we found from $P$ and $Q$ define a new vertex of the TCSO. So `refine_simplex` simply adds this vertex to the existing set defining the simplex to obtain a new simplex defined by 2–4 vertices, ready for the next iteration of the outer loop.

## 4 Results

The theoretical analysis [4] of the algorithm suggests that we would expect the routine to take roughly 93–135 arithmetic operations (i.e., multiplications and divisions) for each computation when tracking. An early version of this routine has been running for some time as part of our OxSim path-planning framework [11], but this only gave us anecdotal evidence of its effectiveness. We have now encoded the enhanced version of GJK in ANSI C, together with a test-harness that allows us to gather statistics on its operation over many (i.e., thousands or millions) runs, and to independently check the answers returned [3].

The objects tested were either boxes, of various or aligned orientations, or randomly generated convex polyhedra. Each polyhedra was generated about the origin, and then one of each pair given a randomly-generated translation. Sequences of tests were then generated by applying a translation and/or a random rotation to one of each pair of objects; a typical test run would consider 100 pairs of objects, each tested using 10 random motions. The distributions for the random shapes, starting positions, and incremental motions were varied to check for sensitivity of the results to these parameters. The random convex hulls were generated by generating random point sets, and then extracting the topological properties of their hulls using QHULL (from the Geometry Center at the University of Minnesota). In order to take account of the relatively coarse nature of the computer's clock the distance routine was timed over a large number ($10^3$–$10^4$) of identical calculations.

[3]The code is available via `http://www.comlab.ox.ac.uk/~cameron/distances.html`.

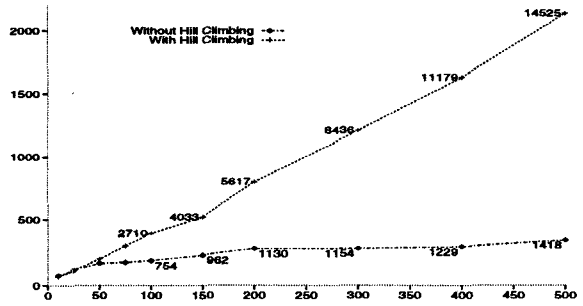

Figure 3: Plots of average times (vertical axis, in $\mu$s) against number of hull points for larger hulls without tracking, with and without hill-climbing.

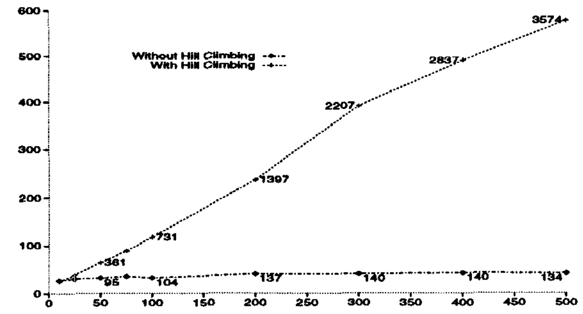

Figure 4: Plots of average times (vertical axis, in $\mu$s) against number of hull points for larger hulls with tracking, sequence length of 20.

Fig. 3 shows average timings for runs with large numbers of vertices per convex hull and without the use of tracking information. The data-points plotted are times (in $\mu$s) for a complete distance calculation. All of the timings in this section were done on a 120MHz Pentium PC running Linux, and the code was compiled using the gcc compiler with optimisation turned on. This figure illustrates clearly the usefulness of hill-climbing, with the effect being most noticeable for larger hulls. The number labels on some of the points refer to the average number of arithmetic operations, and they demonstrate that the times are almost linearly related to the number of operations. We shall return to this point in later discussion.

Fig. 4 shows what happens when we switch on the tracking mechanism. The sequence length here is 20; that is, for each pair of objects we compute the distance once from scratch, and then use the tracking information for another 19 instances. (Any sequence of length greater than about 5 shows a similar reduction in time, as the cost of the first distance calculation is amortised.) Without hill-climbing we see an almost linear function, as the time is dominated by that taken to compute the support function. This takes $6n$ operations per iteration, for $n$-point hulls, demonstrating

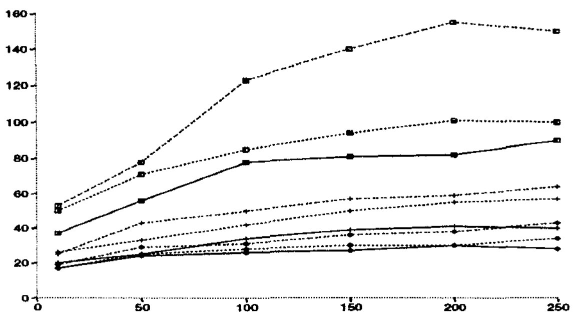

Figure 5: Plots of average times (vertical axis, in $\mu$s) against number of hull points for different types of motion.

that the average number of iterations per calculation is indeed quite close to 1. The other dramatic effect is the time taken with hill-climbing; not only is it just a fraction of the time taken without hill-climbing, it is very close to being a constant, varying between 32–42$\mu$s for this set of (randomised) motion parameters and hull sizes between 10 and 500.

The reason why we see bounded time for our enhanced GJK is that we are essentially testing whether the TC-witness point is moving from one feature of the TCSO or not, and most of the time it will not move. However this depends, of course, on how quickly the TC-witness point does moves. Another thing that can cause a movement to another feature is if the original feature changes, and this will happen if the objects rotate relative to one another. For the experiments shown in Fig. 4 each sequence used only a constant translational motion. Fig. 5 shows the effect of trying different parameters for generating the random motions, using different combinations of average distance, velocity, and rotational velocity. Nine traces are shown, showing different combinations of three choices of no/low/high rotation changes (solid/dotted/dashed lines) and different average distances between the objects. In these results we used incremental rotations chosen randomly to shift up to 0°/6°/12°. Many more data-sets were generated; we have only shown a few that illustrate the trends for the sake of clarity.

The circled points show the objects at a reasonable average distance apart, of about 3 object diameters using roughly ball-shaped objects. As the objects are moved to roughly 1 object diameter apart (crosses) the times increase slightly, but still levelling off at below 50$\mu$s on this processor. An increase like this is to be expected, as the relative shift of the TC-witness point will decrease with distance apart. Finally, as the objects actually interfere much of the time (boxes), then the usefulness of the tracking information drops off, and the times start to increase towards what we

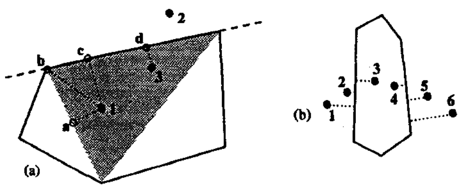

Figure 6: Estimating *MTD* for penetrations.

would see without tracking (i.e., roughly twice the times taken here).

Manufactured objects show much more regularity than the randomly-generated objects used here. With boxes we find that the computation times are even lower, at 10–20$\mu$s for the lower two triples of traces, and 20–30$\mu$s for the upper triple.

## 5 Estimating Penetration Distance

When the objects interpenetrate the origin of TC-space slips into the TCSO, and GJK discovers a simplex (almost certainly a tetrahedron) containing the origin and within the TCSO. We can ensure that all of the vertices of the simplex found by GJK are surface points of the TCSO: when first added to the simplex vertex set we can do this by always generating them by opposing support vertices, and at the next time step we can check the TC-space vertices that have remained in the simplex set by hill-climbing until we do find extremal vertices. Given the vertex set $S = \{\mathbf{x}_i\}$ for the simplex $S^\bullet$ then we must have $MTD^+(\mathbf{O}, \mathrm{TCSO}) \le \min_i |\mathbf{x}_i|$. Furthermore, the simplex solver routine (§3.3) can compute the distance to all of the four bounding faces of $S^\bullet$ to compute $MTD^+(\mathbf{O}, S^\bullet)$, and as $\mathbf{O} \subseteq S^\bullet \subseteq \mathrm{TCSO}$ we have

$$MTD^+(\mathbf{O}, S^\bullet) \le MTD^+(\mathbf{O}, \mathrm{TCSO}) \le \min_i |\mathbf{x}_i|.$$

This process in illustrated in two dimensions in Fig. 6(a), which shows a TCSO, a triangular simplex $S^\bullet$ within it (shaded), and some other detail which is introduced below. If the origin is at point 1, then the test suggested finds points **a** and **b**, and correctly determines that $|\mathbf{a}| \le MTD^+ \le |\mathbf{b}|$. The actual value of $MTD^+$ is $|\mathbf{c}|$.

In fact we expect to be able to do better than this in many applications. If we track small changes in relative position we might see the origin outside of the TCSO at point 2 in one cycle, and then at nearby point 3 within the TCSO at the next cycle. (If the objects are rotating the shape of the TCSO may change slightly, but that does not affect this argument if the incremental rotations are small.) In that case the TC-witness point for $S^\bullet$ is point **d**, which is actually the TC-witness point for the whole of the TCSO. The test for whether we have such a case is straightforward, involving in the worst case two calls to the support vertex routine which may not be needed if the routine 'remembers' to check whether this surface feature from the last call to GJK is still a surface feature at this call.

For deep penetrations such a simple analysis may not give the true answer. The problem is illustrated by Fig. 6(b), which shows a sequence of origin points passing through a TCSO, and in which the TC-witness point will suddenly move from the left-hand edge to the right-hand edge (points 3 and 4). The problem is not convex and therefore naïve hill climbing will not work: [3] solved this problem using a search over all of the TCSO faces, and [10] solve a related problem using an approximation to the Voronoi diagram.

## 6 Conclusions

We have shown experimentally that our enhanced version of GJK takes a time that is almost constant for the tracking problem when there are small changes in relative configurations. Further, the number of arithmetic operations required is very close to that predicted by the theory, namely around 100 multiplications or divisions for each step of simulated time.

It should also be remembered that the detailed analysis of the inner loops dealt only in terms of the number of multiplications and divisions. In fact both algorithms do significant amounts of index manipulation and real-number addition and subtraction, and on modern hardware the cost of multiplication and vector operations do not dominate timings as much as they did once. This can be seen from the results shown by looking at the time taken by the algorithm divided by the number of arithmetic operations that we recorded. For the algorithm without hill-climbing that comes out to be around 160ns, whereas the processor used is rated at about 11 MFlop which would indicate an expected time of nearer 100ns. Some of this discrepancy will be due to the cost of the additional machine operations, and on a modern small computer some of the time will be due to cache misses and pipeline flushes. These latter effects probably account for the increase in average time per operation for the hill-climbing version to around 250–300ns; the difference in the code for these two methods is tiny.

Finally, note that we have assumed here that the coordinates of the object vertices are available on demand. Computing these coordinates for *all* of the vertices would take many more operations than a call to a single instance of our version of GJK, but on modern workstations these numbers are computed within the graphics hardware. It would clearly be possible to implement a distance routine such as described here in hardware to provide distance calculations very quickly. There is a catch though: whereas in visualisation we usually view from single directions, in simulation we are likely to want to keep track of distances between many pairs of objects [10]. Organising this information and returned distances, and informing the hardware what object pairs do need tracking, are significant hurdles.

## Acknowledgements

Thanks are due to Ong Chong Jin and Elmer Gilbert for discussions on the nature of GJK, and to the EPSRC for financial support.

## References

[1] Boris Baginski. Local motion planning for manipulators based on shrinking and growing geometry models. In *ICRA*, pages 3303–3308, Minneapolis, April 1996.

[2] C. E. Buckley and L. J. Leifer. A proximity metric for continuum path planning. In *9th Int. Conf. Artificial Intelligence*, pages 1096–1102, August 1985.

[3] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. In *Int. Conf. Robotics & Automation*, pages 591–596, San Francisco, April 1986.

[4] Stephen Cameron. A comparison of two fast algorithms for computing the distance between convex polyhedra. Conditionally accepted, IEEE TR&A, July 1996.

[5] E. G. Gilbert and C-K Foo. Computing the distance between general convex objects in three-dimensional space. *IEEE Trans. Robotics & Automation*, 6(1):53–61, February 1990.

[6] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Trans. Robotics & Automation*, 4(2):193–203, April 1988.

[7] E. G. Gilbert and C. J. Ong. New distances for the separation and penetration of objects. In *Int. Conf. Robotics & Automation*, pages 579–586, San Diego, May 1994.

[8] Ming Lin and John Canny. A fast algorithm for incremental distance calculation. In *Int. Conf. Robotics & Automation*, pages 1008–1014, Sacremento, April 1991.

[9] T. Lozano-Pérez. Spatial planning—a configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, February 1983.

[10] M. Ponamgi, D. Manocha, and M. Lin. Incremental algorithms for collision detection between solid models. In *Proc. ACM/SIGGRAPH Symposium on Solid Modeling*, pages 293–304, 1995.

[11] Caigong Qin, Stephen Cameron, and Alistair McLean. Towards efficient motion planning for manipulators with complex geometry. In *Proc. IEEE Int. Symp. Assembly and Task Planning*, pages 207–212, August 1995.

[12] Y. Sato, M. Hirata, T. Maruyama, and Y. Arita. Efficient collision detection using fast distance-calculation algorithms for convex and non-convex objects. In *ICRA*, pages 771–778, Minneapolis, April 1996.