Almost-Delaunay Simplices : Robust Neighbor Relations for Imprecise 3D Points using CGAL

Deepak Bandyopadhyay Jack Snoeyink*

Abstract

This paper describes a CGAL implementation of a new computational geometry technique, almost-Delaunay simplices [1], in 3D. Almost-Delaunay simplices capture possible sets of Delaunay neighbors in the presence of a bounded perturbation, and give a framework for nearest neighbor analysis in imprecise point sets such as protein structures. The implementation, available on http://www.cs.unc.edu/~debug/papers/AlmDel is faster and more memory efficient than our prototype MATLAB implementation, and enables us to scale our neighbor analysis to large sets of protein structures, each with 100-3000 residues.

1 Motivation & Definitions

The Delaunay tessellation (DT) is a geometric structure that defines a nearest neighbor relation on a set of points in space (known as *sites*), using an exact geometric criterion – the "empty sphere test" [6]. The DT has been used in the analysis of protein structure[1], among other applications, for detecting pockets and cavities [9, 10], scoring packing interactions and distinguishing native proteins from artificial decoy structures [5, 11, 8], and detecting patterns of local structure [12].

In applications that deal with protein data, point coordinates are known only imprecisely – factors such as experimental errors and protein flexibility introduce small changes in the point coordinates that can produce different sets of Delaunay simplices[2]. So, a natural question arises: whether analyses based on DT are stable and robust under changes to the input coordinates. To answer this question, we defined the *almost Delaunay simplices* [1] that give possible neighbors under a bounded perturbation ϵ of the input.

Definition 1 (Almost-Delaunay) *For a finite set of point sites P and $\epsilon \geq 0$, a* perturbation by ϵ *adds a vector $|v_i| \leq \epsilon$ to each $p_i \in P$. The set of* almost-Delaunay simplices *$AD(\epsilon)$ contains a k-tuple $S \subset P$ iff there exists a perturbation by ϵ producing $S_\epsilon \subset P_\epsilon$ such that S_ϵ has a circumscribing sphere containing no points of P_ϵ.*

Each k-tuple, for $1 \leq k \leq d+1$, is an almost-Delaunay simplex for some minimum value of ϵ, denoted its *threshold*. The Delaunay simplices are those with threshold zero.

We related computation of the almost-Delaunay (AD) threshold to a variant of a minimum-width annulus problem, and proved properties of almost-Delaunay simplices [1] that we specialize here for tetrahedra in 3D: **1).** Local minima of the threshold correspond to annuli defined by 5 points in general position, with at least 2 on the inner sphere and 2 on the outer, though a slab (annulus with infinite center and radii) is a special case defined by 4 points. **2).** Points on the inner sphere form a Delaunay simplex. **3).** Center of the minimum-width annulus for a simplex S can come from the intersection of the Voronoi diagram of all points and the furthest-point Voronoi diagram of the points in S. We then described an algorithm based on these properties.

We identified two parameters arising from biological constraints that help us speed up our algorithm: the **edge length prune**, i.e. the maximum edge length between two neighboring sites, typically 10 Å; and the **threshold cutoff**, i.e. the maximum perturbation allowed, 0.5–2 Å depending on the type of perturbations of interest. Thus, we begin by computing the thresholds for almost-Delaunay edges, and from them, for triangles and tetrahedra as in Figure 1.

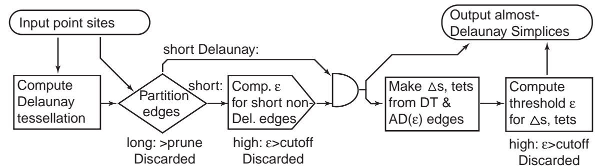

Figure 1. Processing AD edges then AD simplices

In this paper, we describe our implementation of the AD threshold computation for edges, triangles and tetrahedra in 3D, using CGAL. We discuss some design decisions, unexpected issues that arose during development, solutions and the things we learned. Finally we report on verification and timing comparison with the previous implementation.

2 CGAL implementation

We had a MATLAB implementation of the entire algorithm in 3D that was well tested and fast, aggressively using MATLAB's vectorized operations to trade time for space. But it

*Portions of this research were supported by NSF grant 0076984.

[1] For an introduction to protein structure see [3]

[2] In d dimensions, a k-simplex, $k <= d$ is defined by $k+1$ affinely independent points; thus tetrahedra are 3-simplices.

became unreasonably slow and ran out of memory as the input point set grew to 1000 points (large protein chains, represented by one point per residue) or 3000 points (medium-length chains with all atoms). To improve speed and memory utilization, we decided to completely rewrite the code using C++ and CGAL.

The steps for the AD computation in 3D are below:

- Generate list of short non-Delaunay edges, which are the potential AD edges, from a proximity graph.
- Consider any such edge, $\overline{pq}$.
    1. Compute the *candidate centers* of a minimum-width annulus, which are vertices of the intersection of the Voronoi diagram with the bisector plane of $\overline{pq}$. Infinite edges of the Voronoi diagram are candidate centers of slabs and are stored as directions. We use Brown's *lifting* technique [4], which involves computation of the lower convex hull in dual space.
    2. Evaluate the width of the annulus at each candidate center, which has p and q on the outer sphere and three points on the inner sphere (two for slabs).
    3. Find the minimum width over all candidate centers. Half of this width is the AD threshold for edge $\overline{pq}$.
    4. Retain the edge as *valid* if its threshold is $\leq$ cutoff, otherwise remove it from the proximity graph.
    5. Retain candidate centers with threshold $\leq$ cutoff (and candidate centers connected to them by Voronoi edges) for the next stage of computation.
- Output the list of valid AD edges and their thresholds.
- For each valid AD edge, $\overline{pq}$:
    - Generate possible AD triangles by querying the proximity graph for points r that are near both p and q.
    - For each point r:
        1. Select from $\overline{pq}$'s candidate centers, those whose annulus contains r. This is a linear half-plane constraint on the candidate centers, since all centers closer to r than to p or q would lie on the side facing away from p and q of the line equidistant to all three points.
        2. Generate new candidate centers at the intersection points of edges between candidate centers and the constraint, denoted *constraint cuts*. The existence of minimum-width annuli at constraint cuts is the reason we stored candidate centers with high threshold that were connected by edges to ones with threshold $\leq$ cutoff.
        3. Compute the annulus width at constraint cuts. p, q, r are on the outer sphere and the two points of the Voronoi edge are on the inner sphere.
        4. Find the minimum annulus for edge $\overline{pq}$ from among candidate centers satisfying the constraint generated by r, and constraint cuts of r.
        5. Record the threshold if it is less than cutoff.
    - The AD tetrahedron algorithm is similar. We generate the possible AD tetrahedra by querying the proximity graph for two points r and s that are near each other and also near both p and q. Then we generate two constraints (for r and s) and look for the minimum annulus among candidate centers that satisfy *both* constraints.
- The threshold for $\triangle pqr$ is the minimum of the threshold of $\overline{pq}$ constrained by r, $\overline{pr}$ constrained by q, and $\overline{qr}$ constrained by p. Similarly the threshold for a tetrahedron is the minimum constrained threshold over all the edges.
- A final special case: there are AD triangles and tetrahedra with all edges Delaunay, and in our point sets these always arise when three Delaunay tetrahedra share an edge (say $\overline{pq}$) whose vertices lie on opposite sides of the plane of the three other vertices (a, b, c); $\triangle abc$ and tetrahedra $abcp$ and $abcq$ are almost-Delaunay. We enumerate these triangles and tetrahedra during a traversal of the Delaunay tessellation, and compute their thresholds separately for efficiency, since we have verified experimentally that their minimum width annuli contain only the 5 points (a, b, c, p, q).

2.1 Lessons from CGAL implementation : Issues and Design Decisions

Proximity data structures: We decided to use two different structures for proximity information, one to store the Delaunay tetrahedra with short edges, and another to store all short edges and non-Delaunay short edges. We considered using CGAL's neighbor searching algorithms, but since proximity has to be calculated only once, and we needed a way to remove pairs of points from the proximity relationship if their AD threshold is above the cutoff, we decided to store proximity in a graph using the Boost Graph Library (www.boost.org) and encapsulated it in a class `ADPointNeighbors`, with methods to list all short edges and to find points or pairs of points near a short edge. The short Delaunay edges, triangles and tetrahedra were encapsulated into another class `MyDelaunay` that uses CGAL's `Delaunay_triangulation_3`. It uses a boolean state stored in each tetrahedron and each of its faces to provide traversals of only the short edges and the triangles and tetrahedra containing them (an earlier implementation where we deleted cells from the triangulation led to precondition violations and instabilities in the traversal). `ADPointNeighbors` holds a reference to `MyDelaunay` for computing non-Delaunay short edges.

Fast and Robust Computation of 3D Convex Hull Using Delaunay Insertion and Static Filters: We started out using the `Cartesian<double>` kernel since we were concerned about speed, and exact computation was not a big priority since the original MATLAB code also used floating-point computation, and protein coordinates were not expected to have degeneracies (or could be perturbed if they did). We faced problems with the convex hull computation; `Convex_hull_incremental_3`, based on an incremental algorithm, was much slower than the MATLAB code that interfaced with Quickhull [2], and `Convex_hull_3`, based on Quickhull, was faster but crashed or went into an infinite loop for several common inputs. Using exact number types and `Lazy_exact_NT` was too slow, and the `Filtered_kernel` robustified predicates, but the convex hull code depended on *exact*

construction of a plane containing three points, on which an orientation test was failing. We changed the convex hull code so that it used a plane class that stores the three defining points and tests orientation against them using a determinant, instead of computing the plane equation. Now the code did not crash, but it was still slower than the equivalent MATLAB code, since it needed the `Filtered_kernel` to work and could not make use of faster static filters[3] since `Convex_hull_3` uses a predicate (`has_on_positive_side_3()`) for which no filtered version was available. We anticipate that this problem may be fixed in a future release of CGAL.

To make use of faster static filters that have been written for the DT [7], we replaced the computation of convex hull by incremental Delaunay insertion – running a DT point location for each point, and inserting only the points found outside the convex hull of the current DT. In the end all faces on the convex hull of the DT (adjacent to the infinite face) were reported as the convex hull. After this modification, the AD edge code using filtered floating point computation was not much slower that the MATLAB code using floating point computation, as shown in Table 1.

Balancing Library Component Use With Custom Development: We found that achieving a balance between using sophisticated existing packages and writing custom code helped speed our development process. For example, since our definition of minimum-width annulus does not require it to contain all points but only points of the simplex, we did not use CGAL's `Minimum_annulus_d` package. Instead we implemented the lifting technique [4]. Since available functions for projecting a 3D point onto a plane and lifting it to a paraboloid were unsuitable (`Plane_3.projection(Point_3)` did not transform coordinates, `Plane_3.to_2d(Point_3)` had a bug, and `Construct_lifted_point_3` lifted to a plane), we hand-coded both operations in a function object. Also, we chose to calculate the fairly simple furthest point Voronoi regions (bisector plane between 2 points, and lines equidistant from 3 points), rather than use code for k^{th}-order Voronoi diagrams such as Julia Flötotto's prototype listed on the CGAL site.

Genericity through Function Objects: CGAL's templated function objects[4] for predicates and data conversions, as well as many that we designed in the spirit of CGAL, played a major role in giving us efficient, reusable code. This was a major redesign from MATLAB. Some function objects we wrote implement the projection operation involved in lifted Voronoi computation; bounding box and centroid computation on an iterator range or a container of points; mapping an iterator range of points to their vertex index numbers; a constraint equation object that can test ranges of candidate center points (including centers at infinity, stored as rays) to find ones that satisfy it, and find intersection points with edges joining two centers ; a sequence indexer to index a sequence container with multiple subscripts; and a sequence minimizer to find the minimum value in a container over a set of valid indices. In our enthusiasm to make these function objects do everything, we gave many of them constructors that take an iterator range and call the STL algorithm `for_each` to apply the same function operation on the elements of that range (by passing it `*this`). However, we needed to pass the state reached during execution on the range back to the calling instance using a method call, since the function object used is copied by value and the state will otherwise be lost. In this example of code from a function object constructor, `out()` is a method that returns the value of state variable `result`. The first statement fails, while the second works.

```
// op() updates result in temp copy, changes lost!
std::for_each(indices_list.begin(),
indices_list.end(),*this);
// get updated result from copy using method
result=std::for_each(indices_list.begin(),
        indices_list.end(),*this).out();
```

Degeneracies Creep In: Though we had assumed that our point sets were degeneracy-free, we found that degenerate configurations for some predicates and constructions did arise, e.g. "almost vertical" segments on which 2D segment-line intersections fail since the intersection point lies outside the bounding box of the segment, and three points whose plane is perpendicular to the bisector plane of two other points, so that when projected on it they are collinear and when lifted their tangent planes do not intersect in a point. When an imprecise test finds an intersection point in the latter case, it gives a negative value of AD threshold that is impossible and must be rejected. Finally, proteins represented by all atom coordinates contain coplanar atoms in the peptide bond and planar sidechains, and certain decoys built on a cubic lattice have cospherical atoms. In these cases we apply a random perturbation to the input of magnitude much smaller than any intended cutoff value, which changes the AD thresholds of simplices at most by a correspondingly small value.

2.2 Correctness, Time complexity and comparison

We verified that our algorithm is implemented correctly by comparing the almost-Delaunay edge, triangle and tetrahedra thresholds output by it with the MATLAB implementation (which was itself tested against a brute-force implementation). The test set for the comparison comprised over 400 proteins, run at different values of the cutoff and prune parameters. The results were found to match in every case.

Our algorithm takes $O(n^2 \log n)$ time for calculation of AD edges and $O(n^2)$ for calculation of AD tetrahedra in

[3] `Static_filters` <K>, an undocumented CGAL kernel that will be merged into `Filtered_kernel` <K>.

[4] objects that declare operator() and can be used as functions with state

typical proteins, proved with some assumptions about the data in [1]. The times taken by the implementation for 12 proteins are shown in Table 1. We observe that for computing edges, CGAL using filtered computation was not much slower than an optimized MATLAB implementation using floating point. For triangles and tetrahedra, CGAL was an order of magnitude faster, partly because the MATLAB implementation was complex and not fully optimized. Both stages of the MATLAB version ran out of memory for input data with 1000–3000 points, while CGAL did not.

PDB ID	# pts	#nonD edges	AD edges (sec)		ADtri/tet (sec)	
			Matlab	CGAL	Matlab	CGAL
1ab8A	115	252	2.1	1.1	3.3	0.6
1jkeA	145	479	4.4	2.6	7.0	1.5
1bjfA	181	474	4.5	3.5	8.5	1.3
1jmkC	222	691	7.4	6.1	11.1	2.4
1g6aA	266	980	11.0	10.3	18.7	4.0
1c8bA	320	1024	11.9	13.6	20.8	4.5
1crzA	397	1557	20.0	25.1	41.3	8.4
1lj8A	481	1604	21.6	30.1	51.3	9.0
1m2oA	718	2491	40.5	72.1	136.4	19.1
1gaxA	862	2903	53.5	97.8	208.1	27.6
1iw7C	999	3390	108.0	140.1	307.8	34.3
1d2rA	2563	31537	nomem	3408	nomem	1193

Table 1. Time taken by two steps of the AD computation for a few protein chains, on a P4 2.0GHz with 512MB of memory. Cutoff was 2.0 Å and edge length prune was 10 Å, except for the last chain which has coordinates for all atoms, closer together than representative points, where cutoff was 0.5 Å and prune was 6.0 Å.

3 Limitations and Future Work

While profiling the current implementation, we found that it wastes significant time in converting the points on the convex hull computed by Delaunay insertion, into a polyhedral surface, using the *Polyhedron_incremental_builder_3* class; on the average, 10-20% of the time taken to compute the convex hull. This step was necessary to interface with code that used CGAL functions to traverse a polyhedral surface returned by *Convex_hull_3*. Some of these functions, such as computing boundary half-edges that correspond to slabs, were difficult to write for a data structure not based on half-edges. Still, rewriting the code to traverse the surface of the DT would remove this overhead.

The implementation has migrated to a series of compilers and platforms as it got more complex through its development, and right now it uses CGAL-3.0[5], and compiles with the Intel compiler version 7.1 on Windows, and `gcc` 3.3.x on CYGWIN and Redhat Linux 9.0.

Though our framework allows computation of the almost-Delaunay simplices in dD, we have a complete implementation only in 3D since the driving problem of analyzing protein structure is in 3D. A 2D implementation is planned. We plan to make almost-Delaunay simplices available as a CGAL extension package, and pursue further algorithmic improvement (make it output sensitive, incremental and kinetic).

[5] a few files were modified to make the *Static_filters* methods accessible

4 Conclusion

As the CGAL project evolves, users find new applications and provide feedback, many of the tricks we used in our implementation will no longer be needed, and the functionality of many external libraries we used will be readily available in CGAL components. In the meantime, we have shared our CGAL experience with fellow users and are happy with its end result – a practical implementation of our algorithm in 3D that robustly handles large datasets and expands the capabilities of our protein structure analysis tools.

Acknowledgments

We thank David O'Brien, resident STL and CGAL expert, and members of the CGAL developer community for help getting started and later for insightful suggestions and solutions. In particular we thank Andreas Fabri who pinpointed the convex hull robustness problem and wrote the fix for robust plane construction and code for Delaunay-insertion convex hull; Sylvain Pion for help with static filters and triangulations; and Radu Ursu for help with the Microsoft compiler.

References

[1] D. Bandyopadhyay and J. Snoeyink. Almost-Delaunay simplices : Nearest neighbor relations for imprecise points. In *ACM-SIAM Symposium On Discrete Algorithms*, pages 403–412, 2004.

[2] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[3] C. Branden and J. Tooze. *Introduction to Protein Structure*. Garland Publishing, second edition, 1999.

[4] K. Q. Brown. *Geometric transforms for fast geometric algorithms*. PhD thesis, Carnegie–Mellon University, Pittsburgh, Penn., 1980.

[5] C. W. Carter, B. C. LeFebvre, S. Cammer, A. Tropsha, and M. H. Edgell. Four-body potentials reveal protein-specific correlations to stability changes caused by hydrophobic core mutations. *Journal of Molecular Biology*, 311(4):625–638, 2001.

[6] 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.

[7] O. Devillers and S. Pion. Efficient exact geometric predicates for delaunay triangulations. In *5th Workshop on Algorithm Engineering and Experiments (ALENEX 03)*, Baltimore, Maryland, Jan. 2003.

[8] B. Krishnamoorthy and A. Tropsha. Development of a four-body statistical pseudo-potential to discriminate native from non-native protein conformations. *Bioinformatics*, 19(12), 2003.

[9] J. Liang, H. Edelsbrunner, P. Fu, P. Sudhakar, and S. Subramaniam. Analytical shape computing of macromolecules II: identification and computation of inaccessible cavities inside proteins. *Proteins*, 33:18–29, 1998.

[10] J. Liang, H. Edelsbrunner, and C. Woodward. Anatomy of protein pockets and cavities: Measurement of binding site geometry and implications for ligand design. *Protein Science*, 7:1884–1897, 1998.

[11] R. Singh, A. Tropsha, and I. Vaisman. Delaunay tessellation of proteins. *J. Comput. Biol.*, 3:213–222, 1996.

[12] H. Wako and T. Yamato. Novel method to detect a motif of local structures in different protein conformations. *Protein Engineering*, 11:981–990, 1998.