An efficient implementation of Delaunay triangulations in medium dimensions

Samuel Hornus, Jean-Daniel Boissonnat

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An efficient implementation of Delaunay triangulations in medium dimensions

Samuel Hornus — Jean-Daniel Boissonnat

N° 6743

Novembre 2008

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6743--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

An efficient implementation of Delaunay triangulations in medium dimensions

Samuel Hornus, Jean-Daniel Boissonnat

Thème SYM — Systèmes symboliques
Équipes-Projets Géométrica

Rapport de recherche n° 6743 — Novembre 2008 — 15 pages

Abstract: We propose a new C++ implementation of the well-known incremental algorithm for the construction of Delaunay triangulations in any dimension. Our implementation follows the exact computing paradigm and is fully robust. Extensive comparisons have shown that our implementation outperforms the best currently available codes for convex hulls and Delaunay triangulations, and that it can be used for quite big input sets in spaces of dimensions up to 6.

Key-words: geometry, triangulation, Delaunay, implementation, C++

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Une implémentation efficace de la triangulation de Delaunay en dimensions moyennes

Résumé : Nous prśentons une nouvelle impléntation en C++ de l'algorithme incrémental randomisé de construction de la triangulation de Delaunay dans n'importe quelle dimension. Notre implémentatation utilise des calculs numériques exacts, et est ainsi robuste. Nous effectuons de nombreuses comparaisons qui montrent que notre programme se comporte bien ieux que les programmes existant pour le calcul d'enveloppe convexe ou de triangulation de Delaunay. Nous montrons que notre programme peut-être utilisé pour un grand nombre de points d'entrée dans en dimension jusqu'à 6.

Mots-clés : géométrie, triangulation, Delaunay, implémentation, C++

1 Introduction

Very efficient and robust codes nowadays exist for constructing Delaunay triangulations in two and three dimensions [4, 17]. 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 [13, 15]. The situation is less satisfactory in higher dimensions. Although a few codes exist for constructing Delaunay triangulations in higher dimensions, these codes are either non robust, or very slow and space demanding, which make 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. This make our package useful for real applications in 4-dimensional space-time or 6-dimensional phase-space. 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 a new C++ implementation of the well-known incremental construction. The algorithm, recalled in section 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 (section 3). First, we pre-sort the input points along a d-dimensional Hilbert curve to accelerate point location. In addition, the dimension of the embedding space is a C++ template parameter instanciated at compile time. 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). In section 5, we extensively compare our implementation with the best known existing codes, namely `Qhull` [1] (used by Matlab and Mathematica), `Hull` [2] (developped by K. Clarkson), the Cdd [3] library developped by Fukuda, and `CGAL_DT`, the current CGAL [4] implementation for higher dimensional Delaunay triangulations. Our new implementation, called `New_DT` in the sequel, outperforms these codes by far. We intend to submit `New_DT` as a new CGAL package.

2 The algorithm used in the new implementation

`New_DT` follows the implementation of the incremental construction of Delaunay triangulations in dimensions two and three that are available in CGAL. It should be noticed that all the other algorithms mentionned 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.

The *Delaunay triangulation* of P is a partition of the convex hull of P into $O(n^{\lceil \frac{d}{2} \rceil})$ d-simplices, each the convex hull of $d+1$ points of P and each having the so called *empty-ball* property. We use $\mathcal{D}(P)$ to denote the Delaunay triangulation of P. A simplex σ has the empty-ball property (with respect to P) when its circumscribing ball contains no point of P but possibly the vertices of σ. When P is in general position, i.e., when no $d+2$ 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.

A *regular complex* is a simplicial complex of which all maximal simplices have the same dimension. The Delaunay triangulation $\mathcal{D}(P)$ can then be stored as a regular complex. In our

[1] http://www.qhull.org
[2] http://netlib.org/voronoi/hull.html
[3] http://www.ifor.math.ethz.ch/ fukuda/cdd_home/cdd.html
[4] http://www.cgal.org

implementation, we extend $\mathcal{D}(P)$ to a triangulation of $\mathrm{Span}(P)$ by adding a point at infinity. Thus, we actually store a regular complex that topologically triangulates $\mathbb{S}^k$.

Here is an example: Assume that P consists in n colinear points. Then $\mathcal{D}(P)$ is stored as a subdivion 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 `CGAL::Delaunay_triangulation_d` class and used to indicate the actual dimension of the simplices.

2.2 Algorithm overview

The `Delaunay_triangulation_d` class has a `insert()` method that takes as input either a single point, or a range of points. If the set of input points has more than one point, its points are hierarchically sorted along a space filling curve (see section 3) and then inserted in that order in the Delaunay triangulation.

2.3 Insertion

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

Case a. 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.

Case b. If the current dimension $k = \dim(\mathrm{Span}(P))$ of the triangulation is less than the ambient dimension d, we test wether $k' = \dim(\mathrm{Span}(P \cup \{q\}))$ is larger that k, in which case we proceed to a dimension jump, as detailed in subsection 2.4 below, and stop the `insert()` procedure.

Case c. 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 subsection 2.5 below.

2.4 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 $\mathcal{D}(P \cup \{q\})$. First, we "extrude" all k-simplices if $\mathcal{D}(P)$ by adding the point q to each. Second, for each finite k-simplex σ of $\mathcal{D}(P)$, we add a $k+1$-simplex towards the point at infinity. Special care has to be taken when linking the new simplices to their neighbors. This completes the triangulation of $\mathbb{S}^{k+1}$ embedding $\mathcal{D}(P \cup \{q\})$ and the `insert()` procedure ends.

2.5 Localization

To locate a simplex conflicting with q we start from some simplex σ_0 and use a *stochastic walk* 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 garantee 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 is 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.

2.6 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. 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 by rotating around the $d-2$-simplices in the boundary of GC_q, as explained in [5].

Note that the implementation of the *conflict* predicate between 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.

2.7 Complexity issues

Clarkson's `Hull` and `CGAL_DT` are both based on the randomized optimal convex hull algorithm of Clarkson *et al.* [9]. To compute the Delaunay triangulation of the set $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^d$, each point $p_i = (p_i^1, p_i^2, \ldots, p_i^d)$ in the set is lifted to the point $p'_i = (p_i^1, p_i^2, \ldots, p_i^d, ||p_i||^2) \in \mathbb{R}^{d+1}$. The Delaunay triangulation of P is then computed as the lower enveloppe of the convex hull of the set $P' = \{p'_1, p'_2, \ldots, p'_n\}$, to which it is combinatorially equivalent [16, Chapter 2]. The algorithm is incremental and computes an "onion-like" 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 $d \leq 3$ and $O(n^{\lceil d/2 \rceil})$ otherwise. Once created, no simplex is ever destroyed. This implies that all the simplices existing in the triangulations of the collection of sets $\left\{P_j = \{p_1, p_2, \ldots, p_j\} | j \in [1, n]\right\}$ remain stored in the "onion" data-structure and are necessary data for the localization procedure.

In contrast, our implementation `New_DT` makes use of no data-structure other than the current Delaunay triangulation, stored as two arrays of simplices and vertices, with adjacency information. 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. It is expected, therefore, that the asymptotic complexity of the localization procedures be $O(\sqrt[d]{n})$ when input points are drawn from a uniform distribution. 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 localization step becomes non-optimal. When a large set of input points is given at once, we are able to compensate for by preprocessing the input points (see subsection 5.2). Experimentally, the expected complexity of the localization step becomes highly independent of the number of input points, and grows mostly with the ambient dimension.

3 Two important speedups

We describe below two "optimizations" that `New_DT` implements, but `CGAL_DT` doesn't. First we pre-process the input points so as to increase geometric locality while preserving some amount of randomness. Second, we describe how the ambient dimension d is passed as a *compile-time* parameter, which gives more freedom to the compiler for code optimization, and enables more efficient memory management.

3.1 Sorting points

We follow the approach that Delage used in dimension 2 and 3, which is inspired by the *Biased Randomized Insertion Ordering* (BRIO) of Amenta *et al.* [2].

The authors of the later reference explain how one can 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. The set P of input points is partitioned into $B = O(\log |P|)$ subsets $P_1, P_2, \ldots, P_B$ where P_{i+1} consists in points chosen randomly with some constant probability from $P \setminus \cup_{j=1}^{i} P_j$. The subsets are then processed from the smallest to the largest one, ordering their points in some fashion (e.g., using an octree), 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 succesfully implemented in the CGAL library by Christophe Delage in the 2D and 3D cases [10]: To further improve locality, each subset P_i is sorted so as to mimick sorting along a continuous space-filling curve. Delage provides a direct recursive implementation.

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 [14]. In appendix A, we explain how we turn an existing efficient implementation of a specific space filling curve into a recursive procedure for sorting of the input points "along" the curve.

3.2 Specifying the ambient dimension at compile-time

In `CGAL_DT`, most of the execution time is spent in memory management in the d-dimensional CGAL geometric kernel, which provides types such as `Vector_d` and `Matrix_d` as *fully dynamic* arrays of some chosen `Number_type`.

Such a fully dynamic approach allows, for example, to specify the dimension of the ambient space at run-time, but the cost of this freedom is prohibitive when one is concerned with execution time. This is illustrated in Section 5. Another motivation for a dynamic approach of memory management is that, even when the ambient dimension d is known, the dimension of the geometric object under construction may vary. In our case, the Delaunay triangulation may span an affine subspace of dimension k lower than d, yielding geometric predicates with varying number of arguments. Many geometric predicates involve the computation of matrix determinants. There again, the size of the matrix may vary from one geometric predicate to another; for example, let H be a hyperplane of $\mathbb{R}^d$ defined as the affine subspace spanned by d points. Deciding on which side of H lies a query point, boils down to the computation of a $d \times d$ matrix determinant. On the other hand, if σ is a d-simplex in $\mathbb{R}^d$, then deciding if a query point is in conflict with σ requires computing the sign of a $(d+1) \times (d+1)$ matrix determinant.

We now describe the modifications we made to the d-dimensional CGAL geometric kernel to accomodate a compile-time dimension parameter. We choose to use this latter parameter, denoted as `DIM`, as indicating the ambient dimension of the Euclidean space. As such, the kernel class `Cartesian_d<DIM, NT>` [5] defines classes representing points, vectors, simplices, *etc.*... living in a `DIM`-dimensional Euclidean space. `DIM` being know at compile-time, no dynamic memory management is necessary when instances of these geometric objects are created. In our implementation, for example, we use a `std::array<DIM, NT>` to store the coordinates of a vector.

Further, the C++ class `Cartesian_d<DIM, NT>` provides easy access to other ambient dimensions, so that, for example, `Cartesian_d<DIM, NT>::rebind_dimension<DPRIM>::other` is the same as `Cartesian_d<DPRIM, NT>`, where `DPRIM` is any expression whose constant value can be computed by the compiler.

For convenience, and to avoid rewriting too much preexisting code, the matrix class template is parameterized only by the number of its rows [6]. The number of columns is dynamic and each column is stored as a vector.

4 Filtering the predicates

Our implementation is meant, when desired, for the exact computation of the Delaunay triangulation of a set of points. On the one hand, no geometric construction is necessary to represent the Delaunay triangulation in memory, as the triangulation is a purely combinatorial structure added to the input points. In this regard, exact construction is possible. On the other hand, however, various geometric predicates are necessary for the correct deduction of the combinatorial structure of the Delaunay triangulation. These typically take the form of determining the sign (positive, negative or null) of a matrix determinant, whose entries are input points' coordinates.

To achieve exact predicates, we follow [6]: A first evaluation of the value of the polynomial is computed using interval arithmetic. A C++ exception is raised if a bad arithmetic operation is executed (e.g., dividing by an interval containing 0) or if the final interval contains 0, preventing the correct sign determination.

When catching such an exception, the computation of the geometric predicate is restarted using an exact but slower numeric type [12]. In practice, when the ambient dimension is lower than 7,

[5] `NT` is a template parameter used to define the number type for representing coordinates.
[6] And, of course, by the type of numbers to be stored.

the predicates used for the Delaunay triangulation rarely fail during the interval arithmetic stage (see [6]).

5 Experiments

We have conducted several experiments to test the efficiency of our new implementation `New_DT` in computing the Delaunay triangulation of relatively large set of points. In this section, we report on the general performance of `New_DT`, examine the effectiveness of the optimizations we brought in, and compare `New_DT` with other programs capable of computing Delaunay triangulations in high dimension: `Qhull`, `Hull`, `Cddf+` and `CGAL_DT`.

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.

`Qhull` has been compiled on the same machine, but with the system's GCC 4.0.1 (through the `fink` package system), as we were not able to compile a working binary with GCC 4.3.2.

`Hull` and `Cddf+` have been compiled—like our program—with GCC 4.3.2 and the compiler options `-DNDEBUG -O3 -mtune=core2`.

5.1 The case of a uniform distribution

For the general benchmark of `New_DT`'s speed and memory usage, we have run our program on input sets consisting of uniformly distributed random points in a unit cube with floating point (`double`) input coordinate type. In each case, the input points are provided at once, which permits the use of spatial sorting prior to inserting the points in the triangulation. Figure 1 shows the time and space used by `New_DT` when the input size ranges from 1 to 1024 thousands.

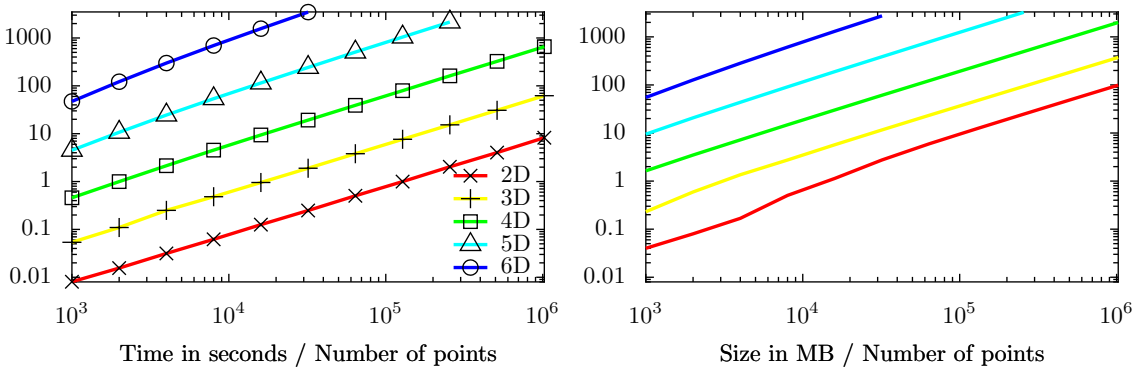

Figure 1: Timings (left) and space usage (right) of our implementation (subsection 5.1). The ambient dimension ranges from 2 to 6 and all axes are logarithmic.

Dimension d	2	3	4	5	6
k	$78 * 10^{-7}$	$53 * 10^{-6}$ ($\times 6.8$)	$37 * 10^{-5}$ ($\times 7$)	$24 * 10^{-4}$ ($\times 6.5$)	$99 * 10^{-4}$ ($\times 4.1$)
α	1.0011	1.011	1.0443	1.1059	1.2360

Table 1: The table shows the result of fitting a function $\text{time}_d(n) = kn^\alpha$ to the timing data of Figure 1.

We have used Maple to fit lines to our plotted data. We thus obtain an numerical approximations of the time and space complexity of our algorithm. These are reported in table 1. As expected, the constant k increases with the dimension. Further, the exponent α is very close to one in dimension two and regularly increases and we can observe a jump in the exponant from dimension five to dimension six.

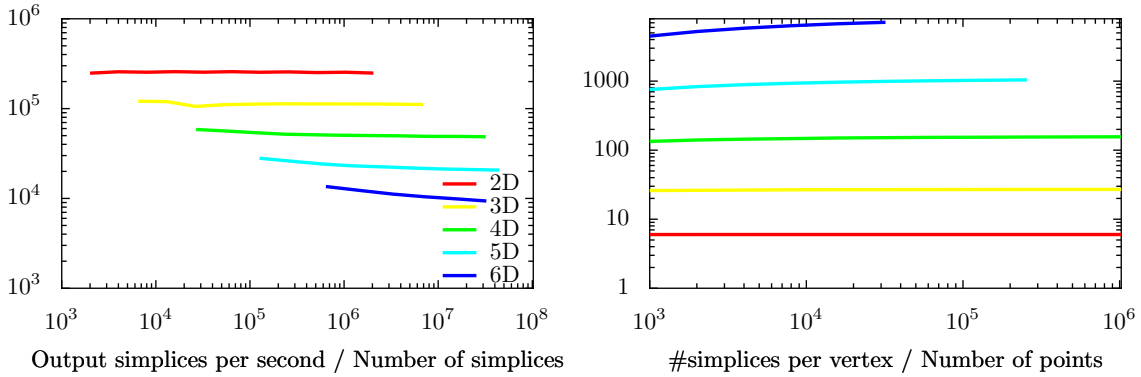

Figure 2: All axes are logarithmic.

Other interesting data gathered during our experiments are displayed on Figure 2:

Let a *final simplex* be a simplex of the Delaunay triangulation of the input points. (A non-final simplex was part of the triangulation but has been destroyed because it conflicted with some later-inserter point.) On the left, we display the "simplex flow", ie, the number of final simplices divided by the time taken to build the Delaunay triangulation, as a function of the number of final simplices. The plots indicates that the simplex flow decreases extremely slowly as the size of the triangulation increases.

The right graph plots the average number of simplices adjacent to one vertex. We do find the well known 6 and 27 simplices per vertex in dimension 2 and 3. We find it surprising, however, the rate at which this valence increases with the dimension. One vertex is, on average, adjacent to roughly 157 simplices in dimension 4, more than 1050 in dimension 5 and probably much more than 7200 in dimension 6. Remember that this valence number is roughly the number of simplices on which the *conflict* predicate is evaluated when inserting a new point. Further, the conflict predicate is the most costly predicate to evaluate since is boils down to the evaluation of the sign of the determinant of a $(d+1) \times (d+1)$ matrix.

These quickly raising valences are in sharp contrast with the small number of simplices visited during the localization procedure called when inserting one point in the Delaunay triangulation. We elaborate in the following subsection.

We conclude that, as soon as we reach dimension 4, and for uniformly distributed input points, the cost of computing and re-triangulating the conflict zone is far more important than the cost of localizing a first conflicting simplex. We also note that the valence of ≈ 157 we obtain in dimension 4 is in complete accordance with the constant obtained theoretically by Dwyer for uniformly distributed points in a ball [11] (he obtains ≈ 158.9, but note that our plot hasn't yet reached a flat state for dimensions 4, 5 and 6).

5.2 Profit (and loss) from spatial sorting

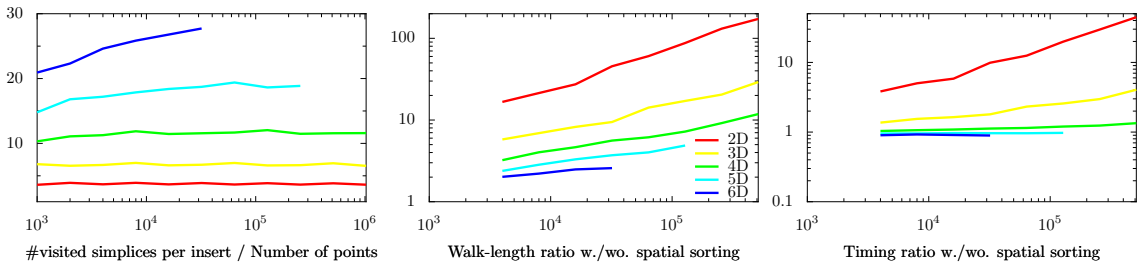

Figure 3: Efficiency of spatial sorting. All axes are logarithmic except for the left vertical axis (*#visited simplices per insert*). See subsection 5.2.

The spatial sorting of the input points has been designed so as to minimize the time spent in the localization procedure. We first note that the time taken to spatially sort the input points is negligible: Sorting one million points with double coordinates takes roughly 2 seconds in dimension 2 and seconds in dimension 6. To check wether our spatial sorting is effective or not, we plot three sets of measures in the graphs of Figure 3.

On the left is shown the average number of simplices visited *during* the localization procedure, as a function $V_{\text{sorted}}(d, n)$ of the ambient dimension and the number of input points. Although the steady state seems not to have been reached for dimension 6 (with 32000 input points), we can observe the remarkably constant behavior of the curves. In other words, spatial sorting makes the cost of the localization procedure depend only on the ambient dimension, and no more on the number of input points. We further support this proposition by showing, in the middle of Figure 3, the ratio $\frac{V_{\text{shuffled}}(d,n)}{V_{\text{sorted}}(d,n)}$, where, in the numerator, the input points has been shuffled but not spatially sorted. As expected, the average number of visited simplices are significantly higher when the input points are not spatially-sorted in dimension 2, 3 and 4. It becomes much less clear however for higher dimensions, but the plotted ratios grow and would become significant with a much higher number of input points.

These encouraging results must be softened however. In the right part of Figure 3, we plot the ratio of the time taken to compute the Delaunay triangulation with spatial sorting to the time taken with shuffling only. In dimensions 2 and 3, the gains are significant. But as we reach dimensions 4, the gain is barely noticeable. It is null is dimension 5 and slightly negative in dimension 6. This suggest that a much higher number of input points are necessary in order to get significant gain from using spatial sort.

In dimensions 5 and 6, we suspect that our spatial sorting procedure does not leave enough randomness in the leaves of the recursion tree, thus creating points in the Delaunay triangulation with temporary high valence. Since, in these dimensions, computing the conflict zone is far more costly that localizing an input point in the triangulation, the high valence of the inserted point has a heavy influence on the running time for computing the conflict zone. Further experimentations with the spatial sort are to be done.

5.3 Comparison with the specialized 2D 3D CGAL implementations

We observe that, for uniform point distributions, our implementation is roughly 1.86 times slower than the specialized `CGAL::Delaunay_triangulation_2` implementation, and 1.4 times slower than `CGAL::Delaunay_triangulation_3`.

5.4 On a sampling of a hypersurface

We experimented with input points taken from a hypersurface. Specifically, the first $(d-1)$ coordinates are uniformly random, while the last coordinate is taken as the sine of the first, thus forming a sampling of a codimension-1 manifold $\{x_1, x_2, \ldots, x_{d-1} \in [0,1) \text{ and } x_d = \sin(\pi x_1)\}$. Figure 4 reports the time and space used for computing the corresponding Delaunay triangulation. It is expected that the complexity of the triangulation of such data sets be higher than for uniformly distributed inputs. The graphs on Figure 4 provide experimental evidence that this indeed the case. The four-dimensional case is especially interesting as numerous practical applications exist in 3D-space time, e.g., for reconstructing animated 3D scenes [18, 1].

5.5 Comparison to `Qhull`

`Qhull` is a widely used high-dimensional convex hull/Delaunay implementation [7]. As our implementation, `Qhull` does not use extra data-structure apart from the convex hull itself (or the Delaunay triangulation in our case). However, the quickhull algorithm—as applied to the computation of Delaunay triangulations—differs quite significantly from ours. The most important difference being that `Qhull` does not compute the geometric predicates exactly, and thus does not garanty that the computed triangulation is exact. When arithmetic problems occurs, `Qhull` produces so-called

[7] `http://www.qhull.org/news/qhull-news.html#users`

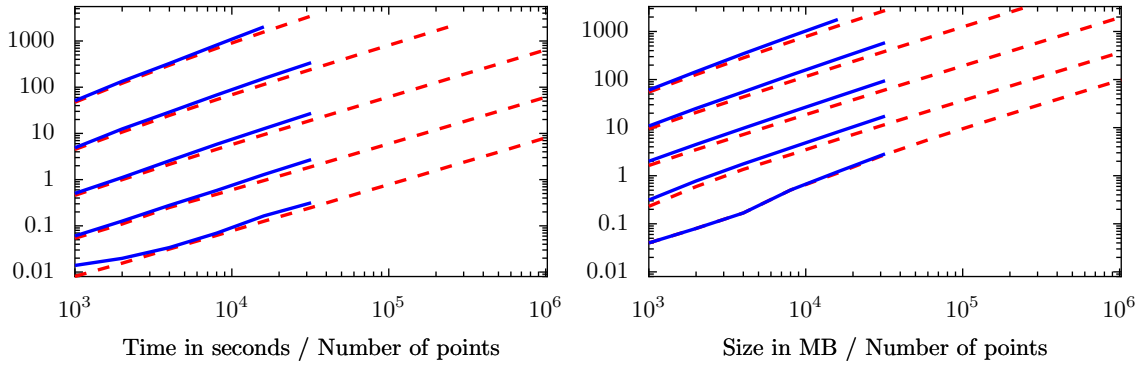

Figure 4: Sampling a manifold. All axes are logarithmic. The red dashed curves are identical to Figure 1. The blue curves are obtained when triangulating a codimension-1 manifold. See subsection 5.4.

"thick faces" [3]. We compared our implementation with `Qhull 2003.1` (the latest version available) for uniformly distributed points in dimension 2 to 5. Figure 5 shows the time and space statistics for this experiment.

As expected, `Qhull` is faster than our implementation with less that 100,000 input points. And we are pleasantly surprised to observe that our implementation becomes faster as the number of input points increases (in dimensions 2, 3 and 4 on the Figure).

`Qhull` uses at least twice the memory used by our implementation in dimensions 4 and 5. We have stop experimenting with more input points when our computer system was trashing memory (the swap memory did start to get used).

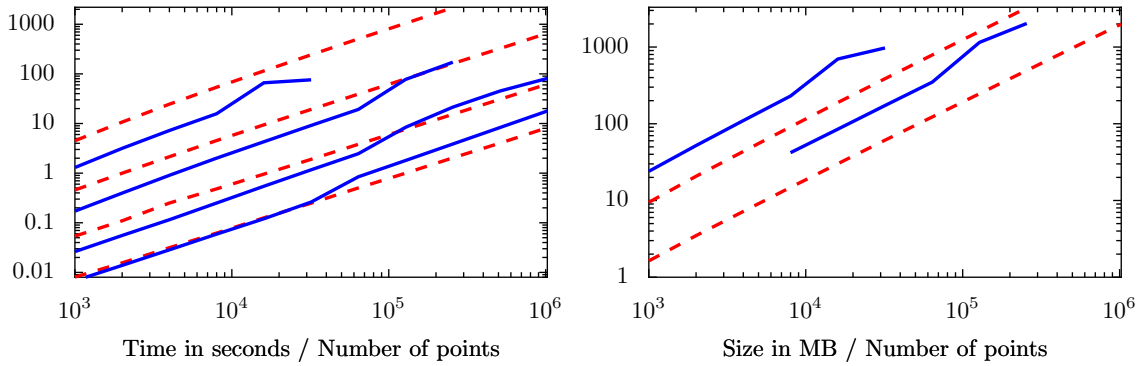

Figure 5: Comparisons with `Qhull`. All axes are logarithmic. The red dashed curves are identical to Figure 1. The blue curves are obtained using `Qhull`. See subsection 5.5. *Left*: Time comparison for dimensions 2 to 5. *Right*: Space comparison for dimensions 4 and 5. In dimension 4, with one, two and four thousand input points, we did not have time to reliably estimate memory usage.

5.6 Time and space compared to `Hull`

`Hull` is an implementation of the convex hull algorithm of Clarkson *et al.* [9]. We tested `Hull` version `0.7` kindly provided by K. Clarkson. This implementation uses exact predicates whenever the input points' coordinates and the dimension permit, see [8]. As for `Qhull`, we tested `Hull` for uniformly distributed points in the unit cube, in dimensions 2 to 6 and stopped increasing the number of input points when the program started trashing memory or when the computation time exceeded one hour. We have also reduced the bit-complexity of the input points' coordinates when necessary to ensure the exact computation of the predicates[8]. The statistics are shown on Figure 6.

[8] We used seven digits (in base 10) in dimensions up to 4 and 6 digits in dimensions 5 and 6.

Overall, `Hull` is about twice as slow as our implementation. Moreover, `Hull` uses much more memory as expected, since the algorithm keeps in memory all the simplices created during the construction.

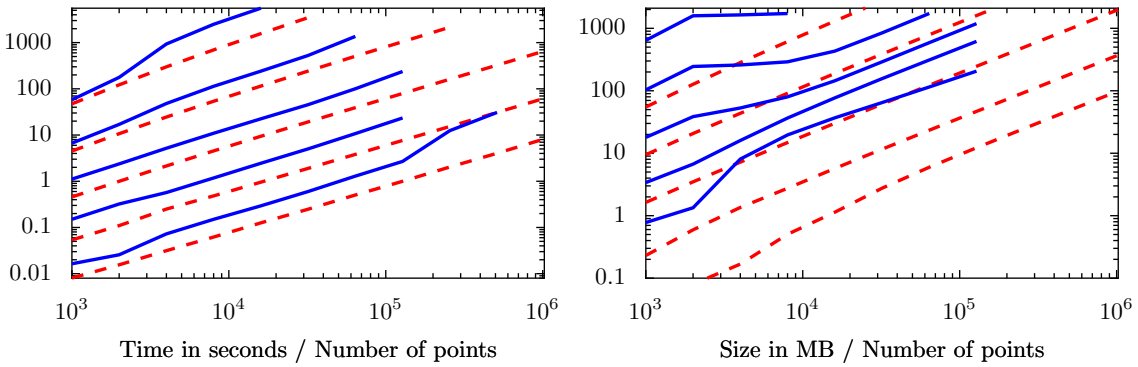

Figure 6: Comparisons with `Hull`. All axes are logarithmic. The red dashed curves are identical to Figure 1. The blue curves are obtained using `Hull`. See subsection 5.6. *Left*: Time comparison for dimensions 2 to 5. *Right*: Space comparison for dimensions 2 to 5. Some time measurements on the left have no corresponding space measurement on the right: This is because the system was trashing memory and the memory measurements became unreliable.

5.7 Comparisons with `Cddf+` and `CGAL_DT`

We tested `Cddf+`, the floating-point version of Cdd+ 0.77, which is not exact, but faster than the version of Cdd that uses exact integer arithmetic, `Cddr+`. We also tested `CGAL_DT`, the current CGAL Delaunay implementation, to which we added exact filtered predicates. Figure 7 shows how less efficient these programs are, in the setting of many medium-dimensional input points.

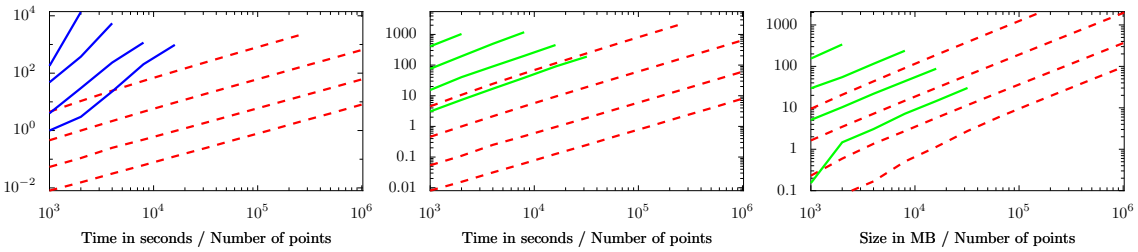

Figure 7: Comparisons with `Cddf+` and `CGAL_DT`. All axes are logarithmic. The red dashed curves are identical to Figure 1. The blue curves are obtained using `Cddf+`. The green curves are obtained using `CGAL_DT`. See subsection 5.7. *Left (`Cddf+`)* and *Middle (`CGAL_DT`)*: Time comparison for dimensions 2 to 5. *Right (`CGAL_DT`)*: Size comparison with `CGAL_DT`, for dimensions 2 to 5.

6 Conclusions

We have presented an implementation of the well-known incremental algorithm for constructing Delaunay triangulations in any dimension. 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 reported in forthcoming papers.

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.

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*, Rio de Janeiro, Brazil, Oct 2007.

[2] Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con BRIO. In *Proceedings of the 19th annual symposium on Computational Geometry (SoCG'03)*, pages 211–219, New York, NY, USA, 2003. ACM.

[3] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[4] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[5] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the delaunay tree. *TCS*, 112:339–354, 1993.

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

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

[8] Kenneth Clarkson. Safe and effective determinant evaluation. In *FOCS '92: Proceedings of the Thirty-First Symposium on Foundations of Computer Science*, pages 387–395, October 1992.

[9] Kenneth Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. *Computational Geometry: Theory and Applications*, 3(4):185–212, September 1993.

[10] Christophe Delage. Spatial sorting. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[11] R. A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 326–333, New York, NY, USA, 1989. ACM.

[12] Michael Hemmer, Susan Hert, Lutz Kettner, Sylvain Pion, and Stefan Schirra. Number types. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. CGAL, 3.3 edition, 2007.

[13] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk, and Jack Snoeyink. Streaming computation of delaunay triangulations. *ACM Trans. Graph.*, 25(3):1049–1056, 2006.

[14] Guohua Jin and John 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.

[15] Piyush Kumar and Edgar A. Ramos. I/o-efficient construction of voronoi diagrams. Technical report, , 2003.

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

[17] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[18] Jochen Süssmuth, Marco Winter, and Günther Greiner. Reconstructing animated meshes from time-varying point clouds. *Computer Graphics Forum (Proceedings of SGP 2008)*, 27(5):1469–1476, 2008.

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 decribed 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 $\mathcal{C}$, interpolating a finite set of points 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 implementation of an any-dimensional space filling curve written by Doug Moore [9] who implemented and optimized an algorithm due to Butz [7]. In particular, we use as a black box the function `i2c` [10] 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 $\texttt{hilbert\_sort\_d}(P, 0, 2^{dl})$.

$\texttt{hilbert\_sort\_d}(P, \texttt{idx}, \texttt{width})$ 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 [11] than the points in the second half P_2. We use `std::nth_element` for that purpose.
5. Recursively call $\texttt{hilbert\_sort\_d}(P_1, \texttt{idx}, \texttt{width}/2)$ and
   $\texttt{hilbert\_sort\_d}(P_2, \texttt{idx} + \texttt{width}/2, \texttt{width}/2)$.

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, mimick 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).

[9] Hopefully, you can find his notes and implementation here: `http://web.archive.org/web/*/www.caam.rice.edu/~dougm/twiddle/`

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

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

Contents

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399