A comparison of five implementations of 3d Delaunay tessellation

Yuanxin Liu Jack Snoeyink
Department of Computer Science
University of North Carolina at Chapel Hill

Abstract

When implementing Delaunay tessellation in 3d, a number of engineering decisions must be made about update and location algorithms, arithmetics, perturbations, and representations. We compare five codes for computing 3D Delaunay tessellation: qhull, hull, CGAL, pyramid, and our own tess3, and explore how these decisions affect the correctness and speed of computation.

1 Introduction

The Delaunay tetrahedralization is a useful canonical decomposition of the space around a given set of points in a Eucliean space E^3, frequently used for surface reconstruction, molecular modelling and tessellating solid shapes [13, 8, 26]. The Delaunay is often used to compute its dual Voronoi diagram, which captures proximity. In its turn, it is often computed as a convex hull of points lifted to the paraboloid of revolution in one dimension higher [10, 11]. As we sketch in this paper, there are a number of engineering decisions that must be made by implementors, including the type of arithmetic, degeneracy handling, data structure representation, and low-level algorithms.

We wanted to know what algorithm would be fastest for a particular application: computing the Delaunay tessellation of points that represent atoms coordinates in proteins, as represented in the PDB (Protein Data Bank) format [1]. Atoms in proteins are well-packed, so points from PDB files tend to be evenly distributed, with physically-enforced minimum separation distances. Coordinates in PDB files have a limit on precision: because they have an 8.3f field specification in units of Ångstroms, they may have three decimal digits before the decimal place (four if the number is positive), and three digits after. Thus, positions need about 20 bits, with differences between neighboring atoms needing 12 bits. Since the the experimental techniques do not give accuracies of thousandths or even hundredths of Ångstroms, we may even reduce these limits.

We therefore decided to see whether we could stretch the use of standard IEEE 754 double precision floating point arithmetic [2] to perform Delaunay computation for this special case. We implemented a program, tess3 [23], which we sketch, and compared it with four popular codes available in the public domain: Barber et al.'s Qhull [6], the CGAL geometry library's Delaunay Hierarchy [14], Shewchuk's Pyramid [29] and Clarkson's Hull [12]. Our program, designed to handle limited precision, uniformly-spaced input using only double precision point arithmetic, was fastest on both randomly generated input and points from PDB files, although it did compute incorrect tetrahedra for one of the 20,393 PDB files that did not satisfy the input assumptions.

The performance of Delaunay code is affected by a number of algorithmic and implementation choices. We compare these choices made by all five programs in an attempt to better understand what makes a Delaunay program work well in practice. In Section 2, we review the problem of computing the Delaunay tetrahedralization and describe the main algorithmic approaches and implementation issues. In Section 3, we compare the programs for computing the Delaunay. In section 4, we show experiments that compare all five programs in speed, and some experiments that look at the performance of tess3 in detail.

There are many other programs that can compute Delaunay Tedrahedralization: Edelsbrunner's deltri [19] and Watson's nnsort [30, 31], to cite just two. The candidate programs were selected for their popularity and their speed. In particular, all of these programs are able to rely mostly on floating point operations when given coordinates that have small number of bits, and can handle at least hundreds of thousands of points when running on a desktop computer.

2 Delaunay Tetrahedralization

There are several common element in the five programs that we survey.

2.1 Definition

The *Delaunay diagram* in E^3 can be defined for a finite set of point *sites* P: For every sphere S whose interior contains no sites of P, the interior of the convex hull of $S \cap P$ is in the Delaunay diagram. The Delaunay diagram is dual to the *Voronoi diagram* of P, which is defined as the partition of E^3 into maximally-connected regions that have the same set of closest sites of P. The Delaunay diagram completely partitions the convex hull of P.

If the sites are in *general position*, in the sense that no more than four points are co-spherical and no more than three are co-planar on the convex hull, then Delaunay diagram is a *Delaunay tetrahedralization*—each of the empty spheres defines a hull that is a tetrahedron, triangle, edge, or single point. The programs we survey have different approaches to enforce or simulate general position, so that only tetrahedra need be represented.

2.2 Representation

A Delaunay tetrahedralization, or any simplicial complex, can be represented its full facial lattice: its vertices, edges, triangles and tetrahedra and their incident relationships. A programmer will usually choose to store only a subset of the simplices and the incidence relationships, deriving the rest as needed.

All five programs store the set of tetrahedra, and for each tetrahedron t, references to its vertices and *neighbors*—a neighbor is another tetrahedron that shares a common triangle with t. A *corner* is the use of a vertex in a tetrahedron. Two corners are opposite if their tetrahedra are neighbors, but neither is involved in the shared triangle.

It is common to include a point at infinity, ∞, so that for every triangle $\{a, b, c\}$ on the convex hull, there is a tetrahedron $\{\infty, a, b, c\}$. Thus, each tetrahedron in the Delaunay has exactly 4 neighbors.

2.3 Incremental Construction

Each of the five programs compute the Delaunay incrementally, adding one point at a time. A new point p is added in two steps: First, a point location routine finds the tetrahedron (or some sphere) that was formerly empty, but that now contains the new point p. Second, an update routine removes tetrahedra that no longer have an empty sphere after adding p and fills in the hole with tetrahedra emanating from p. The running time of an incremental algorithm is proportional to the number of tetrahedra considered in point location, plus the total number of tetrahedra created.

The worst-case number of tetrahedra created in adding a vertex is linear, so the total number of tetrahedra is at most quadratic. This is also the worst-case number in any one tetrahedralization, and simple examples, such as $n/2$ points on each of two skew lines or curves, give a matching lower bound. Nevertheless, linear-size Delaunay tessellations are most commonly observed—the practice is better than the theory predicts. Some theoretical works explain this under assumptions on the input such as random points or uniform samples from surfaces [5, 16, 21].

For the linear-sized Delaunay tessellations observed in practice, point location can actually become the bottleneck in 3d, as it is in 2d. There are a wide variety of point location algorithms in the programs we survey, so we will discuss this primarily in Section 3.4.

2.4 Numerical Computations

The geometric tests in Delaunay code are performed by doing numerical computations. The most important is the InSphere test. Let p be a point whose Cartesian coordinates are p_x, p_y and p_z. We can represent p by a tuple $(p_1, p_x, p_y, p_z, p_q)$, where $p_1 = 1$ is a homogenizing coordinate and $p_q = p_x^2 + p_y^2 + p_z^2$. Mathematically, any positive scalar multiple of p can be taken to represent the same point, but for computation, we prefer the computer graphics convention that $p_1 = 1$, and assume that the Cartesian coordinates are b-bit integers. The special point $\infty = (0, 0, \ldots, 0, 1)$, representing the *point at infinity*, is the sole exception. Four non-coplanar points a, b, c and d define an oriented sphere and point p lies inside, on, or outside of the sphere depending on whether the sign of `InSphere`$(a, b, c, d;\, p)$ in equation 1 is negative, zero, or positive.

$$\texttt{InSphere}(a, b, c, d;\, p) = \begin{vmatrix} a_1 & a_x & a_y & a_z & a_q \\ b_1 & b_x & b_y & b_z & b_q \\ c_1 & c_x & c_y & c_z & c_q \\ d_1 & d_x & d_y & d_z & d_q \\ p_1 & p_x & p_y & p_z & p_q \end{vmatrix} \tag{1}$$

Note that if one of the four points on the sphere is ∞, the determinant is equal to an orientation determinant that tests a point against a plane. Therefore, when a tetrahedron includes the ∞ vertex, we can still use this determinant to perform the InSphere test on its sphere and a chosen point; the test will return the position of the point with respect to an "infinite sphere" that is an oriented convex hull plane.

Computers store numbers with limited precision and perform floating-operations that could result in round-off errors. In the Delaunay algorithms, round-off errors change the sign of a determinant and produce the wrong answer for an InSphere test. Therefore, we look at the *bit complexity* of the numerical operations: Assuming that the input numbers are b-bit integers, how large can the results of an algebraic evaluation be as a function of b?

The InSphere determinant can be expanded into an alternating sum of multiplicative term, each of degree five. Therefore, if we use the determinant directly, we need at least $5b$ bits to compute each multiplicative term correctly. The determinant itself can take no longer than $5b$ bits, since the InSphere determinant gives the volume of a parallelepiped in R^4, where the thickness of the parallelepiped along the x, y, z and the lifted dimension take no more than b, b, b and $2b$ bits, respectively.

Knowing that, e.g., a is a finite point, and that the homogenizing coordinate for points is unity, we can rewrite the determinant to depend on the differences in coordinates, rather than absolute coordinates by just subtracting the row a from all finite points, and then evaluate the determinant. The last coordinate can also be made smaller by lifting after subtraction, but this is usually not done for two reasons: first, it adds extra squaring operations that must be done within each InSphere determinant, and second, it makes it harder to generalize to compute power diagrams.

When an InSphere determinant is zero, then the five points being tested lie on a sphere, and are not in general position. (Subjecting the points to a random perturbation will make them no longer co-spherical, except for a set of measure zero.) Edelsbrunner and Mücke[18] showed how to simulate general position for determinant computations by infinitesimal perturbations of the input points, and there have been many approaches since. We describe the approaches taken by the different programs in Section 3.6.

3 Comparison of Delaunay Codes

With this background, we elaborate on the engineering choices made in the five programs for representation, arithmetic, perturbation, update and point location. A summary table is provided at the end of the section. Many of the implementation details we report here cannot be found in the research papers describing the programs, though they do have impact on the performance of the program.

3.1 Implementation goals

The five programs that we survey were implemented with different goals in mind.

CGAL is a `C++` geometric algorithm library that includes a `Delaunay_triangulation_3` class that encapsulates functions that implement Delaunay tetrahedralization [14]. It uses traits classes to support various types of arithmetic and point representations; we tested both `Simple_cartesian<double>`, which uses floating point arithmetic and `Filtered_kernel<Simple_cartesian<double>>`, an exact arithmetic with floating point filters.

Clarkson's hull [12] computes convex hull of dimension 2, 3 and 4 by an incremental construction that can either shuffle the input points or take them as is. It uses a low bit-complexity algorithm to evaluate signs of determinants in double precision floating point.

Qhull [6], initially developed at the geometry center of University of Minnesota, is a popular computing convex hulls in general dimensions. It supports many geometric queries over the convex hull and connects to geomview for display. It has a many more lines devoted to debugging and portability than to the actual algorithm.

Shewchuk's pyramid [29] was developed primarily to generate quality tessellation of a solid shape. In addition to taking points and producing the Delaunay tessellation, it can take lines and triangles and compute a conforming Delaunay, adding points on these features until the final tessellation contains, for each input feature, a set of edges or triangles is a partition of that feature.

Our program, tess3, is specialized to the Delaunay tessellation of near-uniformly spaced points with limited precision, of the sort found in the crystallographic structures deposited in the PDB [1]. We have been pleased to find that it also works with NMR structures, which often have several variants of the same structure in the same file, and therefore violate the separation assumptions under which our code was developed.

3.2 Representation

Each program stores pointers from tetrahedra to their neighbors. Pyramid and tess3 have special ways to indicate which corners in a neighboring tetrahedra correspond: Pyramid stores four bits with each neighbor pointer to indicate the orientation of the neighboring tetrahedron and location of the vertices of the shared triangle. Tess3 uses a corner-based representation that is a refinement of the structure of Paoluzzi et al. [27] or Kettner et al. [22]. An array stores *corners* so that each subsequent block of four corners is one tetrahedron. Each corner points to its vertex and its *opposite* corner—the corner in the neighboring tetrahedron across the shared triangle. Each block is stored with vertices in increasing order, except that the first two may be swapped to keep the orientation positive. The correspondence between vertices in neighboring tetrahedra, where vertex $0 \leq i < 4$ is replaced by vertex at position $0 \leq j < 4$, can be recorded in a table indexed by i, j. This supports operations such as walking through tetrahedra, or cycling around an edge without requiring conditional tests.

Since a tetrahedron's sphere can be used repeatedly for InSphere tests, the minors of the determinant expanded along the last row can be pre-computed and stored in a vector S so that the test becomes a simple dot product: $\texttt{InSphere}(a, b, c, d; p) = S \cdot p$. Hull, pyramid, and tess3 store these sphere vectors.

3.3 Incremental Computation

Each of the programs must update data structure as tetrahedra are destroyed and created. One of the biggest decisions is whether an algorithm uses flipping [20] to always maintain a tessellation of the convex hull, or uses the Bowyer-Watson approach [9, 30] of removing all destroyed tetrahedra, then filling in with new. Flipping assigns neighbor pointers to twice as many tetrahedra, since many tetrahedra created by flips with a new vertex p are almost immediately destroyed by other flips with p.

Amenta, Choi and Rote [4] pointed out that the number of tetrahedra is not the only consideration. Since modern memory architecture is hierarchical, and the paging policies favor programs that observe locality of reference, a major concern is *cache coherence:* a sequence of recent memory references should be clustered locally rather than randomly in the address space. A program implementing a randomized algorithm does not observe this rule and can be dramatically slowed down when its address space no longer fits in main memory. Their Biased Randomized Insertion Order (BRIO) preserves enough randomness in the input points so that the performance of a randomized incremental algorithm is unchanged but orders the points by spatial locality to improve cache coherence. More specifically, they partition the input points into expected $O(\log n)$ levels as follows: Randomly sample half of the input points and put them in the first level; for each of the following level, sample half of the points left. The points with each level are then ordered by bucketing them with an octree and traversing the buckets in a depth-first order.

To partition the points, tess3 uses a deterministic approach that we call *bit-levelling.* Bit-levelling group the points whose three coordinates share i trailing zeros (or any other convenient, popular, bit pattern) in the ith level. Levels are inserted in increasing order, and points within each level are ordered along a space-filling curve. With experimentally-determined data, the least-significant bits tend to be random, so bit leveling generates a sample without the overhead of generating random bits. The real aim for bit-levelling, however, is to reduce the bit-complexity of the InSphere computation. Recall that when we evaluate the determinant for the InSphere test, one point can be used for the local origin and subtracted from all finite points. Using floating point, the effective number of coordinate bits in the mantissa is reduced if some of the most- and/or least-significant bits agree. Since the points are assumed to be evenly distributed (the next section describes how tess3 adds all the points ordered along a Hilbert curve), in the final levels the points used for InSphere tests tend to be close and share some most-significant bits. Since bit-levelling forms the ith level by grouping points with the same i least-significant bits, giving cancellation in the early, sparse levels as well.

3.4 Point location

In theory, point location is not the bottleneck for devising optimal 3D Delaunay algorithms. In practice, however, the size of the neighborhood updated by inserting a new point is close to constant, and point location to find the tetrahedron containing a new point p can be more costly than updating the tetrahedralization if not done carefully.

Hull and Qhull implement the two standard ways to perform point location in randomized incremental constructions of the convex hull: Hull maintains the history of all simplices, and searches the history dag to insert a new point. QHull maintains a conflict list for each facet of the convex hull in the form of an *outside set*, which is the set of points yet to be processed that can "see" the facet. These are equivalent in the amount of work done, although the history dag is larger, and the conflict list requires that all points be known in advance.

The other programs use some form of walk through the tetrahedra. Pyramid uses the *jump-and-walk* introduced by Mücke et al. [25] for point location. To locate a point p in a mesh of m tetrahedra, it measures the distance from p to a random sample of $m^{1/4}$ tetrahedra, then walk from the closest of these to the tetrahedron containing p. Each step of the walk visits a tetrahedron t, shoots a ray from the centroid of t towards p, and go to the neighboring tetrahedron intersected by the ray. In the worst case, this walk may visit almost all tetrahedra, but under some uniformity assumptions the walk is $O(n^{1/4})$; it would be $O(n^{1/3})$ if a single starting point was used.

CGAL uses the Delaunay hierarchy scheme invented by Devillers [15], which combines a hierarchical point location data structure with jump-and-walk. Delaunay hierarchy takes a small random sample of the input points P' and create a sequence of levels so that the 0th level is P' and each level is a random subset of the previous level. A Delaunay tetrahedralization is created for each level, and the tetrahedra that share vertices between levels are linked. To locate p, at each step, a jump-and-walk is performed within a level to find the vertex closest to p. This vertex is then used as the starting point for the next step. This point location scheme is an improvement over the jump-and-walk alone when the input point are not uniformly distributed in space. The walk CGAL's implements is a "zig-zag" walk: at each step, visit a tetrahedron t_1, choose a triangle face (out of at most three) so that p and t_1 are on the opposite side of the plane through the triangle and walk to the neighboring tetrahedron t_2. This is easier to implement than the straight line walk or the walk Pyramid implements, and the acyclic theorem by Edelsbrunner [17] guarantees that the walk terminates.

Tess3 locates a sphere (rather than a tetrahedron) containing a new point p by a zig-zag walk from the last point inserted. Tess3 uses sphere equations to perform the plane test. Suppose neighboring tetrahedra t_1 and t_2 share a triangle in plane P_{12}, have vertices q_1 and q_2 that are not on P_{12}, and have circumspheres $S_1 \neq S_2$. Tess3 can determine the side of plane P_{12} that contains p by the sign of $P_{12} \cdot p = q_1(S_2 \cdot p) - q_2(S_1 \cdot p)$. Note that this reuses the InSphere tests already performed with p, and reduces the orientation determinant to the difference of two dot products. When $S_1 = S_2$, a degenerate configuration, we would have to test the plane, but this happens rarely enough that tess3 simply chooses the side randomly.

To make the walks short, tess3 initially places all input points into a grid of $N \times N \times N$ bins, which it visits in Hilbert curve order so that nearby points in space have nearby indices [24]. To order a set of points with a Hilbert curve, tess3 subdivides a bounding cube into $(2^i)^3$ boxes and reorders the points using counting sort on the index of the box on the Hilbert curve that contains each point. Points in a box can be reordered recursively until the number of points in each subbox is small. Parameter i is chosen large enough so that few recursive steps are needed, and small enough that the permutation can be done in a cache-coherent manner. We find that having $(2^3)^3 = 512$ boxes works well; ordering 1 million points takes between 1–2 seconds on common desktop machines.

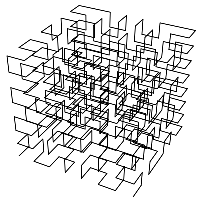
Figure 1: Hilbert curve for an $8 \times 8 \times 8$ grid.

3.5 Numerical Computations

Each of the programs takes a different approach to reducing or eliminating errors in numerical computation.

Qhull and tess3 use floating point operations exclusively, and are written so that they do not crash if the arithmetic is faulty, but they may compute incorrect structures. Qhull checks for structural errors, and can apply heuristics to repair them in postprocessing. Tess3 assumes that input points have limited precision and are well distributed, and uses bit-leveling and Hilbert curve orders to try to ensure that the low-order or high-order (or both) bits agree, and that the bit differences take even fewer bits of mantissa. We explore this in more detail in the experiments.

CGAL uses an exact arithmetic, but uses floating point filters first, checking error bounds and using exact computation only when necessary.

Hull uses an low bit-complexity algorithm for evaluating the sign of an orientation determinant that is based on Graham-Schmidt orthogonalization. The idea is that since we only care about the sign of the determinant, we can manipulate the determinant so far as its sign does not change. The implementation uses only double precision floating operations and is able to compute the signs of InSphere determinants exactly for input whose coordinates have less than 26 bits.

Pyramid uses adaptive precision [28]. to carry out the arithmetic on just enough significant bits that it can guarantee the correct sign is computed for a predicate.

3.6 Perturbation to handle degeneracies

In Delaunay computation, InSphere determinants equal to zero are degeneracies—violations of the general position assumption that affect the running of the algorithm. They occur when a point is incident either on the Delaunay sphere of some tetrahedron or on the plane of the convex hull, which can be considered a sphere through the point at infinity.

Qhull allows the user to select a policy when the input contains degeneracies or the output contains errors: either it perturbs the input numerically and tries again, or it attempts to repair the outputs with some heuristics.

Edelsbrunner and Mücke[18] showed how to simulate general position for the Delaunay computation directly (handling 2), but advocated "perturbing in the lifted space" as easier. For lifted points, perturbation can be handled by simple policies: either treat all 0s as positive or treat them all as negative. These are consistent with perturbing a point outside or inside the convex hull in 4D, respectively. These perturbation schemes have three short-comings; usually only the third has impact on practice.

1. The output from the perturbation is dependent on the insertion order of the points.
2. Perturbing the lifted points in 4D may produce a tessellation that is not the Delaunay tetrahedralization of any actual set of points.
3. The perturbation (either the "in" or the "out" version) may produce "flat" tetrahedra near the convex hull. Figure 2 illustrates the 2D analog.

Hull and pyramid perturb points inside.

Tess3 first perturbs a point p down in the lifted dimension so that it is not on any finite sphere; next, if p is on an infinite sphere S, p is perturbed either into or away from the convex hull in 3D depending on these two cases: If q is inside the finite neighbor of S, q is perturbed into the convex hull; otherwise, q is perturbed away. This perturbation guarantees that there are no flat tetrahedra (handling 3), yet is still simple to implement.

CGAL removes degeneracies by "perturbing the world", a technique introduced by Alliez et al. [3]. Their perturbation is consistent with an infinitesimal perturbation of the coordinate system so that it is independent of the order in which the points are inserted. Another important feature of their perturbation is that it is carried out before the points are lifted into one higher dimension. Thus, they handle 1 and 2 (but not 3).

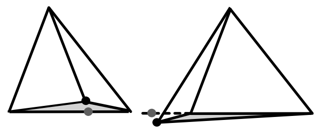
Figure 2: Perturbing point inside (left) or outside (right) produces flat triangles (shaded).

Program	flipping?	point location	arithmetic	caching spheres?	degeneracy
CGAL (2.4)	no	Delaunay hierarchy	Floating point filter.	no	perturbing E^3
Hull	no	history dag	exact arithmetic	yes	perturb points into hull in E^4
Pyramid	yes	jump-and-walk	adaptive exact arithmetic	no	Perturb points into hull in E^4. Remove flat tetrahedra by post-processing
QHull (2003.1)	no	outside set	floating point	no	Perturb points into hull in E^4. Remove flat tetrahedra by post-processing.
Tess3	no	Hilbert ordering, zig-zag walk	floating point	yes	Perturbation in E^4 with no flat tetrahedra.

Table 1: Program comparison summary.

4 Experiments

In this section we report on experiments running the five programs on randomly generated points and on PDB files. We first report on running time. Then, because tess3 uses only standard floating point arithmetic, we report on the (small number of) errors that it makes.

We have tried to use the latest available codes of these programs. Hull and Pyramid codes were given to us by the authors. CGAL and Qhull codes were downloaded from their web sites. The latest version of CGAL in April, 2004 is 3.0.1; however, we found that the it is more than two times slower than CGAL 2.4 due to possibly bugs in the `gcc` compiler. We therefore proceed to use CGAL 2.4. Qhull 2003.1 we used is the latest version.

The plots in Figures 3 and 4 show the running time comparisons using random data and PDB data as input, respectively, using a logarithmic scale on the x axis and the running time per point in micro-seconds on the y axis. Using time per point removes the expected linear trend and allows easier comparison across the entire x-coordinate range. Lines indicate the averages of ten runs; individual runs are plotted with markers. Hull's running time is much slower than the rest of the programs. So in Figure 4, we omitted its plot so other plots can compare more easily. Its time per point are mostly between 0.4–0.6 ms. All timings are performed on a single processor of a Dell with dual 2GHZ Intel Xeon processors and 2GB of memory, running Red Hat Linux 7.3.

We generated random data by choosing coordinates uniformly from 10-bit non-negative integers. This ensures that the floating point computations of both Qhull and tess3 are correct. For the PDB data, for each input size n that is indicated on the x-axis, try to find 10 files whose number of atoms are closest to n, though there is only one (with the indicated name) for each of the three largest sizes.

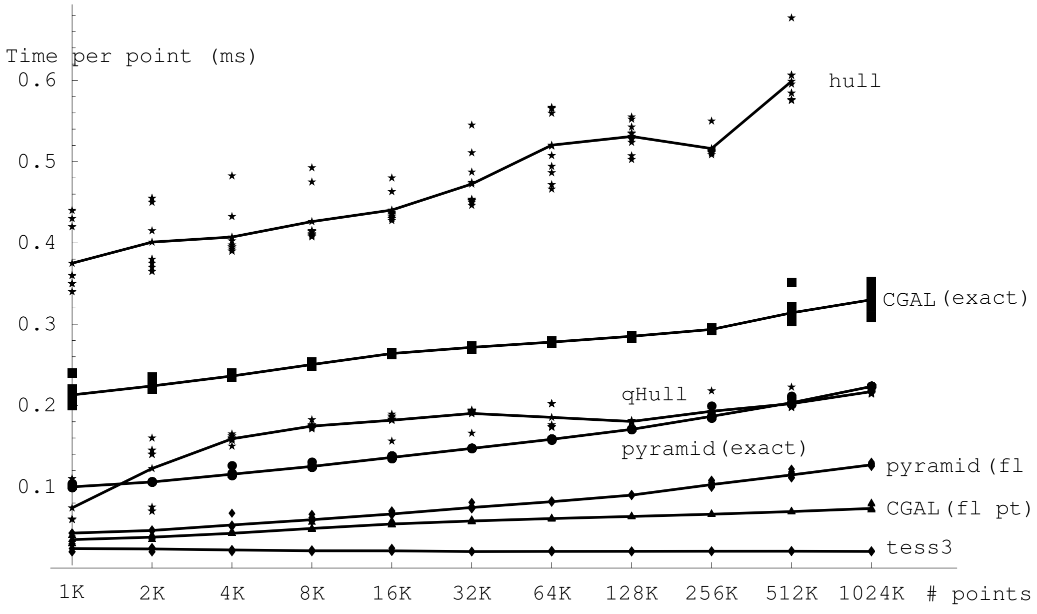

Figure 3: Running time of the programs with 10 bit random points.

There are a few immediate conclusions: The ordering of programs, tess3 < CGAL (fp) < pyramid (fp)< Qhull & pyramid < CGAL < hull, is consistent, although hull had extra trouble with the PDB files and is therefore not shown. There is clearly a penalty for exact arithmetic, because even when an exact arithmetic package is able to correctly evaluate a predicate with a floating point filter, it must still evaluate and test an error bound to know that it was correct. Time per point shows some increase for everything but CGAL and tess3, which we believe is due to point location.

	total created spheres/tetra	MakeSphere (μs)	InSphere (μs)		Update (μs)	Point Location	
			fl. pt.	exact		fl. pt.	exact
CGAL (2.4)	2,760,890	–	0.06^p 0.24^t	18.5^p 1.72^t	0.1^p 16.1^t	$21.8\%^p$ $22.1\%^t$	$25.3\%^p$ $27.9\%^t$
Hull	2,316,338	10.02	0.14	–	2.40	–	73.1%
Pyramid	$5{,}327{,}541^f$ $2{,}662{,}496^n$	-	0.21	0.72	2.44	50.2%	38.1%
QHull (2003.1)	2,583,320	0.65	0.12		> 4.39	9.0%	–
Tess3	2,784,736	0.13	0.04	–	2.42	$3.88\%^h$ $0.43\%^w$	–

Table 2: Summary of timings from profiler, running the programs against the same 100k randomly generated points with 10 bit coordinates. Notes: For pyramid tetrahedra creation, numbers marked f include all initialized by flipping and marked n include only those for which new memory is allocated—equivalently, only those not immediately destroyed by a flip involving the same new point. For CGAL timings, p indicates profiler and t direct timing. For tess3 point location, h includes the preprocessing to order the points along Hilbert curve; w is walk only.

To further explain the difference in these programs' running time, we used the `gcc` profiler to determine the time-consuming routines. There are caveats to doing so; function level profiling turns of optimizations such as inlining, and adds overhead to each function call, which is supposed to be factored out, but may not be. (This affects CGAL the most. , With its templated `C++` functions we could not get reasonable profiler numbers, so we also tried to time its optimized code, but this has problems with clock resolution.) The table shows some of our findings for running the programs against the same 100k randomly generated points with

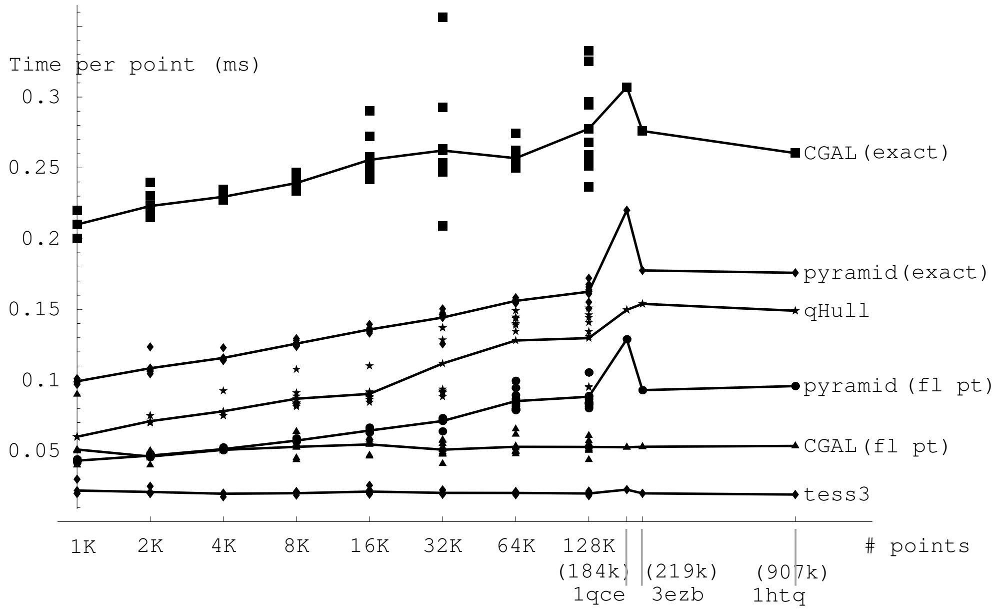

Figure 4: Running time of the programs with PDB files.

10 bit coordinates.

The “total created spheres/tetra” column shows that flipping must initialize many more tetrahedra. “MakeSphere” and “InSphere” columns, which record time to make sphere equations and test points against them, indicate that there are speed advantages to using native floating point arithmetic for numerical computations. Even simple floating point filters must check error bounds for computations. Note that for the programs that do not cache spheres, the InSphere test is a determinant computation. The “Update” column indicates the time to update the tetrahedral complex and does not include any numerical computation time. The “Point Location” column indicates the percentage of time a program spends in point location (for tess3, this number includes the time for sorting the points along the Hilbert curve).

As we can see from the table, tess3 benefited particulary from its fast point location. Our data structure for the tetrahedral complex, although requiring the smallest amount of space, is no more expensive than others to be updated. We should also point out some observations about the bottlenecks of the other programs: Qhull’s data structure is expensive to update and the code contains debugging and option tests; Hull’s exact arithmetic incurs a significant overhead even when running on points with few bits; Pyramid was bogged down mainly by its point location, which samples many tetrahedra.

4.1 Point Ordering

Since tess3 does not use exact arithmetic, we did additional runs using audit routines to check the correctness of the output. We first check the topological correctness—that is, whether our data structure indeed represents a simplicial complex (it always has)—then we then check the geometric correctness by testing (with exact arithmetic) for each tetrahedron if any neighboring tetrahedron vertex is inside its sphere. We also did some runs checking every InSphere test.

For the random data with 10 bits there are no errors, although we do find geometric errors for larger numbers of bits. For 20,393 PDB files, our program computes topologically correct output on all files and geometrically correct output on all except one.

Figure 5 displays the 266 incorrect tetrahedra, and shows that the assumptions of uniform distribution are egregiously violated. The comments to 1H1K state: "This entry corresponds to only the RNA model which was built into the blue tongue virus structure data. In order to view the whole virus in conjunction with the nucleic acid template, this entry must be seen together with PDB entry 2BTV."

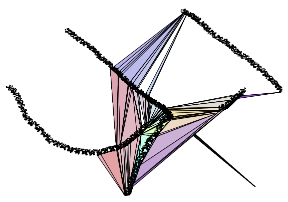
Figure 5: 1H1K points and bad tetrahedra

We investigate how much the ordering points along a Hilbert curve and bit-levelling helps speed up tess3 and make it more resistent to numerical problems. Figure 6 shows a log-log plot of the the percentage of InSphere computation that contain round-off errors with three different ordering: random, Hilbert ordering only, and Hilbert ordering combined with bit-levelling. The percentages of errors are affected by both the number of coordinate bits and the number of points in the input; the plot illustrates variations in both of these controls. Given an input with a certain number of coordinate bits, we can see that the combined ordering has the lowest amount of numerical errors—and the difference becomes more dramatic as the number of input points increases. We should emphasize that the InSphere errors here are observed during the incremental construction and the final output always contains much fewer errors. For example, for the combined ordering, no output contains an error until the number of coordinate bits reaches 17.

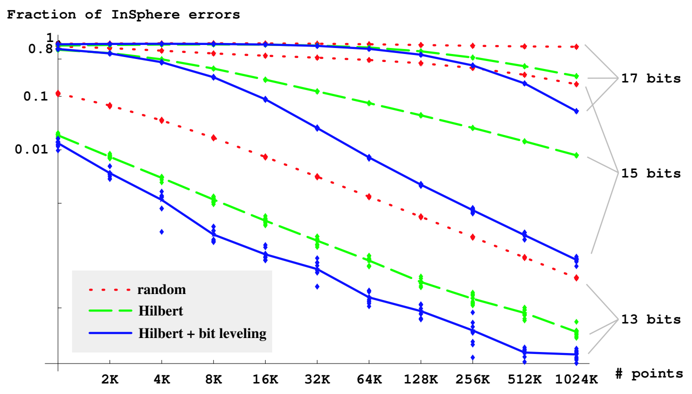

Figure 6: Semilog plot showing percentage of InSphere tests with sign errors by number of points n and number of coordinate bits, for three orderings. We plot a dot for each of 10 runs for given n and bit number, and draw lines through the averages of 10 runs.

Since BRIO [4] also uses a spatial-locality preserving ordering to speed up point location, we close by comparing BRIO insertion order with a Hilbert curve order. Figure 7 compares the running times of CGAL, which uses a randomized point location data structure, under the BRIO and Hilbert insertion orders. The Hilbert curve is faster on average and has a smaller deviation. This suggests that for input points that are uniformly distributed, adding randomness into the insertion ordering perhaps will only slow down the program.

5 Conclusions

We have surveyed five implementations of 3D Delaunay tessellation and compared their speed on PDB files and randomly generated data. We find that the biggest bottlenecks of the programs are, first, overhead imposed by exact arithmetics and, second, point location. Exact arithmetic is required if the input bit precision is arbitrary and the output is required to be exact. However, for particular application data, such as PDB files, we show that it is possible to have an implementation that works well even when straightforward bit-complexity analysis suggests otherwise. For point location, our experiments suggest that randomization should be avoided if the input points are uniformly distributed in space.

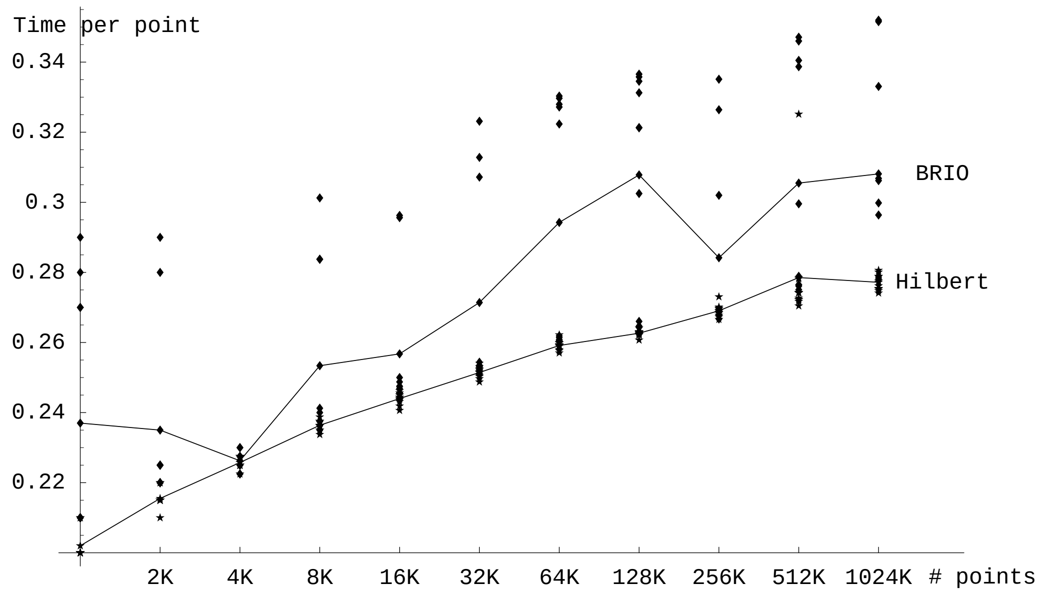

Figure 7: Running time of the CGAL Delaunay hierarchy using BRIO and Hilbert point orders.

6 Acknowledgments

We would like to thank Jonathan Shewchuk, Kenneth Clarkson and Sunghee Choi for providing us with their codes and Sylvain Pion for the quick response to our questions about CGAL. This research has been partially supported by NSF grant 0076984.

References

[1] Brookhaven protein data bank. http://www.rcsb.org.

[2] *IEEE Standard for binary floating point arithmetic, ANSI/IEEE Std* $754 - 1985$. IEEE Computer Society, New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[3] Pierre Alliez, Olivier Devillers, and Jack Snoeyink. Removing degeneracies by perturbing the problem or perturbing the world. *Reliable Computing*, 6:61–79, 2000.

[4] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con BRIO. In *Proceedings of the Nineteenth ACM Symposium on Computational Geometry*, pages 211–219, 2003.

[5] Dominique Attali, Jean-Daniel Boissonnat, and Andre Lieutier. Complexity of the Delaunay triangulation of points on surfaces the smooth case. In *Proceedings of the Nineteenth ACM Symposium on Computational Geometry*, pages 201–210, 2003.

[6] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996. `http://www.qhull.org/`.

[7] J.-D. Boissonnat and M. Yvinec. *Géométrie algorithmique.* Ediscience international, Paris, 1995.

[8] J.-D. Boissonnat and M. Yvinec. *Algorithmic geometry.* Cambridge University Press, UK, 1998. translated from [7] by H. Brönnimann.

[9] A. Bowyer. Computing Dirichlet tesselations. *Comput. J.*, 24:162–166, 1981.

[10] K. Q. Brown. Voronoi diagrams from convex hulls. *Inform. Process. Lett.*, 9(5):223–228, 1979.

[11] K. Q. Brown. *Geometric transforms for fast geometric algorithms.* Ph.D. thesis, Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1980. Report CMU-CS-80-101.

[12] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 387–395, 1992.

[13] B. Delaunay. Sur la sphère vide. A la memoire de Georges Voronoi. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskih i Estestvennyh Nauk*, 7:793–800, 1934.

[14] O. Devillers. Improved incremental randomized Delaunay triangulation. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 106–115, 1998.

[15] Olivier Devillers. The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13:163–180, 2002.

[16] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete Comput. Geom.*, 6:343–367, 1991.

[17] H. Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 145–151, 1989.

[18] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[19] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, January 1994.

[20] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 43–52, 1992.

[21] Jeff Erickson. Dense point sets have sparse Delaunay triangulations. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 125–134. Society for Industrial and Applied Mathematics, 2002.

[22] Lutz Kettner, Jarek Rossignac, and Jack Snoeyink. The Safari interface for visualizing time-dependent volume data using iso-surfaces and a control plane. *Comp. Geom. Theory Appl.*, 25(1-2):97–116, 2003.

[23] Yuanxin Liu and Jack Snoeyink. Sphere-based computation of Delaunay diagrams in 3d and 4d. In preparation.

[24] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering*, 13(1):124–141, 2001.

[25] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.

[26] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, UK, 1992.

[27] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension-independent modeling with simplical complexes. *ACM Trans. Graph.*, 12(1):56–102, January 1993.

[28] Jonathan R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[29] Jonathan R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 86–95, 1998.

[30] D. F. Watson. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *Comput. J.*, 24(2):167–172, 1981.

[31] David F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data.* Pergamon, 1992.