Incremental construction of the Delaunay graph in medium dimension

Jean-Daniel Boissonnat, Olivier Devillers, Samuel Hornus

Incremental Construction of the Delaunay Triangulation and the Delaunay Graph in Medium Dimension

Jean-Daniel Boissonnat* Olivier Devillers* Samuel Hornus*

Abstract

We describe a new implementation of the well-known incremental algorithm for constructing Delaunay triangulations in any dimension. Our implementation follows the exact computing paradigm and is fully robust. Extensive comparisons show that our implementation outperforms the best currently available codes for exact convex hulls and Delaunay triangulations, compares very well to the fast non-exact `Qhull` implementation and can be used for quite big input sets in spaces of dimensions up to 6. To circumvent prohibitive memory usage, we also propose a modification of the algorithm that uses and stores only the Delaunay graph (the edges of the full triangulation). We show that a careful implementation of the modified algorithm performs only 6 to 8 times slower than the original algorithm while drastically reducing memory usage in dimension 4 or above.

1 Introduction

Very efficient and robust codes nowadays exist for constructing Delaunay triangulations in two and three dimensions [5, 24].[1] As a result, Delaunay triangulations are now used in many fields outside Computational Geometry. There exist also streaming and I/O efficient variants that allow to process huge data sets [17, 20]. The situation is less satisfactory in higher dimensions. Although a few codes exist for constructing Delaunay triangulations in dimensions higher than 3, these codes are either non robust, or very slow and space demanding, which makes them of little use in practice. This situation is partially explained by the fact that the size of the Delaunay triangulation of points grows very fast (exponentially in the worst-case) with the dimension. However, we show in this paper that a careful implementation can lead to dramatic improvement. As a consequence, our code can be used for quite big input sets in spaces of dimensions up to 6, thus enabling its use in real applications in 4-dimensional space-time or 6-dimensional phase-space.

*INRIA Sophia Antipolis – Méditerranée `http://www.inria.fr/sophia/geometrica/`

Other applications can be found in robotics where the dimension of the space reflects the number of degrees of freedom of the mechanical system, and in machine learning. See `www.qhull.org` for applications of convex hulls and Delaunay triangulations in higher dimensions.

In this paper, we propose two new C++ implementations of the well-known incremental construction. The first algorithm, recalled in Section 2.2 maintains, at each step, the complete set of d-simplices together with their adjacency graph. Two main ingredients are used to speed up the algorithm. First, we pre-sort the input points along a d-dimensional Hilbert curve to accelerate point location (Section 2.4). Second, the dimension of the embedding space is set as a C++ template parameter instantiated during the code compiling (Section 4). We thus avoid a lot of memory management. Our code is fully robust and computes the exact Delaunay triangulation of the input data set. Following the central philosophy of the CGAL library, predicates are evaluated exactly using arithmetic filters (Section 4). We have extensively compared our implementation with the best known existing codes, namely `Qhull` [2] (used by Matlab and Mathematica), `Hull` [3] (developed by K. Clarkson), the Cdd [4] library developed by K. Fukuda, and `CGAL_DT`, the current CGAL [5] implementation for higher dimensional Delaunay triangulations. Our new implementation, called New_DT in the sequel, outperforms these codes by far. This is fully reported in reference [16] and briefly reported in Section 5.1,

The main limitation of the previous algorithm is its memory usage. In fact, storing the whole triangulation becomes prohibitive as the dimensionality increases. This is seconded by theoretical results. As is well-known, the worst-case size of a Delaunay triangulation grows exponentially as a function of the number of points. Even under more favorable (and practical) situations, the dimensionality has a dramatic impact on the size of the triangulation. For instance, for points evenly distributed in a ball, although the size of the triangulation remains $O(n)$ [14], the constant hidden in the big O notation depends exponentially on d. In order to circumvent this issue, we propose a variant (which we call Del_graph) of the previous algorithm that only maintains the 1-dimensional skeleton of the Delaunay triangulation, also called Delaunay graph in the paper. Locating or inserting a new point is done by reconstructing on the fly some simplices around the new point. This reconstruction has to be done carefully since the degree of the Delaunay vertices grows exponentially with the dimensionality. In Section 5.2, we report on this approach and show that the time penalty w.r.t. the previous

[1] As an example, the CGAL 3.3.1 release, processes 20,000,000 points in 30 seconds if the points are evenly distributed in 2D and in 223 seconds if they are evenly distributed in 3D (Intel processor 2.3GHz, 16 GB RAM).

[2] http://www.qhull.org

[3] http://netlib.org/voronoi/hull.html

[4] http://www.ifor.math.ethz.ch/ fukuda/cdd_home/cdd.html

[5] http://www.cgal.org

algorithm ranges from 6 to 8 in dimensions up to 6. Although similar compact representations of 2-dimensional triangulations have been proposed in the literature [19, 3], we are not aware of similar efforts in higher dimensions.

2 New_DT and the simplex based representation

New_DT follows the implementation of the incremental construction of Delaunay triangulations in dimensions two and three that are available in CGAL [6]. It should be noticed that all the other implementations mentioned in the introduction, including `CGAL_DT`, are based on different algorithms.

2.1 Background and notations

We call $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^d$ the set of input points. We set $n = |P|$. The input points live in $\mathbb{R}^d$; we call d the *ambient* dimension. At any time, the input points to the Delaunay triangulation span a k-dimensional affine subspace of $\mathbb{R}^d$ which we call $\mathrm{Span}(P)$; we call k the *current* dimension; it satisfies $0 \leq k \leq d$ and most often $k = d$.

The *Delaunay triangulation* of P is a partition of the convex hull of P into k-simplices, each the convex hull of $k+1$ points of P and each having the so called *empty-ball* property. A simplex σ has the empty-ball property with respect to P when its circumscribing ball B_σ contains no point of P (but possibly the vertices of σ in its boundary ∂B_σ—we only consider *open* balls). We use $\mathcal{D}(P)$ to denote the Delaunay triangulation of P. When P is in general position, *i.e.,* when no $k+2$ points are co-spherical, $\mathcal{D}(P)$ is uniquely defined.

When augmenting the set P with a new point, it will be useful to define the *conflict predicate* between a point q and a simplex σ: we say that simplex σ *conflicts* with point q (or *vice versa*) if q is inside the circumscribing ball of σ. For this reason, the conflict predicate is also called the `in_sphere` predicate (we type it as a C++ function name).

A *pure complex* is a simplicial complex of which all simplices that are maximal with respect to inclusion have the same dimension. The Delaunay triangulation $\mathcal{D}(P)$ can then be stored as a pure complex. In our implementation, we extend $\mathcal{D}(P)$ to a triangulation of $\mathrm{Span}(P)$ by adding a point at infinity. Thus, we actually store a pure complex that topologically triangulates $\mathbb{S}^k$.

Here is an example: assume that P consists in n collinear points. Then $\mathcal{D}(P)$ is stored as a subdivision of $\mathbb{S}^1$ (a topological circle) into $n+1$ segments of which $n-1$ are finite and two are infinite and adjacent to the point at infinity.

In order to simplify the implementation, each simplex actually has enough

memory to describe a full d-dimensional simplex. The dimension k of $\mathrm{Span}(P)$ is stored in the `Delaunay_triangulation_d` class and used to indicate the actual dimension of the simplices.

2.2 Algorithm overview

The C++ `Delaunay_triangulation_d` class has an `insert` method that takes as input either a single point, or a range of points. If the input set contains more than one point, its points are hierarchically sorted along a space filling curve (see Section 2.4) and then inserted in that order in the Delaunay triangulation.

A. Insertion. We insert point q into the current Delaunay triangulation $\mathcal{D}(P)$:

A.1. If P is empty, we build a 0-dimensional Delaunay triangulation $\mathcal{D}(\{q\})$ with two 0-simplices; one containing the point q and the other the point at infinity. Both simplices are neighbors of each other and the `insert` procedure ends.

A.2. If the current dimension $k = \dim(\mathrm{Span}(P))$ of the triangulation is less than the ambient dimension d, we test whether $k' = \dim(\mathrm{Span}(P \cup \{q\}))$ is larger than k, in which case we proceed to a dimension jump, as detailed in step **B** below, and stop the `insert` procedure.

A.3. Finally, we have $k = k'$ (and typically, $k = d$), and we proceed to localizing a simplex in $\mathcal{D}(P)$ that conflicts with point q, as detailed in step **C** below.

B. Dimension jump. We have found in the previous step that q lies outside the affine subspace spanned by P. All k-simplices of $\mathcal{D}(P)$ are simplices of co-dimension 1 in $\mathcal{D}(P \cup \{q\})$ and $k' = k+1$. First, we "extrude" all k-simplices in $\mathcal{D}(P)$ by adding the point q to each. Second, for each finite k-simplex σ of $\mathcal{D}(P)$, we add a k'-simplex towards the point at infinity. Finally, to complete the triangulation of $\mathbb{S}^{k'}$ embedding $\mathcal{D}(P \cup \{q\})$, we update the adjacency relations between simplices. The `insert` procedure ends here.

C. Localization. To locate a simplex conflicting with q we start from some simplex σ_0 and use a *stochastic visibility walk* [13] towards q until a conflicting simplex is found. To leverage the sorting of the input points along a space filling curve, we set σ_0 to a simplex adjacent to the previous input points. In this way we guarantee that the walk to q should be short. Note, that if the stochastic walk reaches a simplex having the "point at infinity" as a vertex, then that simplex must be conflicting with q and we can stop the localization procedure. When a conflicting simplex has been found we proceed to computing and triangulating the *conflict zone*.

D. Computing and triangulating the conflict zone. The conflict zone C_q of point q with respect to set P, is simply the set of simplices in $\mathcal{D}(P)$ that conflict with q. The geometric union GC_q of the simplices in the conflict zone is simply connected and star-shaped around q. Simple connectedness

implies that C_q can be found using depth-first search. Then, the simplices of C_q are deleted and GC_q is re-triangulated by creating simplices with base the faces of the boundary of GC_q and apex q. It is an easy exercise to check that we do indeed obtain the Delaunay triangulation of $P \cup \{q\}$. The creation of the new simplices and the computation of their adjacencies is performed during a traversal of the boundary of GC_q. We travel from a face of the boundary to a neighboring one by rotating around the $(k-2)$-simplex that they share, as explained in [7].

Note that the implementation of the *conflict* predicate on a simplex and a point distinguishes two cases when the simplex is bounded or unbounded. However, the combinatorial re-triangulation of GC_q is agnostic to this distinction: no special case need be distinguished when GC_q is unbounded.

2.3 Complexity issues

Clarkson's `Hull` and `CGAL_DT` are both an implementation of the randomized optimal convex hull algorithm of Clarkson *et al.* [10]. To compute the Delaunay triangulation of the set $P \subset \mathbb{R}^d$, each point $p = (p^1, p^2, \ldots, p^d)$ in the set is lifted to the point $p' = (p^1, p^2, \ldots, p^d, ||p||^2) \in \mathbb{R}^{d+1}$. The Delaunay triangulation of P is then computed as the lower hull of the set $P' = \{p' \mid p \in P\}$, to which it is combinatorially equivalent [22, Chapter 2]. The algorithm is incremental and computes an "onion-like" layered simplicial subdivision of the convex hull, which also serves as a localization data-structure, ensuring an overall $O(n \log n)$ time complexity for the construction when $d = 2$ and $O\left(n^{\lceil \frac{d}{2} \rceil}\right)$ otherwise. To be more precise, each new layer in the "onion" is the lifted set of simplices that re-triangulate the conflict zone. Thus, once created, no simplex is ever destroyed. This implies that all the simplices existing in the triangulations of the prefix-sets of an ordering of P remain stored in the "onion" data-structure and are necessary data for the localization procedure.

In contrast, our implementation New_DT follows the same insertion procedure but makes use of no data-structure other than the current Delaunay triangulation, stored as two arrays of simplices and vertices, with adjacency information. It resorts to a straightforward walk for localizing a conflicting simplex and, when input points are provided dynamically (one after the other), the localization procedure starts from the first bounded simplex in the array of simplices: while this simplex may change when inserting new points in the triangulation, its geometrical locus will typically remain stable. Although we do not have bounds on the expected length of the localization walk when the input points are uniformly distributed, we can expect this length to decrease as the dimension increases while the input size is kept constant. This is verified experimentally [16]. Thus, by storing only the bare Delaunay triangulation, New_DT is more efficient space-wise, since much less simplices are stored, but the theoretical complexity of the localiza-

tion step becomes non-optimal. When a large set of input points is given at once, we are able to optimize the localization step by preprocessing the set of input points (see Section 2.4), in which case the measured average complexity of the localization step becomes highly independent of the number of input points, and grows mostly with the ambient dimension.

2.4 Speed-up: sorting points

Both our implementations use spatial sorting of the input points to speed-up localization. As far as we know, spatial sorting is not used in other implementations of Delaunay triangulations in high dimension. We follow the approach that Delage uses in dimensions 2 and 3, which builds upon the *Biased Randomized Insertion Ordering* (BRIO) of Amenta *et al.* [2]. The goal of the BRIO technique is to retain the theoretical optimality of a randomized incremental construction of the Delaunay triangulation, while at the same time arranging the sequence of input points so as to increase its geometric (and hopefully, in-memory) locality, thus minimizing the time spent in the localization procedure and in accessing different memory cache levels.

We process the points as follows: first, the set P of input points is partitioned into $B = O(\log |P|)$ subsets $P_i, i \in \{1, \ldots, B\}$ where P_{i+1} consists in points chosen randomly with some constant probability from $P \setminus \cup_{j=1}^{i} P_j$, so that the size of P_i is exponentially decreasing with i. The subsets are then processed from the smallest to the largest one, ordering their points along a continuous space-filling curve, and inserting them, in this order, in the Delaunay triangulation. The partitioning of P ensures a randomized "sprinkling" of the points over the domain P, while the sorting of each subset ensures locality.

This technique was successfully implemented in the CGAL library by Christophe Delage in the 2D and 3D cases [11]. He provides a direct recursive implementation for sorting the points along a Hilbert curve. In higher dimension, however, the situation becomes more difficult: it is difficult to understand and error-prone to write a program for recursively computing the axis along which one has to split the point set, so as to globally follow one continuous space filling curve [18]. Appendix A explains how we turn an existing efficient implementation of a specific space filling curve into a recursive procedure for sorting the input points "along" that curve.

3 Del_graph and the graph based representation

Although the worst-case $O\left(n^{\lceil \frac{d}{2} \rceil}\right)$ behavior does not occur in practice, keeping a full list of the Delaunay simplices and their adjacencies in memory

quickly becomes a problem as the dimension grows. For example, the 12th row of Table 1 agrees with the theoretical results of Dwyer [14] that the number of d-simplices adjacent to one vertex grows exponentially with the dimension. As a practical example, the Delaunay triangulation of 32K 6-dimensional points contains more than 32 millions simplices, each of which uses more than 14×4 bytes, for a total larger than 1.7 gigabytes. We can estimate that more than 52 gigabytes of memory would be necessary to store the Delaunay triangulation of one million 6D points. The astrophysicists Maciejewski *et al.* report that a computation of the Delaunay triangulation of roughly 1 million points in 6D took three days and the resulting billion of simplices needed 40 gigabytes of disk storage [21]. (They use their own implementation, whose algorithm we do not know.)

We propose to save space by—somewhat brutally—getting rid of the simplices and keeping only the graph of the triangulation (*i.e.,* its vertices and edges) in memory. Fully dimensional information seems however necessary to carry the computation of the Delaunay graph. Thus, we do in fact build some full-dimensional simplices when we compute the conflict zone upon insertion of a new point q. But once the new neighbors of q have been computed, and the neighbors of q's neighbors updated, we get rid of the simplices and keep only the edges between vertices. We rely on the spatial sort and a *most-recently-used* cache mechanism to ensure reasonable practical running times.

The Del_graph algorithm is very similar to New_DT, and differs most notably in the procedure used to find a neighbor of a simplex. Trivial in the previous algorithm, this procedure becomes more involved, as explained in Section 3.1.

Before delving into the details of our Delaunay graph construction algorithm, it is worth asking what we can do with it. In computing only the graph of the Delaunay triangulation one looses time but, as we shall see below, one can compute the Delaunay graph of much larger input point sets without running out of memory. The graph can be stored on disk with a much smaller memory footprint. Then, parts of the full Delaunay triangulation can be "reconstructed" on-demand. One practical scenario is inspired by reference [1]: a 4D space-time volume is reconstructed as a subset of a Delaunay triangulation, and 3D slices are extracted to visualize an animated object. In that case, only vertices adjacent to an edge crossing an hyperplane orthogonal to the *time*-axis need be considered, and only the simplices they span need be reconstructed. Another scenario is the space or phase-space density estimation algorithm DTFE [23, 21]. The density at a vertex is estimated as the inverse of the volume of the vertex's star (its set of adjacent Delaunay simplices). It is not difficult to see that our Del_graph algorithm below permits the computation of these densities along the way of computing the Delaunay graph. Finally it is possible to recover all the Delaunay simplices by a breadth-first traversal of the triangulation.

3.1 The simplex cache and the computation of a neighbor

In our implementation Del_graph, the Delaunay graph is stored in a very simple fashion: an array contains an entry for each input vertex. Each such vertex v contains

- its spatial coordinates,
- the set N^v of references to its neighbors in the Delaunay graph, and
- references to d neighbors of v forming a Delaunay simplex of $\mathcal{D}(P)$ adjacent to v.

The *simplex cache* is a doubly linked list of *cached simplices* augmented with a search data-structure, whose key for a cached-simplex is the sorted set of references to the vertices of the simplex.[6] A cached-simplex σ stores

- its key,
- $d+1$ references that are used to reference neighbors which are known to be in the simplex-cache and are *null* otherwise, and
- some flags for graph traversal and bookkeeping, *e.g.,* to avoid the computation of the conflict status of the simplex with the current input point more than once; or to cache the orientation (positive or negative) of the simplex for the current ordering of its vertices.

When a simplex is accessed in the simplex-cache, it is moved to the front of the list, in such a way that the most recently accessed simplices are grouped towards the front of the list. After inserting a new point q in the Delaunay graph, the simplex-cache is cut down to a user-chosen size `|Cache|` by successively removing cached-simplices at the back of the list. Coupled with the spatial sorting of the input points, the cache mechanism ensures that many of simplices needed to compute the conflict zone will be readily available in the simplex-cache.

In order for the insertion procedure to be correct, we enforce the following properties.

Property 3.1 *Before the insertion of point q into the Delaunay graph $\mathcal{G}(P)$, all the cached-simplices in the simplex cache are Delaunay simplices of $\mathcal{D}(P)$.*

Property 3.2 *At any time, if a cached-simplex σ has a reference to one of its neighbor σ' in the cache, then σ' also has a reference to its neighbor σ.*

[6] In our implementation, we sort the vertices according to their memory address.

The procedure for inserting a new point in the Del_graph algorithm manipulates only cached-simplices and accesses other cached-simplices[7] only through the neighbor computation which we describe below. We thus consider σ to be a cached-simplex.

Neighbor computation. Given a Delaunay d-simplex $\sigma = (a_0, a_1, \ldots, a_d)$ and an index $i \in [0..d]$, the procedure `neighbor(`σ`,`i`)` returns the unique Delaunay d-simplex that shares all vertices of σ but the vertex a_i. In other words, we are looking for the vertex a'_i such that the simplex σ', obtained by replacing a_i by a'_i in σ, is a Delaunay simplex. The procedure `neighbor(`σ`,`i`)` proceeds as follows:

1. If the i-th neighbor reference stored in σ is non-*null*, we simply return the cached-simplex that it references.

2. Otherwise, since a'_i is a neighbor of all vertices of $\sigma \setminus \{a_i\}$, we compute the set A of vertices adjacent to each vertex in $\sigma \setminus \{a_i\}$. Specifically, we define $A = \left(\bigcap_{k \in [0..d] \setminus \{i\}} N^{a_k}\right) \setminus \{a_i\}$ and call it the *set of candidate vertices*. The set A must contain a'_i but may contain other vertices. For each candidate $a \in A$ we search for the simplex $\sigma' = (a_0, a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_d)$ in the cache. If the search succeeds for some a, then we have surely found the unique neighbor σ' since the cache contains only Delaunay simplices (thanks to property 3.1). We then update the neighbor references between σ' and σ and return σ'.

3. Finally, if σ' is not in the cache, we must resort to some geometric computations to select the correct candidate in A. Consider the hyperplane H spanned by vertices $\sigma \setminus \{a_i\}$. We first eliminate from A the vertices that lie on the same side of H than a_i. Then if several vertices remain we compare them using the `in_sphere` predicate: candidate a' is "smaller" than candidate a if it lies inside the sphere spanned by vertices in $\{a\} \cup \sigma \setminus \{a_i\}$. The simplex σ' is then constructed using the "smallest" candidate and inserted into the cache; the neighbor references between σ and σ' are updated and σ' is returned.

3.2 The Del_graph algorithm

The general algorithm to insert a new point q is the same as in the simplex based representation, that is an incremental algorithm, using walking strategies for localization, determination of the conflict zone and final update of the data structure. Nevertheless, some adaptations must be done to fit the new representation of the triangulation.

Localization. The walk is done in two steps. In the first step, we start from the previously inserted point and walk along the graph as long as the current

[7] With the exception of the first accessed cached-simplex.

vertex has a neighbor closer to q. When we reach a vertex v_q that has no neighbor nearer to q, the Delaunay simplex stored in v_q is used as a key and searched in the simplex-cache, to retrieve the corresponding cached-simplex, from which the stochastic visibility walk is started, as in the New_DT algorithm.

Computing and triangulating the conflict zone. Once a conflicting simplex σ has been found, the recursive exploration of the conflict zone C_q is started using calls to the `neighbor` procedure described above. As the zone is explored, flags are used so as to compute the conflicting state of each visited simplex only once, and new adjacency relations found along the way are remembered in order to speed-up future calls to the `neighbor` procedure. Conflicting simplices are marked as *dead* and move to the back of the simplex-cache.

During this exploration, two tasks are undertaken. The first task is to build two sets of edges: the set E_c of all edges in C_q (the union of the edges of all conflicting simplices), and the set E_b of all edges of facets of the boundary of C_q (these facets are shared by a conflicting simplex and a non-conflicting one). Edges in E_b will remain edges of the graph $\mathcal{G}(P \cup \{q\})$ while edges in $E_c \setminus E_b$ will have to be deleted.

The second task is to construct the new Delaunay simplices that will triangulate C_q and place them—in advance—in the simplex-cache, because they are cheaper to compute at that time rather than later in the `neighbor` procedure. As we reach each boundary facet of C_q, we build a new simplex, as explained in Section 2.2, insert it into the cache and mark it and its non-conflicting adjacent cached-simplex as neighbors. Note that we do not attempt to compute the adjacencies of the new simplices inside the conflict zone itself, as this will be taken care of, "on demand", by the `neighbor` procedure.

After the exploration of the conflict zone, the new vertex q is added to the graph. Its set of neighbors N^q is constructed easily, as the union of the vertices of the edges in E_b (or E_c). Symmetrically, q is inserted in N^v for all $v \in N^q$, and edges in $E_c \setminus E_b$ are removed from the graph: an edge connecting vertices v and w is removed by deleting w from N^v and v from N^w.

Updating the vertices of the conflict zone. To terminate the maintenance of the graph, it just remains to update the simplex σ^v that is stored in each vertex v of the conflict zone, as the one previously stored may have been a conflicting simplex. To do so, as we compute a new simplex in C_q, we simply assign it to each of its vertices. All these new simplices share q as a vertex, so we simply choose any of them to be stored as σ^q into q's data structure.

Cleaning-up the cache. The last phase of the insertion procedure is to clean-up the simplex-cache. First, all *dead* cached-simplices (which were in conflict with q) are conveniently visited at the back of the simplex-cache and deleted from it. Then, as long as the size of the simplex-cache is larger than

the user-given `|Cache|`, we delete the oldest non-*dead* simplex (it appears at the back of the cache). Note that prior to deleting a cached-simplex σ, we must remove references to σ in its cached neighbors:[8] this is possible because property 3.2 holds. Together with the steps 2 and 3 of the `neighbor` procedure, this reference "clean-up" ensures that property 3.2 continues to hold.

Property 3.1 is now satisfied again and we are ready for the insertion of the next vertex.

4 Implementation issues

In this section, we describe technical issues that arise when implementing New_DT and Del_graph, and how we address them. More details are provided in our technical report [16].

Compile-time ambient dimension. An important flaw of the current d-dimensional CGAL "kernel" (that manages basic geometric objects, such as points and hyperplanes) is that its memory management is fully dynamic. For example the dimension of the ambient Euclidean space can be specified at run-time. This is very flexible, but is terrible for performance. We have designed a new d-dimensional kernel in which the ambient dimension is instantiated at compile-time as a C++ template-parameter. Experiments have shown a huge improvement in the speed of our implementation.

Exact geometric predicates. Another flaw of the current d-dimensional CGAL kernel is its lack of filtered computations to obtain fast and exact geometric predicates. We have followed the approach used in CGAL's 2-3-dimensional kernel, as described in [8, 15]. First, the computation of a predicate is carried using interval arithmetic. Then, only if the sign of the result can not be decided do we turn to an exact number type to recompute the value of the predicate.

List intersection. In the Del_graph algorithm, more specifically in the `neighbor(`σ`,`i`)` procedure, one needs to compute the intersection of d lists of neighbor vertices. We store references to the neighbors of each vertex as a sorted list. The naive way to intersect d such lists is to call `intersection(l1,l2)` $d-1$ times. We get faster list intersection by implementing the simple algorithm of Demaine *et al.* [12] which traverses all the lists at the same time, and makes efficient use of the logarithmic search available in each list.

[8] Not doing so may trick the `neighbor` procedure into finding a fake Delaunay simplex pointed to by an invalid reference.

5 Experimental results

We now report the salient facts about our experiments. All experiments have been performed on a 2.6 GHz Intel Core 2 Duo processor with 6 MB of level 2 cache and 4 GB of 667 MHz DDR2 RAM (Mac OS X 10.5.4). Our test programs have ben compiled using GCC 4.3.2. A sample of our time and space measurements appear in Table 2. Further experimental results on New_DT are reported in reference [16].

5.1 Experiments with New_DT

We have successfully tested New_DT in dimension 5 with up to 256K input points and in dimension 6 with up to 32K input points. Figure 1 shows the time and space used by New_DT when the input size ranges from 1 to 1024 thousands points drawn from a uniform distribution in the unit cube of $\mathbb{R}^d$.

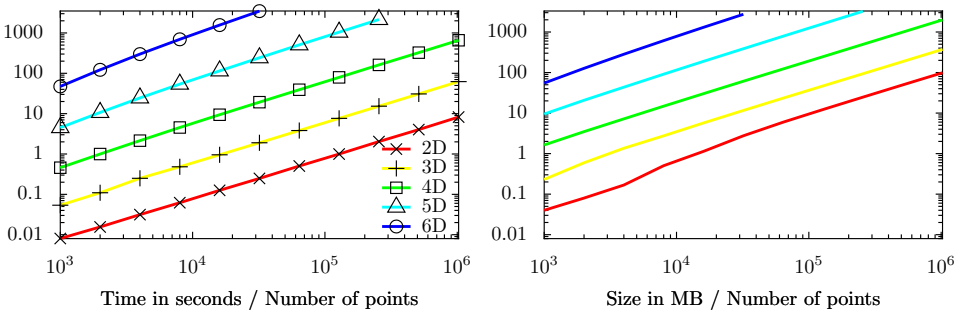

Figure 1: Timings (left) and space usage (right) of our New_DT implementation (subsection 5.1). The ambient dimension ranges from 2 to 6 and all axes are logarithmic.

In our 4D experiments, the average number of simplices adjacent to one vertex (the *simplex-valence* of a vertex) is in close agreement with the constants obtained theoretically by Dwyer for points uniformly distributed in a ball [14] (he obtains ≈ 158.9 while we measure 157 and growing). In dimension 2 to 6, we do observe that the average simplex-valence is indeed growing exponentially with the dimension (see line 12 of Table 1).

These quickly raising valences are in sharp contrast with the small number of simplices *visited* during the localization procedure. For example, in our 6-dimensional experiments, no more than 28 simplices on average are traversed during the localization phase of the insertion procedure, please refer to [16] for more details. We conclude that, as soon as we reach dimension 4, the cost of computing and triangulating the conflict zone is far more important than the cost of localizing a first conflicting simplex, when the points are uniformly distributed.

1	Dimension	2	3	4	5	6
2	Number of input points	1024K	1024K	1024K	256K	32K
3	Size of the simplex-cache	1K	1K	10K	300K	1000K
4	Size of the conflict zone	4.1	21	134	940	6145
5	Calls to `neighbor`	12.2	84.6	671.2	5631	43021
6	Number of candidates	2	2.6	4	6.7	11.6
7	Fast cache hit (non-*null* pointer)	56.6 %	57.5 %	54.6 %	55.5 %	54.3 %
8	Cache hit	37 %	39.6 %	40.1 %	42.3 %	43.1 %
9	Cache miss	6.4 %	2.9 %	5.3 %	2.2 %	2.6 %
10	Time ratio (Del_graph/New_DT)	6.1	5.7	6.0	6.5	8.1
11	Space ratio (Del_graph/New_DT)	2.7	1.7	0.6	0.2	0.1
12	Number of simplices per vertex	6	$27_{(\times 4.5)}$	$157_{(\times 5.8)}$	$1043_{(\times 6.7)}$	$7111_{(\times 6.8)}$
13	Number of edges per vertex	6	$15.5_{(\times 2.6)}$	$36.5_{(\times 2.4)}$	$73_{(\times 2)}$	$164.6_{(\times 2.25)}$

Table 1: Statistics for Del_graph. In rows 12 and 13, the parenthesized multiplicand is the ratio of the current column value with the previous one.

Spatial sorting is used to reduce the number of simplices visited during the localization part of the insertion. We find that it is extremely efficient in 2D and 3D, somewhat efficient in 4D and 5D and that its efficiency is unclear in 6D, as we didn't see much improvement due to spatial sorting in 6D. We do believe however that such an improvement should be witnessed if one sharply increases the input size.

We have compared New_DT with several other implementations presented in the introduction. As a rough summary, we find that our implementation performs much better than all the exact implementations (time-wise and memory-wise) and compares very well to the fast non-exact `Qhull` implementation.[9] For these comparisons and for a lengthier analysis of the effect of spatial sorting, we again refer the interested reader to [16] and proceed now to experiment with our implementation of the Delaunay graph, Del_graph.

5.2 Experiments with Del_graph

We have experimented with Del_graph on input points uniformly distributed in a cube. We sum up some of the statistics that we obtain in Table 1.

Each column of Table 1 corresponds to a single run of our implementation in a different dimension, as shown in the first line. The second line displays the number of input points (drawn at random from a uniform distribution in a cube) and the third line indicates the size of the simplex-cache that we chose. In dimension 5 and 6, we believe that choosing a smaller cache size

[9] In dimensions 2, 3 and 4, our implementation is faster that `Qhull` when the number of (uniformly distributed) points exceeds 100K.

should not hamper the timings too much.

Line 4 shows, in line with the measurements of line 12, how quickly the average size of the conflict zone grows with the dimension.

Line 5 shows the average number of calls to the `neighbor` procedure during each exploration of the conflict zone. These should be compared with the less than 30 visited simplices in the localization procedure in 6D. Line 6 shows the average number of candidate vertices for the completion of the neighboring simplex: this is the average size of the intersection of the neighbor-lists of d vertices forming a Delaunay $(d-1)$-simplex.

Line 7, 8 and 9 show, respectively as a percentage of line 5, the number of times a reference to a neighbor was readily available in a cached-simplex, the number of times the neighbor was present in the cache but had to be searched from the list of candidates, and the number of times the neighbor was not in the cache and had to be computed by sorting the candidates with the `in_sphere` predicate. It is expected that the "fast cache hit" (line 7) stays above 50 % percent since a simplex-simplex adjacency is most often visited twice (in both directions), and at the second visit, we are guaranteed to get a "fast cache hit". The low percentages of line 9, however, suggest that our cache mechanism (together with the spatial sorting) is quite effective.

Line 10 of Table 1 displays the ratio of the time taken by Del_graph to the time taken by New_DT to complete their respective computation. Considering the amount of additional work that has to be done in the `neighbor` procedure, we are quite pleased with these ratios. And we further believe that further optimization should reduce them even more.

For example, we have found experimentally that storing the lists of neighbors in sorted arrays (C++ `std::vector`) decreases the running time of Del_graph by 5 to 10 %, when compared to using the tree-structured `std::set`.

Line 11 is the reason why we wanted to implement the Delaunay graph in the first place: to reduce memory usage. On the one hand, the results that we obtain are convincing in dimension 5 and 6; In dimension 6, one is able to divide space usage by a factor of 10. Nevertheless, the estimates that we get in the next section show that we should be able to get much lower memory usage. We currently attribute the current state of affairs to the non adapted memory management of the C++ library for our purpose (storing millions of somewhat short lists of references to the vertices' neighbors).

Lines 12 and 13 show respectively the average number of d-simplices and edges adjacent to each vertex. They provide an experimental evidence for the geometric growth of the number of these adjacencies.

5.3 Memory usage

We fix a set P of n points in $\mathbb{R}^d$ and examine the memory required to store $\mathcal{D}(P)$ and $\mathcal{G}(P)$, the two data-structures described above. We assume that

pointers occupy one word of memory, while numbers (coordinates) occupy two words.

Memory usage for storing the full triangulation. The full triangulation consists of a list of vertices and a list of d-simplices. The vertices are stored in a simple list and each vertex additionally stores a pointer to an adjacent simplex. The total memory needed for storing vertices is then $n(2d+1)$. Each simplex record contains a state flag, $d+1$ pointers to its vertices, and $d+1$ pointers to its neighboring simplices. Let $\texttt{Simp}(P)$ denote the number of simplices in $\mathcal{D}(P)$. The storage of the simplices then requires $(2d+3)\,\texttt{Simp}(P)$ and the total memory usage is $\boxed{n(2d+1)+(2d+3)\,\texttt{Simp}(P)}$.

Our implementation stores another pointer in each vertex for the sake of memory management. For speed, each simplex also stores the $d+1$ integer indexes of the mirror vertex of its vertices.[10], thus totaling $n(2d+2)+(3d+4)\,\texttt{Simp}(P)$ words of memory.

Memory usage for storing the Delaunay graph. From the description of the simplex-cache above, we can deduce that the total memory usage for Del_graph algorithm is
$\boxed{n(3d+1)+2\,\texttt{Edge}(P)+(2d+5)|\texttt{Cache}|}$, where |Cache| is the maximum allowed number of cached-simplices.

In practice though, additional memory is necessary for efficient lookup into the cache (we use a hash table) and for the internal representation of the std::sets that we use to store the neighbors of each vertex. (The gcc libstdc++ implementation uses red-black trees.)

		CGAL_DT	Cddf+	Hull	Qhull	New_DT	Del_graph
5D, 2K points	time(s)	1034	13494	17	3.2	10.7	58
	space(MB)	338	n.a.	245	52	20.7	10.1
5D, 32K points	time(s)	swap	slow	527.4	76	242.5	1463.46
	space(MB)	swap	n.a.	844	973	386.6	106
6D, 32K points	time(s)	swap	slow	swap	swap	3472.54	28296
	space(MB)	swap	n.a.	swap	swap	2736	267

Table 2: A sample of our measurements. The word "swap" indicates that the process ran out of random access memory. The word "slow" indicates an *estimated* running time of several weeks or more.

Predictions for uniformly distributed points. In 6D, for 32K uniformly distributed points, we measured that the number of simplices in $\mathcal{D}(P)$ is approximately 12.2 times higher than the number of edges and that each vertex has about 162 neighbors. We deduce that storing the full triangulation of 32K 6D uniformly distributed points takes roughly $15\,109 \times n$ words com-

[10] The mirror vertex v' of a vertex $v \in \sigma$ (σ being a simplex) is the unique vertex of the unique simplex σ' such that $\sigma \cup \sigma' = (\sigma \cap \sigma') \cup \{v\} \cup \{v'\}$.

pared to $183 \times n$ words for storing their Delaunay graph (if we don't take into account the memory used by the cache). This makes a ratio of 82.3 that our implementation Del_graph is currently a little far from (we obtain a ratio of 10). Similar reasoning leads us to the following memory-usage ratio in dimensions 2 to 5: 1.46 in 2D, 2.65 in 3D, 7 in 4D and 22.8 in 5D.

6 Conclusions

We have presented an implementation of the well-known incremental algorithm for constructing Delaunay triangulations in any dimension, New_DT, as well as a variant implementation that uses a simplex-cache and keeps only the graph of the triangulation: Del_graph. The New_DT code is fully robust and outperforms the existing implementations. We believe that New_DT can be used in real applications in spaces of dimensions up to 6. We are currently working on applications for meshing in 6D phase-space and reconstructing dynamic scenes in 3D-space-time. Results will be reported in forthcoming papers. We plan to further experiment using our implementation for computing triangulations in even higher dimensions (7, 8, ...). In these dimensions, the simple filtering mechanism that we use for evaluating the geometric predicates may not be sufficiently efficient, and revert too often to exact computations. Solutions to this problem exist [8] but have to be implemented into our d-dimensional kernel. Concerning Del_graph, several possibilities of improvement will be studied in the future: first the effect of the size of the simplex-cache has not been studied in details and its effect on the running-time should be optimized. Second, we want to experiment with keeping the simplex circumsphere in each cached-simplex so as to speed up the `in_sphere` predicate. Third, the geometric locality of consecutive input points (obtained via the spatial sort) could be used to produce a compact representation of vertex references in a way similar to Blandford *et al.* [4].

Two theoretical questions arise from our experimental results. First, we would like to obtain bounds on the number of neighbors of a vertex when the input points are nicely distributed. This number raises exponentially with the dimension; can we get a more precise formulation? Second, we were surprised by the very slowly growing size of the set of candidates when looking for a simplex neighbor (see Section 3.1 and line 6 of Table 1). Can we obtain a precise bound on this quantity?

Acknowledgments

We owe a lot to Sylvain Pion for help in C++ programming and the inner workings of CGAL. We thank Mariette Yvinec for helpful discussions, and Kenneth Clarkson for providing us with his current `Hull` code. We are grateful to the reviewers, for they have abounded in comments and suggestions.

This research has been partially supported by the ANR project GAIA (Géométrie Informationnelle et Applications) n° BLAN-2_200008.

References

[1] E. Aganj, J.-P. Pons, F. Ségonne, and R. Keriven. Spatio-temporal shape from silhouette using four dimensional Delaunay meshing. In *IEEE International Conference on Computer Vision*, 2007.

[2] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proceedings of the 19th annual symposium on Computational Geometry*, pages 211–219, New York, NY, USA, 2003. ACM.

[3] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow. Compact representations of simplicial meshes in two and three dimensions. In *Proc. 12th International Meshing Roundtable*, pages 135–146, 2003.

[4] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 679–688, 2003.

[5] C. E. Board. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[6] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[7] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. *Theoret. Comput. Sci.*, 112:339–354, 1993.

[8] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109(1-2):25–47, Apr. 2001. A preliminary version of this paper appeared in the 14th annual ACM symposium on Computational Geometry, Minneapolis, June 1998.

[9] A. R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Trans. Comput.*, 20(4):424–426, 1971.

[10] K. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Computational Geometry: Theory and Applications*, 3(4):185–212, Sept. 1993.

[11] C. Delage. Spatial sorting. In C. E. Board, editor, *CGAL User and Reference Manual*. www.cgal.org, 3.3 edition, 2007.

[12] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 743–752, 2000.

[13] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

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

[15] M. Hemmer, S. Hert, L. Kettner, S. Pion, and S. Schirra. Number types. In C. E. Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[16] S. Hornus and J.-D. Boissonnat. Efficient construction of the delaunay triangulation in medium dimension. Research Report 6743, INRIA, 2008.

[17] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, 2006.

[18] G. Jin and J. Mellor-Crummey. SFCGen: A framework for efficient generation of multi-dimensional space-filling curves by recursion. *ACM Trans. Math. Softw.*, 31(1):120–148, 2005.

[19] M. Kallmann and D. Thalmann. Star-vertices: A compact representation for planar meshes with adjacency information. *Journal of Graphics Tools*, 6:7–18, 2001.

[20] P. Kumar and E. A. Ramos. I/O-efficient construction of Voronoi diagrams. Technical report, Max Planck Institute Informatik, Saarbrücken, 2002.

[21] M. Maciejewski, S. Colombi, C. Alard, F. Bouchet, and C. Pichon. Phase-space structures I: A comparison of 6D density estimators. http://arxiv.org/abs/0810.0504v1, Oct. 2008.

[22] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[23] W. E. Schaap. *DTFE: the Delaunay Tessellation Field Estimator*. PhD thesis, University of Groningen, 2007. `http://irs.ub.rug.nl/ppn/298831376`.

[24] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*. Association for Computing Machinery, May 1996.

A More on sorting points

A continuous space-filling curve $\mathcal{C}$ usually covers the unit cube $[0,1]^d \subset \mathbb{R}^d$ and is typically described as the limit of a recursive process wherein one level $l \in \mathbb{N}$ of the recursion tree defines a piecewise linear (polyline) approximation $\mathcal{C}^l$ of $\mathcal{C}$, with vertices regularly spaced along $\mathcal{C}$. Such a polyline at level l can be described by an ordering O_l of the points

$$\{2^{-l}(x^1, x^2, \ldots, x^d) \mid \forall i \in [1..d], x^i \in [0..2^l)\}$$

such that the l^1 or l^2 distance between two successive points in O_l is precisely 2^{-l} (a property ensuring that the limit curve $\mathcal{C}^i$s continuous).

For our purpose we use an optimized implementation of an any-dimensional space filling curve written by D. Moore.[11] The algorithm is due to Butz [9]. In particular, we use as a black box the function `i2c` [12] that returns the n-th points in $\mathbb{R}^d$ along the curve $\mathcal{C}^l$. The parameters of the function `i2c` are

- d the dimension of the ambient space,
- l the level of approximation of the curve and
- n the index of the point in O_l. We have $0 \leq n < 2^{dl}$.

If $|P|$ is the number of points to be sorted along the space filling curve, we select the level l such that $|P| \leq 2^{dl}$. Our procedure `hilbert_sort_d` for sorting the points take two ranges as parameters. The first range is the set P of input points to be sorted. The second range is an interval $[\texttt{idx}, \texttt{idx} + \texttt{width})$ included in $[0..2^{dl})$, such that `width` is a power of two, and the inequality $|P| \leq \texttt{width}$ holds. The parameter l is also passed as a constant parameter to `hilbert_sort_d`, and the initial call to the function is `hilbert_sort_d`$(P, 0, 2^{dl})$.

`hilbert_sort_d`$(P, \texttt{idx}, \texttt{width})$ performs the five following steps:

1. If $|P| < 2$ then stop.

2. Compute $p_a = \texttt{i2c}(d, l, \texttt{idx} + \texttt{width}/2 - 1)$
   Compute $p_b = \texttt{i2c}(d, l, \texttt{idx} + \texttt{width}/2)$

3. Compute $axis \in [1..d]$ as the unique axis of $\mathbb{R}^d$ along which the coordinates of p_a and p_b differ. Use the sign of the difference of coordinates to orient the $axis$.

4. Rearrange the array of points P in such a way that the points in the first half P_1 of P have their $axis$-th coordinate lower [13] than the points in the second half P_2. We use `std::nth_element` for that purpose.

[11] Hopefully, you may find Doug Moore's notes and implementation here: `http://web.archive.org/web/*/www.caam.rice.edu/~dougm/twiddle/`

[12] `i2c` stands for "index to coordinates".

[13] Recall that the $axis$ has been oriented in the previous step.

5. Recursively call `hilbert_sort_d`$(P_1, \texttt{idx}, \texttt{width}/2)$ and `hilbert_sort_d`$(P_2, \texttt{idx}+\texttt{width}/2, \texttt{width}/2)$.

The crucial element for the correctness of our procedure is the following property of the `i2c` function.

For all $\texttt{width} = 2^k, k \in [1..dl)$, for all $\texttt{idx} = \texttt{width} * r, r \in [0..2^{dl-k})$, let *axis* be as computed in step (4) above. Let $P_a = \{\texttt{i2c}(d, l, \texttt{idx} + n) | n \in [0..\texttt{width}/2)\}$ and $P_b = \{\texttt{i2c}(d, l, \texttt{idx} + n) | n \in [\texttt{width}/2..\texttt{width})\}$. Then the sets P_a and P_b are separated by a hyperplane orthogonal the the *axis*-th axis. We omit the simple proof of this property in this extended abstract.

This property ensures that the partitioning of P along the *axis*-th axis in step (4) is consistent with the "convolutions" of the space-filling curve $\mathcal{C}$: we, in effect, mimic the curve by recursively splitting our input points along the computed *axis*-th axes.

Note how, although $\mathcal{C}$ only covers the unit cube $[0, 1]^d$, we are able to sort any point set by leveraging this property of the curve `i2c`. Indeed, `hilbert_sort_d` does not need to apply scaling and translation to the input points prior to the sorting. Only coordinates comparisons are needed. This permits the use of our sorting procedure to many different number types (*e.g.,* `int`egers or more complex exact algebraic types). Further, the number of coordinates comparisons necessary to sort n points remain the same, whatever the dimension of the ambient space (assuming a fixed average complexity for the `std::nth_element` function).