The Quickhull Algorithm for Convex Hull

C. Bradford Barber* David P. Dobkin†
Hannu Huhdanpaa‡

Geometry Center Technical Report GCG53

July 30, 1993

Abstract

The convex hull of a set of points is the smallest convex set that contains the points. This paper presents a convex hull algorithm that combines the 2-d QUICKHULL algorithm with the general dimension Beneath-beyond algorithm. It is similar to the randomized, incremental algorithms for convex hull and Delaunay triangulation. Our algorithm is simpler, uses less memory, and allows good use of virtual memory. A simple modification produces an approximate convex hull. We provide empirical evidence that the algorithm runs faster and is output sensitive.

1. Introduction

The convex hull of a set of points is the smallest convex set that contains the points. The convex hull is a fundamental construction for mathematics and computational geometry. Other problems can be reduced to convex hull, e.g., Delaunay triangulation, half space intersection and linear programming.

For example, Boardman analyzes imaging spectrometry data from AVIRIS (a remote sensor attached to a U-2 plane). His goal is identifying and mapping the pure materials in the scene. Each pixel, 20 meters square, has a 224-band spectrum. The observed spectrum can be modeled as a simple linear combination of some set of unknown pure endmember spectra, each weighted by its fractional abundance. Each pixel represents multiple materials and adjacent pixels have

*The Geometry Center, University of Minnesota, Minneapolis, MN 55454, barber@geom.umn.edu. This research was supported in part by the National Science Foundation under Grant Numbers CCR90-02352 and NSF/DMS-8920161.

†Department of Computer Science, Princeton University, Princeton, NJ 08544, dpd@princeton.edu

‡The Geometry Center, University of Minnesota, Minneapolis, MN 55454, hannu@geom.umn.edu

related spectra. The first step makes the noise in the data isometric and homogeneous by determining the noise covariance of adjacent pixels. The second step determines the principal components of the data and the coordinates for each pixel in this space. Boardman can then determine the endmember spectra from the vertices of a simplex that fits the reduced data. Each facet of the simplex corresponds to a set of nearly coplanar facets of the data's convex hull. Although the method has some inherent weaknesses, initial results have been promising [4].

Another example is the canonical triangulation of a cusped hyperbolic 3-manifold. A cusped manifold is a manifold that extends to infinity under some metric; examples include most knot complements and a torus with a missing, surface point. Given a cusped hyperbolic 3-manifold, Weeks identifies points on a light cone in 4-space with cusps of the lifts of the manifold in its universal covering space. The canonical triangulation corresponds to the convex hull of a sufficiently complete set of these points [26].

Other applications for convex hull arise from the Delaunay triangulation and its relatives, Voronoi diagrams and power diagrams. In his review article, Aurenhammer describes applications in mesh generation, file searching, cluster analysis, collision detection, crystallography, metallurgy, urban planning, cartography, image processing, numerical integration, statistics, sphere packing, and point location [1].

Recent work on convex hulls and Delaunay triangulations has focused on a randomized, incremental algorithm that has optimal, expected performance [9] [10] [14] [19] [20]. The algorithm constructs the convex hull or Delaunay triangulation by adding points in a random order. After each step, the algorithm produces the convex hull or Delaunay triangulation of the previously processed points.

In order to add a point to a convex hull, the randomized algorithms identify the facets below the point. These are the *visible facets* for the point and their boundary is the point's *horizon*. If the point is below all facets, the point is inside the convex hull and can be discarded. Otherwise the visible facets are replaced by a cone of new facets. Each new facet is defined by the point and one horizon facet. The incremental Delaunay triangulation algorithms work in a similar fashion once the problem is transformed to the equivalent convex hull problem [6].

For convex hull algorithms of this type, the inner loop consists of finding the facets visible from a point. Naively, this is done by exhaustively testing existing facets. The randomized incremental algorithms retain previously constructed hulls to perform this search in expected logarithmic time (in 3-d).

The randomized algorithms are simple enough to implement. In practice, their running times are competitive with existing algorithms [16]. However, we believe that simpler algorithms are possible which run even faster and use less space. Our search for such an algorithm is driven also by robustness issues [2]. A difficulty in simplifying the algorithm is that performance analysis becomes more difficult. We present empirical evidence to justify our claims.

Our algorithm is the same as the randomized incremental algorithms once we select a point and a visible facet. The differences occur in locating visible facets and in selecting points to process. While the randomized algorithms store the previously constructed hulls, we store an *outside set* of unprocessed points for each facet. A facet is visible from all points in its outside set. Furthermore, our algorithm processes the furthest point of an outside set instead of a random point. We call our algorithm Quickhull because in 2-d, it is the same as the QUICKHULL algorithm [7] [13] [15] [17].

For a given point, the cone of new facets and the sequence of distance tests can be the same for Quickhull and the randomized algorithms. The difference is how we interleave the distance tests. Quickhull tests all points for a facet when the facet is created, while the randomized algorithms test all facets for a point when the point is processed.

The expected performance of Quickhull is an open problem. We define two balance conditions that hold empirically. When they do hold, the performance of Quickhull for n input points and r output vertices is $O(n \log r)$ for $d <= 3$ and $O(n f_r / r + f_r)$ for $d >= 4$ (f_r is the maximum hull size for r vertices). This is the same as for the randomized algorithms when $r = n$.

When the balance conditions hold, the performance of Quickhull is output sensitive. Output-sensitivity is important for convex hull algorithms because the output size can be much smaller than the worst case size. In 2-d, Kirkpatrick and Seidel found an optimal, output-sensitive algorithm for convex hull that runs in $O(n \log h)$ where h is the output size [22]. Clarkson & Shor give a 3-d convex hull algorithm with optimal, output-sensitive, expected time [11]; it was derandomized by Chazelle and Matoušek [9]. In higher dimensions, the best output-sensitive algorithm is gift wrapping at $O(nh)$ [8].

Quickhull uses less space than the randomized algorithms because it stores outside sets instead of intermediate hulls. Furthermore, Quickhull can process a group of neighboring facets together. This would reduce the paging overhead that Fortune noticed [16].

Quickhull produces an approximate convex hull when it ignores points near a facet. The complexity does not change. In 2-d, Li and Milenkovic produce an approximate convex hull in $O(n \log n)$ time [24], while Bentley et al produce an approximate convex hull in $O(n + k)$ time where k is the number of "strips" [3]. While the later algorithm generalizes to higher dimensions, its efficiency rapidly disintegrates.

We have also used Quickhull for constructing approximate convex hulls with floating point arithmetic or imprecise data [2]. Selecting a furthest point is important for bounding the maximum error.

2. Convex hulls and triangulations

This section is a quick review of hyperplanes, convex hulls, and triangulations. Given normal vector v and an offset a, a point p is *above* the corresponding hyperplane if $\langle v, p \rangle > a$. A *convex combination* of a set of points is a linear combination with positive coefficients and unit sum. The *convex hull* of a set of points S, $conv(S)$, is the smallest set that is closed under convex combinations. An *extreme point* of a point set is a vertex of the convex hull. The maximum size, f_r, of a convex hull of d dimensions and r vertices is $f_r = O(n^{\lfloor d/2 \rfloor}/\lfloor d/2 \rfloor!)$ [23].

An *i-simplex* is the convex hull of $i+1$ independent points. An *j-face* of a i-simplex is the convex hull of $j+1$ of its points. An i-simplex is also the convex hull of an $(i-1)$-simplex (its *base*) and a 0-simplex (its *apex*).

A *simplicial d-complex* is a set of i-simplices $(0 \leq i \leq d)$ where every simplex is a face of some d-simplex. In addition, every non-empty intersection of i-simplices of a d-complex is a common face. The *trace* of a simplicial complex is the union of its simplices.

A *triangulation* of a set of sites in R^d is a simplicial d-complex of the sites; its trace is convex. A simplex has an *empty circumsphere* if the circumsphere of its vertices contains no other sites. A *Delaunay triangulation* is a triangulation with empty circumspheres. It is equivalent to the convex hull of the sites lifted to an R^{d+1} paraboloid [6].

Processing a point in Quickhull and the randomized incremental algorithms is an implementation of the Beneath-beyond theorem by Grünbaum [Th. 5.2.1] [18]:

Theorem 2.1. (Beneath-beyond) *Let H be a convex hull in R^d, and let p be a point in $R^d - H$. Then the faces f of $conv(p \cup H)$ are:*

1. *f is also a face of H iff there is a facet F of H such that f is in F and p is below F.*

2. *f is not a face of H iff $f = conv(p \cup f')$ with $f' \in H$ and either (a) p is a linear combination of vertices of f', or (b) p is above one facet of H containing f' and below another facet containing f'.*

Kallay's beneath-beyond algorithm for convex hull is a direct translation of the theorem [21] [25]. Quickhull and the randomized incremental algorithms do not explicitly build the convex hulls of lower dimensional faces. Instead they construct new facets from horizon ridges and the processed point. For points in general position, the corresponding theorem is below. It can be extended to singular data by triangulating non-simplicial facets [2].

Theorem 2.2. (Simplified beneath-beyond) *Let H be a convex hull in R^d and let p be a point in $R^d - H$. Then the facets F of $conv(p \cup H)$ are:*

1. *F is also a facet of H iff p is below F*
2. *F is not a facet of H iff its apex is p and its base is a ridge of H with one incident facet below p and the other incident facet above p.*

Proof: Under general position, all faces are simplices. Let $G = conv(p \cup H)$. If a facet is in H (resp. G) then all of its faces are in H (resp. G). By Theorem 2.1, if p is below a facet F in H, F and all of its faces are also in G. Also from Theorem 2.1, if a ridge r has one neighboring facet above p and the other facet below p, the simplex with base r and apex p is a facet of G as are all of its faces. ∎

3. The Quickhull algorithm

Theorem 2.2 is the foundation for Quickhull and the randomized incremental algorithms. The central problem is determining the visible facets efficiently. Since visible facets are connected by ridges, locating one visible facet allows the rest to be located quickly. The randomized algorithms use the prior hulls to locate the first visible facet for a point. Our solution is simpler. After initialization, Quickhull always knows a visible facet for each point. So given a point, Quickhull can quickly determine its visible facets.

We use the following terminology in describing Quickhull. An *visible facet* for a point is a facet that is below the point. The *horizon* for a point is the boundary of its *visible facets.* The horizon consists of *horizon ridges.* A *new facet* is a facet with the point as its apex and a horizon ridge as its base. The *cone* for a point is the set of new facets.

When Quickhull creates a cone of new facets, the visible facet for each point is updated. This process is called *partitioning* because it is similar to the partition step of Quicksort. The *outside set* for a facet contains points that are above the facet. The facet is visible from each of these points. Partitioning reassigns the outside sets of replaced facets and their immediate neighbors. It rejects points of replaced facets if the point is inside the cone. Partitioning also records the furthest point of each outside set. An outline of the algorithm is in Figure 1.

To prove the correctness of Quickhull, we first prove that a point could either be partitioned into any legal outside set or the furthest outside set.

Lemma 3.3. *If an extreme point of the convex hull is above two or more facets at a partition step in Quickhull, it will be processed irrespective of which facet it is assigned to.*

create an initial hull of linearly independent points
partition the remaining points into the initial hull's outside sets
for each facet with a non-empty outside set
    select the furthest point of the set
    find the horizon and other visible facets for the point
    make a cone of new facets from the point to the horizon
    partition the outside sets into the cone

Table 1: Quickhull algorithm for convex hull in R^d.

Proof: Assume the contrary and consider an extreme point P assigned to a facet F. By assumption there must be a furthest point whose cone is above or coplanar with P and whose visible facets include F. But then P is inside the convex hull and not an extreme point. ∎

Lemma 3.4. *Let partitioning assign points to the facet they are furthest above. If a point is furthest above a new facet, it is in the outside set of a replaced facet or its immediate neighbor.*

Proof: Using induction, assume points have been assigned to the facet they are furthest above. By convexity, the bisectors of the dihedral angle for each ridge separate the outside sets from each other. After the new facets are added to the hull, the bisectors are the same except for the ridges of new facets. So output sets are the same except for new facets, replaced facets, and their immediate neighbors. Partitioning rebuilds these outside sets. ∎

Theorem 3.5. *The Quickhull algorithm produces the convex hull of a set of points in R^d. If partitioning assigns a point to the facet it is furthest above, Quickhull only processes extreme points.*

Proof: The Quickhull algorithm is a specialization of Theorem 2.2. In particular, Quickhull partitions the points into outside sets and picks furthest points for processing. After a visible facet is located, the algorithm follows Theorem 2.2 and creates a cone of new facets. By Lemma 3.3, partitioning can not prevent an extreme point from being processed. If partitioning assigns the furthest facet, then Lemma 3.4 shows that points are assigned to the facet they are furthest above. The furthest point above a facet is an extreme point. The termination conditions are the same, so the correctness of beneath-beyond proves the correctness of Quickhull. ∎

The randomized, incremental algorithms for constructing convex hulls perform the same steps as Quickhull but in a different order [10] [16] [19]. They use beneath-beyond on a random permutation of the points. They retain the old convex hulls to speed up the selection of a visible facet. They use depth-first search to find a sequence of visible facets from the initial hull to the current hull. The search is equivalent to a sequence of partitioning steps.

Edelsbrunner and Shah [14], Joe [20], and Boissonnat and Devillers-Teillaud [5] use a similar method for Delaunay triangulations. They express their algorithm in terms of triangulations and the insphere test. By the correspondence between Delaunay triangulation and convex hull, each triangle is a facet of the convex hull and the insphere test determines the visible facets for the lifted point [6].

If all of the algorithms perform essentially the same steps, why do we prefer Quickhull? The answer is simplicity, spatial efficiency, optimality, and control. Compare our outside sets with their previous hulls. While the previous hulls are simplicial complexes, an outside set is a simple list of points. The total size of the outside sets can't be greater than the input size.

Now consider a distribution with interior points. The randomized algorithms select a random point for processing. If the point is an interior point, a later iteration will delete the facets created for it. Quickhull is greedy and selects the furthest point above the facet. Depending on which Quickhull variation is used, this is either an extreme point or it is likely to be an extreme point. Furthest points are also important for producing approximate convex hulls and for bounding the effects of round-off error and imprecise data.

Finally, consider the sequence of facets processed by the algorithm. The randomized algorithms process a random sequence of facets. When the intermediate hulls become significantly larger than available memory, every iteration may need to load pages from virtual memory. Because of the exponential growth of high dimensional hulls, available memory is easily filled. Quickhull is not random. It can process facets in any sequence desired. In particular, it can process all points for a neighborhood of facets before processing the next neighborhood of facets.

4. Complexity analysis of Quickhull

The complexity analysis of Quickhull is an open problem. By Table 1, the performance of Quickhull on cospherical points is similar to the performance of the randomized incremental algorithms. This is significantly better than Quickhull's worst-case complexity which is bounded by n iterations that partition n points into the maximum number of facets for n vertices, or $O(n^2 f_n)$.

If the input includes interior points, Quickhull performs better than the randomized incremental algorithms. In Table 2, we show time and space requirements for a cospherical distribution of points nearly inscribed in a cube. The main costs for Quickhull is reading in the input points and performing a few partition steps per point. The randomized incremental algorithms may build large portions of the sphere before processing the cube's vertices. Empirically, it appears that Quickhull performs like an output-sensitive, randomized incremental algorithm.

In these tests, we assigned outside points to any visible facet. In the following discussion, we use the Quickhull variation that only processes extreme points. This reduces the points processed by Quickhull and simplifies the analysis, but it more than doubles the number of partition tests.

The following modification turns Quickhull into a randomized, incremental algorithm. Instead of selecting a furthest point to process next, Quickhull could select a random point in a random outside set. Except for the interleaving of distance tests, an execution of the randomized Quickhull would be no different than an execution of the randomized, incremental algorithms. Both algorithms would create the same facets and compare the same points with the same hyperplanes.

The actual Quickhull selects a furthest, extreme point to process next instead of a random point. In most cases, this point should be better than a random point. A bad case occurs when partitioning assigns all of the points to a single facet. This is similar to Quicksort picking the smallest or largest element for its pivot. With Quicksort, it's easy to design such a pivot rule and input set. With Quickhull, it's harder. First you need to know how Quickhull selects the initial hull. Then you need to run the algorithm backwards. If you use the version of Quickhull that assigns any outside set, you also need to know the order that Quickhull creates new facets. For example, place points along a half-loop of a parametric spiral at $t = 0$ and at $t = 2^{-i}$ for $i = 0 \ldots n-2$. Adding the j'th vertex will partition $n - j$ points. In practice though, the coordinates for these points quickly run out of significant digits.

Another bad case for Quickhull would occur if each processed point rebuilt most of the hull. With the randomized, incremental algorithms, this case occurs when points are processed, in order, along one rotation of a spiral in R^3. Quickhull does not succumb to this case since it will process the furthest point on the spiral before most of the other points. The construction of a bad case for new facets in Quickhull is an open problem.

We conjecture that the bad cases for Quickhull are rare in the same way that bad cases for Quicksort are rare. Consider a pivot rule for Quicksort that always selects the middle element. This is the optimum choice if the input is already sorted, and a good choice if the input is nearly sorted. Out of all possible inputs, only a small fraction of them would be bad. Quickhull's use of furthest, extreme points is a similar selection rule.

Let d be the dimension, n be the number of input points, r be the number of output vertices, and f_r be the maximum number of facets of r vertices. We will assume that all interior points are just inside the convex hull. There are two main costs to Quickhull, creating new facets and partitioning. Each new facet takes $O(d^3)$ work to create a hyperplane and each partition takes $O(d)$ work to compute a distance. We conjecture that each Quickhull iteration is an average case, i.e., each iteration creates an average number of new facets whose outside sets are average size. With j vertices, the maximum number of facets is f_j. There are d vertices per facet, so the average number of facets per vertex is df_j/j and the average number of outside points per facet is $(n-j)/f_j$. The average, total number of partitioned points for an iteration is then $d(n-j)/j$. We call these averages, the *balance conditions* for Quickhull.

Definition 4.6. *An execution of Quickhull is* balanced *if*

- *the expected number of new facets for the j'th vertex is* $O(df_j/j)$
- *the expected number of partitioned points for the j'th vertex is* $O(d(n-j)/j)$

Tables 3 and 4 give histograms for the differences between actual and expected values for cospherical points in R^3 and R^5. Note that the expected, total number of new facets, $O(\sum_{j=1}^{r} df_j/j)$, is the same as the expected, total number of created facets in Clarkson, et al.'s analysis of the randomized, incremental algorithms [10].

Theorem 4.7. *If an execution of Quickhull is balanced, its expected complexity is* $O(n \log r)$ *for* $d <= 3$ *and* $O(nf_r/r + f_r)$ *for* $d >= 4$.

Proof: There are two costs to Quickhull, adding a point to the hull and partitioning. The first is proportional to the total number of new facets created, i.e., $O(d^3 \sum_{j=1}^{r} df_j/j)$. This simplifies to $O(f_r)$ (each term is less than df_r/r and the $\lfloor d/2 \rfloor!$ denominator of f_r subsumes d^4).

The cost of partitioning a point for an iteration is proportional to the number of new facets. The total cost is $O(d \sum_{j=1}^{r} d^2(n-j)f_j/j^2)$. Expanding f_j yields $O(d^3 n \sum_{j=1}^{r} j^{\lfloor d/2 \rfloor - 2} / \lfloor d/2 \rfloor !)$. If $d <= 3$, the sum is $O(n \log r)$. Otherwise, each term is less than f_r/r^2 and the sum is $O(nf_r/r)$. ∎

If $r = n$ and the balance conditions hold, the expected cost of Quickhull is $O(n \log n)$ for $d <= 3$ and $O(f_n)$ otherwise. This is the same as the expected cost of the randomized, incremental algorithms [10].

5. Discussion

Our goal is a practical algorithm for general dimension convex hulls. Quickhull processes the furthest or locally furthest point instead of a random point. This eliminates or reduces non-extreme vertices and makes Quickhull output-sensitive. We have shown empirical evidence that the algorithm satisfies its balance conditions and performs like an output-sensitive, randomized incremental algorithm.

Quickhull stores the outside set for each facet instead of storing the previously constructed facets. This reduces the space requirements for Quickhull and allows Quickhull to build approximate hulls. Quickhull selects a facet for processing instead of selecting a random facet. This allows Quickhull to use virtual memory appropriately and to develop specified sections of the hull.

For higher dimensions, space is the primary resource limitation for convex hull algorithms. For example, the convex hull of 300 cospherical points in R^6 produces 30,310 facets, yet Boardman wants to determine the convex hull of 10,000 points in R^6. Table 5 shows the rapid growth in complexity as dimension increases. One solution is to generate an approximate convex hull. Table 6 compares Quickhull on a cubical distribution of points in R^6, with and without approximation.

In higher dimensions, even approximate convex hulls will produce too many facets. Also, neither Quickhull nor the randomized incremental algorithms always work with floating point arithmetic. To solve these problems, we can merge non-convex facets at each iteration [2].

References

[1] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[2] C. B. Barber. *Computational Geometry with Imprecise Data and Arithmetic*. PhD thesis, Princeton University, 1992. Tech Report CS-TR-377-92.

[3] J. L. Bentley, M. G. Faust, and F. P. Preparata. Approximation algorithms for convex hulls. *Communications of the ACM*, pages 64–68, 1982.

[4] J.W. Boardman. Automating spectral unmixing of aviris data using convex geometry concepts. Fourth Airborne Geoscience Workshop, Washington, D.C., October 1993.

[5] J.-D. Boissonnat and M. Devillers-Teillaud. On the randomized construction of the delaunay tree. *Theoretical Computer Science*, Dec 1989. to appear. Available as Tech. Report INRIA 1140, Dec 1989.

[6] D.F. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9:223–228, 1979.

[7] A. Bykat. Convex hull of a finite set of points in two dimensions. *Information Processing Letters*, 7:296–298, 1978.

[8] D.R. Chand and S.S. Kapur. An algorithm for convex polytopes. *Journal of the ACM*, 7:78–86, 1970.

[9] B. Chazelle and J. Matoušek. Derandomizing an output-sensitive convex hull algorithm in three dimensions. *Computational Geometry: Theory and Applications*, 1991.

[10] K.L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *Symposium on Theoretical Aspects of Computer Science*, 1992. to appear.

[11] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, ii. *Discrete Computational Geometry*, 4:387–421, 1989.

[12] S. Dorward. Personal communication. AT&T Bell Labs, 1992.

[13] W. Eddy. A new convex hull algorithm for planar sets. *ACM Transactions on Mathematical Software*, 1977.

[14] H. Edelsbrunner and N.R. Shah. Incremental topological flipping works for regular triangulations. In *Proceedings of the Symposium on Computational Geometry*, pages 43–52. ACM, 1992.

[15] R.W. Floyd. Private communication to Preparata & Shamos on Quickhull, 1976.

[16] S. Fortune. Computational geometry. In R. Martin, editor, *Directions in Geometric Computing*. Information Geometers, 1993.

[17] P.J. Green and B.W. Silverman. Constructing the convex hull of a set of points in the plane. *Computer Journal*, 22(262-266), 1979.

[18] B. Grünbaum. Measures of symmetry for convex sets. In *Proceedings of the Seventh Symposium in Pure Mathematics of the American Mathematical Society, Symposium on Convexity*, pages 233–270, 1961.

[19] L. Guibas, D.E. Knuth, and M. Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, pages 381–413, 1992.

[20] B. Joe. Construction of three-dimensional delaunay triangulations using local transformations. *Computer-Aided Geometric Design*, 8:123–142, 1991.

[21] M. Kallay. Convex hull algorithms in higher dimensions. Unpublished manuscript, Dept. Mathematics, Univ. of Oklahoma, Norman, Oklahoma. See also Preparata & Shamos 1985, 1981.

[22] D.G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Computing*, 15:287–299, 1986.

[23] V. Klee. Convex polytopes and linear programming. In *Proc. IBM Sci. Comput. Symp.: Combinatorial Problems*, pages 123–158, 1966.

[24] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *Proceedings of the Symposium on Computational Geometry*, pages 197–207. ACM, 1990.

[25] F.P. Preparata and M.I. Shamos. *Computational Geometry. An Introduction.* Springer-Verlag, 1985.

[26] J.R. Weeks. Convex hulls and isometries of cusped hyperbolic 3-manifolds. Technical Report TR GCG32, The Geometry Center, Univ. of Minnesota, August 1991.

6. Appendix A. Structure of the code

The Quickhull program takes a set of points and produces the smallest convex set that contains the points. The program has options for Delaunay triangulation, approximate convex hull, partitioning, and output formats. Currently, three output options are available, a summary, an OOGL-format for a 3-d or 4-d viewer (Geomview), and lists of vertices for each facet.

Quickhull represents the top and the bottom levels of the facial graph as a structure of facets and vertices. The underlying data structure is the set. A set consists of two fields, the maximum number of elements and an array of element pointers. Sets are NULL terminated.

Each facet has a unique identifier. A facet has a normal and an offset, the distance to the furthest point from the facet, pointers to the previous and next facets in the doubly linked facet list, an outside set of points, an oriented set of vertices for the facet, and the set of neighboring facets. Simplicial ridges are represented implicitly by intersecting the vertex sets of neighboring facets. For now, all ridges and facets are simplicial.

Each vertex has a unique identifier, and a pointer to its coordinates. All the structures described above represent their objects in general dimension.

The program is divided into seven modules. They are qhull.c, poly.c, geom.c, mem.c, set.c, io.c, and globals.c

The qhull.c module contains the top level routines to handle arguments, to initialize calculation, to build a hull incrementally by adding points one at a time to the simplex, to find the horizon and interior for a point, and to partition points into outside sets of facets.

initialhull() takes $dim + 1$ vertices as an argument, and returns the initial hull as a doubly linked list of facets. Each facet is given an orientation, either top or bottom.

partitionall() partitions all points into the outside sets of facets. There are three possible options in partitioning, and the user specifies which one is used. A point can be assigned to the outside set of any facet it is above, or it can be assigned to the outside set of the facet it is furthest above. The third option is to also partition horizon facets. The later two options guarantee that processed points are extreme points.

buildhull() constructs a hull incrementally by adding points, one at a time, to the simplex. It goes down the facet list, taking the furthest point of the outside set, calls findhorizon() to determine the horizon facets and interior facets for the selected point, and calls makecone() to construct new facets specified by the selected point and the horizon ridges. A normal is calculated for each new facet, and outside points of the interior facets are repartitioned into the outside sets of new facets.

findhorizon() determines which facets are visible from the selected points. These facets are called interior facets. findhorizon() also determines a set of facets that have interior facets as neighbors. These facets are called horizon facets.

The poly.c module has routines to implement polyhedra and simplices, to check the correctness of the result, to intersect the vertex sets of two facets in order to create the implicit representation of a ridge, and to construct a cone of new facets.

createfacet() creates a facet from an oriented set of vertices. It creates a neighbor set for the facet by calling updateneighbors() repeatedly with different skip indices. The skip index specifies which of the facet's vertices is skipped to get the vertices of a ridge between the facet and one of its neighbors. updateneighbors() maintains a hash table that is used to find the neighbor sharing that ridge.

createnewfacet() is called by makecone() to create the facets consisting of the apex vertex and the vertex set of one horizon ridge. This procedure is equal to createfacet(), except that skip index 0 is not used in updating the neighbor set. This is because the first vertex is the apex vertex, and it is already known that removing this vertex gives the horizon ridge.

createsimplex() creates the initial simplex from a set of vertices. The set has $dim+1$ vertices, and removing one vertex at a time gives all possible sets of facet vertices. The orientation alternates.

makecone() function constructs a cone of new facets from the selected point to a set of horizon ridges. The horizon ridges are derived from the intersections of horizon and interior facets.

The geom.c module contains the geometric routines of the application, such as calculating distances from points to facets and calculating facet normals. sethyperplane() determines the normal of a facet from its vertices. The facet's orientation sets the normal's orientation. In 2-d and 3-d, the normal is calculated by a cross-product. In higher dimensions, sethyperplane() uses Gaussian elimination with partial pivoting, and if this produces a degenerate pivot, Gram-Schmidt orthogonalization. If this also fails, the program generates an error. A future release may use singular value decomposition to detect rank deficiencies, and minimum spanning trees to reduce numeric error.

The mem.c module contains the memory management routines. For efficiency reasons, malloc() and free() are not used for small allocations. These allocations are done from a list of free blocks or from a preallocated buffer.

The set.c module contains the low level set manipulation routines, such as adding, appending, and prepending elements into a set, deleting elements, testing for equality between two sets, creating, and destroying sets. These routines are optimized for speed since they occur in the inner loop, especially for high dimensions.

The io.c module contains input and output functions of Quickhull application. The globals.c module declares global variables used by Quickhull.

7. Tables

We compared Quickhull (qhull) with Dorward's implementation (hullio) of Clarkson et al's randomized, incremental convex hull algorithm [10] [12] [16]. We used four distributions:

- random cospherical points (generated uniformly in a cube and projected to a sphere).
- nearly inscribed, cospherical points. The cospherical points are random points projected to a sphere. We inscribed the sphere in a cube and uniformly shrunk the cube to place at least one cospherical point above each cube facet.
- random points in the unit cube, $[-1, +1]^d$.

The timing runs for Tables 1, 2, and 5 are from a Sun SPARCstation1+ with 15 Mbytes of memory. The timing run in Table 6 is from an Iris 4D/30 with 40 Mbytes of memory. Each hullio run was repeated three times to average the effects of randomization. In Tables 3 and 4, we computed the expected number of new facets from the actual number of total facets. In Table 6, we stopped processing a facet when its furthest point was at most 0.1 from the facet. We did not repartitioned horizon facets (the 'f' option). For the 1,000 points case, this misses 1 point that is 0.1035 from a facet.