A ROBUST IMPLEMENTATION FOR THREE-DIMENSIONAL DELAUNAY TRIANGULATIONS

ERNST P. MÜCKE

Los Alamos National Laboratory

Computer Research and Applications Group

CIC-3, Mail Stop M986, Los Alamos, NM 87545

12-Apr-95

ABSTRACT

This paper presents `Detri 2.2`, an implementation for Delaunay triangulations of three-dimensional point sets. The code uses a variant of the randomized incremental-flip algorithm, and employs a symbolic perturbation scheme to achieve robustness. The algorithm's time complexity is quadratic in n, the number of input points, and this is optimal in the worst case. However, empirically, we can expect running times roughly proportional to the number of triangles in the final triangulation, which typically is linear in n. Even though the symbolic perturbation scheme relies on exact arithmetic, the resulting code is efficient in practice. This is mainly due to a careful implementation of the geometric primitives and the arithmetic module. The source code is freely available on the Internet.

Keywords: Three-dimensional Delaunay triangulation, robustness, symbolic perturbation, implementation.

1 Introduction

A *triangulation* $\mathcal{T}$ of a three-dimensional point set S is a collection of nonoverlapping tetrahedra covering the convex hull of S such that each point of S is endpoint of at least one tetrahedron in $\mathcal{T}$. The triangulation is called a *Delaunay triangulation* of S, denoted by $\mathcal{D}$, if the circumsphere of each tetrahedron $t_{a,b,c,d}$ contains no point of S other than its endpoints p_a, p_b, p_c, and p_d. Delaunay triangulations are dual to Voronoi diagrams (also known as Dirichlet tessellations). Both concepts have a long history, going back to the works of Gauss, Dirichlet, and Voronoi [32, 33], and there are a variety of applications in different sciences; for relevant surveys and bibliographies consult, among others, Aurenhammer [1], Dobkin [6], or Yao [37]. Various approaches for their construction are described in the computational-geometry literature. Naturally, it was the planar case, which was solved first, but extensions to $\mathbb{R}^3$ and higher dimensions followed soon.[1] Delaunay triangulations are considered particularly useful for automatic three-dimensional mesh generation,[2] a key problem in scientific computing, and a robust and efficient implementation is therefore in high demand.

A classical lemma by Delaunay [5] implies that a triangulation is (globally) Delaunay if and only if each triangle is locally Delaunay. In three dimensions, a triangle $f_{a,b,c}$ is called *locally Delaunay* if it either lies on the convex hull, or it is the shared face of two tetrahedra $t_{a,b,c,d}$ and $t_{a,b,c,e}$, and the circumsphere of $t_{a,b,c,d}$ does not contain point p_e. This local characterization of $\mathcal{D}$ is the basis for the idea to construct $\mathcal{D}$ from *some* triangulation $\mathcal{T}$ using a sequence of local tests, that is, orientation and sphere tests, and local transformations, so-called "flips."

Lawson [25] was the first to do so, but only for the two dimensional case: he shows that an arbitrary sequence of "edge flips" will transform an arbitrary triangulation into the Delaunay triangulation, after $O(n^2)$ flips. Although, his algorithm is not optimal,[3] it is by far the easiest method to implement. Moreover, the idea of flipping can be used to prove several optimality properties of Delaunay triangulations: see, for example, Rajan [28]. Unfortunately, a naïve generalization of the algorithm from two to three dimensions does not work. Starting with an arbitrary triangulation, and using an arbitrary sequence of "flips," it is possible to obtain a triangulation with triangles that are not locally Delaunay, but cannot be flipped. This was first reported by Joe [22]. However, the same author derived an alternative procedure which can be proven to always succeed: the "incremental flip" algorithm [21].

Outline. For completeness, section 2 provides a more formal definition of spatial triangulations using the terminology of simplicial complexes; it also defines the useful lifting map. Section 3 sketches the main steps of the incremental-flip algorithm; in particular, it specifies what we mean by "flipping" triangles and edges in three dimensions. It also discusses how the implementation enables randomization by using a simple yet efficient point-location scheme, and which geometric primitives are necessary to achieve all this. Section 4 shows how we can achieve robustness using a symbolic perturbation scheme, without any "perturbation artefacts." Section 5 presents empirical data about the implementation's actual performance. Section 6 remarks on `Detri`'s input/output files and internal data-structure. It closes with `ftp` references to the freely available code.

[1] For example, Bowyer [2] and Watson [34].

[2] The list of relevant mesh-generation papers is truly endless. As a starting point, the interested reader might want to refer to the works of Schroeder and Shephard [30], Sapidis and Perucchio [29], George and Hermeline [17], Golias and Tsiboukis [18], Wright and Jack [36], Weatherhill and Hassan [35], or Conti, Tomizawa, and Yoshii [4].

[3] Delaunay triangulations of a planar point set of size n can be computed in $O(n \log n)$ time; see, for example, Guibas and Stolfi [20], or Fortune [16].

2 Definitions

This paper deals with triangulations of finite point sets in the three-dimensional Euclidean space $\mathbb{R}^3$. Any of the following definitions can easily be extended to $\mathbb{R}^d$.

Points and orientation. A *point* p_i in $\mathbb{R}^3$ is specified by the vector $(\pi_{i,1}, \pi_{i,2}, \pi_{i,3})$ of its three coordinates, given in a (right-handed) Cartesian coordinate system. A point x is called the *affine combination* of a point set $T = \{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$ if $x = \sum_{\ell=1}^{k} \lambda_{i_\ell} p_{i_\ell}$, for suitable $\lambda_{i_\ell} \in \mathbb{R}$ with $\sum_{\ell=1}^{k} \lambda_{i_\ell} = 1$. If none of the λ_{i_ℓ} is negative, then x is called a *convex combination* of T. The set of all convex combinations of T is the *convex hull* of T, denoted by $\mathrm{conv}(T)$. Furthermore, T is said to be *affinely independent* if no point $p \in T$ is an affine combination of $T - \{p\}$.

We will use the notion of the "orientation" of a sequence of $d+1$ points in $\mathbb{R}^d$, for $d = 3, 4$. The *orientation* of $(p_{i_0}, p_{i_1}, \ldots, p_{i_d})$ is *positive* if

$$D_{(d+1)\times(d+1)}(i_0, i_1, \ldots, i_d) = \det \begin{pmatrix} \pi_{i_0,1} & \pi_{i_0,2} & \cdots & \pi_{i_0,d} & 1 \\ \pi_{i_1,1} & \pi_{i_1,2} & \cdots & \pi_{i_1,d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \pi_{i_d,1} & \pi_{i_d,2} & \cdots & \pi_{i_d,d} & 1 \end{pmatrix} > 0,$$

and *negative* if the determinant is negative. If the points are not affinely independent, the determinant evaluates to zero, and their orientation is not defined.

Simplicial complexes. Let T be a set of $k+1$ points in $\mathbb{R}^3$ with $0 \le k \le 3$. If T is affinely independent, its convex hull, $\mathrm{conv}(T)$, forms a *k-simplex*, denoted by σ_T. The *dimension* of σ_T is k. The empty set is, by definition, called a (-1)-simplex. For convenience, we call a 0-simplex a *vertex*, a 1-simplex an *edge*, a 2-simplex a *triangle*, and a 3-simplex a *tetrahedron*. Note that, if T forms a simplex σ_T, then every subset of T again forms a simplex; it is called a *face* of σ_T. The empty set and σ_T itself are, by definition, trivial faces of σ_T.

A collection $\mathcal{K}$ of k-simplices, for $0 \le k \le 3$, is called a *(three-dimensional) simplicial complex*, if $\mathcal{K}$ satisfies the following two properties: (i) if $\sigma \in \mathcal{K}$ and τ is a face of σ, then $\tau \in \mathcal{K}$, and (ii) if σ_1, $\sigma_2 \in \mathcal{K}$, then $\sigma_1 \cap \sigma_2$ is a face of both.

Delaunay Triangulations. A three-dimensional simplicial complex $\mathcal{K}$ is a *(geometric) triangulation* of a point set S in $\mathbb{R}^3$, if the points of S are the vertices of $\mathcal{K}$ and the union of all simplices in $\mathcal{K}$ is identical to $\mathrm{conv}(S)$. Let $\mathcal{T}$ be a triangulation of S, and assume that its tetrahedra obey the so-called "empty-sphere" criterion, namely, a tetrahedron σ_T is part of $\mathcal{T}$ if and only if its circumsphere does not enclose nor contain any points of $S - T$. Then $\mathcal{T} = \mathcal{D}$, the *Delaunay triangulation* of S. It turns out that the empty-sphere criterion specifies a unique triangulation of S, provided the points are in general position.

General position assumption. We assume that the points of S are in *general position*. Here, this means that

(1) no four points lie on a common plane, and
(2) no five points lie on a common sphere.

Assumption (1) implies that all subsets, $T \subseteq S$ with $|T| = k+1$,and $1 \leq k \leq 3$, are affinely independent and form proper k-simplices; (2) assures that $\mathcal{D}$ is unique.

Clearly, the general position assumption is not acceptable in practice. However, it simplifies definitions and algorithms tremendously. And it is justified, because we can use a symbolic perturbation scheme (as discussed in section 4) which elegantly "implements" general position even for point sets that are in arbitrary position.

Lifting map. One can use a lifting map to transform, generally speaking, the Delaunay triangulation problem in $\mathbb{R}^d$ to the problem of constructing the convex hull in $\mathbb{R}^{d+1}$. This idea goes back to Brown [3]; details on the construction of convex hulls in d dimensions can be found texts like Edelsbrunner [9]. In this paper, we will use the lifting map for $d = 2, 3$.

Identify $\mathbb{R}^d$ with the $x_1x_2 \ldots x_d$-space in $\mathbb{R}^{d+1}$, that is, the subspace $x_{d+1} = 0$. The *lifting map* is a geometric transform that projects points $p = (\pi_1, \pi_2, \ldots, \pi_d)$ in $\mathbb{R}^d$ along the x_{d+1}-axis onto the paraboloid of revolution U: $x_{d+1} = \sum_{i=1}^{d} x_i^2$ in $\mathbb{R}^{d+1}$. Let $p_U = (\pi_1, \pi_2, \ldots, \pi_d, \sum_{i=1}^{d} \pi_k^2)$ be the image of p, and define $S_U = \{p_U \,|\, p \in S\}$. The convex hull of S_U, $\text{conv}(S_U)$, is a convex polytope with n vertices in $\mathbb{R}^{d+1}$. A facet belongs to the *lower* boundary of this polytope if the polytope lies on the side of the positive x_{d+1}-axis of the hyperplane that contains the facet. Otherwise, it belongs to the *upper* boundary of $\text{conv}(S_U)$. If we project all facets of the lower boundary of $\text{conv}(S_U)$, along with their faces, parallel to the x_{d+1}-axis, into $\mathbb{R}^d$, then we obtain the Delaunay triangulation of S.

3 The Randomized Incremental-Flip Algorithm

The idea of constructing Delaunay triangulations by local transformations or flips goes goes back to Lawson [24, 25]. He introduced this method (also known as the "Lawson's flip" algorithm) for constructing two-dimensional Delaunay triangulations in 1972. Given a finite point set $S \in \mathbb{R}^2$, the method initially constructs an arbitrary triangulation $\mathcal{T}$ of S. This triangulation is then altered step by step through a sequence of "edge flips" until all edges are Delaunay edges, and thus the Delaunay triangulation is obtained.

Consider, for example, a convex set $\{p_i, p_j, p_k, p_u\}$ of four points in $\mathbb{R}^2$, as in figure 1. It has two possible triangulations. One triangulation contains edge $e_{i,j}$, shared by two triangles $f_{i,j,k}$ and $f_{i,j,u}$. If it is not Delaunay, we can replace $e_{i,j}$ by edge $e_{k,u}$, shared by $f_{i,u,k}$ and $f_{j,u,k}$, and the resulting triangulation is Delaunay. This operation is called an *edge flip* in $\mathbb{R}^2$.

It helps to use the lifting map from $\mathbb{R}^2$ to $\mathbb{R}^3$ to illustrate this procedure. The triangulation $\mathcal{T}$ is the Delaunay triangulation $\mathcal{D}$ if and only if its projection $\mathcal{T}_U$ is identical to the lower boundary of $\text{conv}(S_U)$; call it $\mathcal{D}_U$. If an edge $e_{i,j}$ is not locally Delaunay, then its projection is not part of $\mathcal{D}_U$. Therefore, it is possible to attach a tetrahedron underneath the projected edge and its two incident triangles. In other words, flipping edges in $\mathbb{R}^2$ corresponds to attaching tetrahedra in $\mathbb{R}^3$. This process terminates when $\mathcal{T}_U = \mathcal{D}_U$, that is, when $\mathcal{T} = \mathcal{D}$.

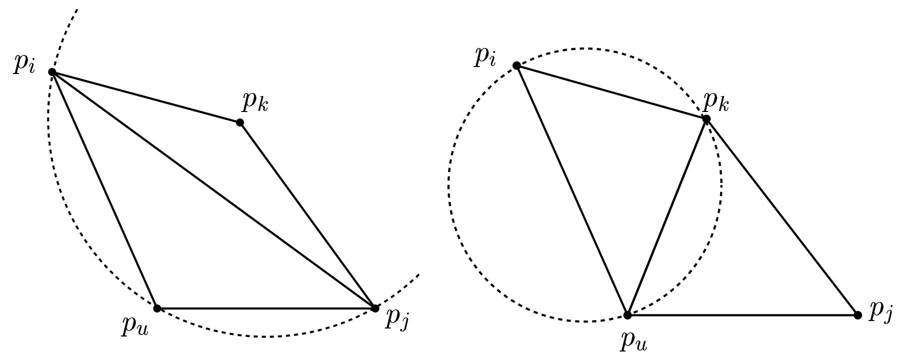


Figure 1: The two triangulations of a convex set $\{p_i, p_j, p_k, p_u\}$ in $\mathbb{R}^2$.

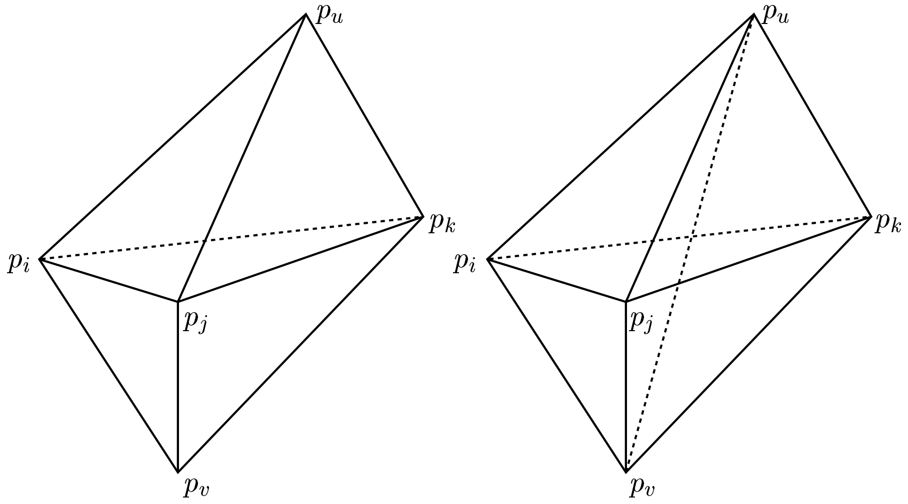


Figure 2: The two triangulations of the convex set set $S = \{p_i, p_j, p_k, p_u, p_v\}$ in $\mathbb{R}^3$. The triangulation to the left consists of two tetrahedra: $t_{i,j,k,u}$ and $t_{i,j,k,v}$, while the triangulation to the right contains three tetrahedra: $t_{u,v,i,j}$, $t_{u,v,j,k}$, and $t_{u,v,k,i}$.

Flips in $\mathbb{R}^3$. Before generalizing the flip algorithm to three dimensions, we have to specify what we mean by a "flip" in $\mathbb{R}^3$. Consider a set $S' = \{p_i, p_j, p_k, p_u, p_v\}$ of five points in $\mathbb{R}^3$ in general position. If there exists a point, say p_v, such that $\text{conv}(S') = \text{conv}(S' - \{p_v\})$, then S' has only one possible triangulation with four tetrahedra, all incident to p_v. This triangulation is trivially Delaunay. Otherwise, all points of S' lie on the boundary of $\text{conv}(S')$, and S' has two possible triangulation, as in figure 2; one of them is the Delaunay triangulation of S'.

The lifting map from $\mathbb{R}^3$ to $\mathbb{R}^4$ can be used for illustration. The two triangulations of S' correspond to the lower and upper boundary of $\text{conv}(S'_U)$, which is a four-dimensional simplex. The triangulation corresponding to the lower boundary is the Delaunay triangulation. A *flip* replaces the triangulation that corresponds to the upper boundary by the one that corresponds to the lower boundary.

We distinguish two different ways to flip faces that are not locally Delaunay:

1. Let $f_{i,j,k}$ be a triangle in $\mathcal{T}$ which is not locally Delaunay, and let $t_{i,j,k,u}$ and $t_{i,j,k,v}$ be the two incident tetrahedra. If the union of the two tetrahedra is

convex, then they can be replaced by three tetrahedra: $t_{u,v,i,j}$, $t_{u,v,j,k}$, and $t_{u,v,k,i}$. In other words, we flip a triangle $f_{i,j,k}$, which is not locally Delaunay, to the edge $e_{u,v}$, whose three incident triangles are all locally Delaunay. We call this a *triangle(-to-edge) flip* or *2-3 flip*.

2. If there is an edge $e_{u,v}$ in $\mathcal{T}$ of degree 3, and none of the three incident triangles are locally Delaunay, then the three incident tetrahedra $t_{u,v,i,j}$, $t_{u,v,j,k}$, and $t_{u,v,k,i}$, can be replaced by two, namely, $t_{i,j,k,u}$ and $t_{i,j,k,v}$.[4] The new triangle $f_{i,j,k}$ is locally Delaunay. This is called a *edge(-to-triangle) flip* or *3-2 flip*.

Incremental flip algorithm. Unfortunately, the straightforward generalization of Lawson's flip to $\mathbb{R}^3$, using the above two local transformations, does not work, see Joe [22]; however, in a follow-up paper he proves the following lemma [21]: If a single point p is added to the Delaunay triangulation of a point set S in $\mathbb{R}^3$, then an arbitrary sequence of flips results in the Delaunay triangulation of $S \cup \{p\}$.

Subsequently, Rajan [28] shows that flipping incrementally works in any dimension, but his constructive proof requires a priority queue to find a specific sequence of flips. Edelsbrunner and Shah [12] remove this technicality, and also generalize the result to regular triangulations (also known as weighted Delaunay triangulations).

In three dimensions, the procedure is as follows. Start with 4 points, and form a tetrahedron; obviously this is the Delaunay triangulation of these points, call it $\mathcal{D}_4$. Then perform the following loop.

for $i = 5, \ldots, n$ **do**
  assume $\mathcal{D}_{i-1}$ to be the Delaunay triangulation of $\{p_1, p_2, \ldots, p_{i-1}\}$;
  add point p_i to it, forming a new triangulation $\mathcal{T}_i$;
  transform $\mathcal{T}_i$ into $\mathcal{D}_i$ via a sequence of flips;
end for

Joe's result assures that this procedure always succeeds. Its time complexity can be shown to be of $O(n^2)$, which is worst-case optimal. The argument uses the lifting map and the upper-bound theorem for convex polytopes in four dimensions.[5] The original algorithm sorts the points along the x-axis and incrementally adds them in this order. This scheme has the advantage that adding point p_i to $\mathcal{D}_{i-1}$, and thus forming $\mathcal{T}_i$, becomes easy,[6] but it also has the tendency to produce intermediate triangulations of higher complexity than the final triangulation.[7] This is especially true for degenerate data.

[4] If an edge $e_{u,v}$ is incident to three triangles that are not locally Delanuay, then $e_{u,v}$ cannot be part of $\mathcal{D}$; moreover, it implies that $t_{i,j,k,u} \cup t_{i,j,k,v}$ is convex.

[5] See [9] for references to the upper-bound theorem for convex polytopes.

[6] Hint: point p_i will always lie outside $\mathcal{D}_{i-1}$, and point p_{i-1} lies on the boundary of $\mathcal{D}_{i-1}$ and will always be visible from p_i.

[7] A triangulation of n points in $\mathbb{R}^3$ can have anywhere between $O(n)$ and $O(n^2)$ triangles for the very same point set; see, for example [9]. Still, Delaunay triangulations typically only have linear size; see Dwyer [8].

Randomization. Randomizing the algorithm overcomes this: we follow the same procedure as above, only now we add points in a random order. There exists no rigorous expected-time analysis giving a better bound than $O(n^2)$, but the arguments in [12] indicate that the expected number of flips is roughly proportional to the expected number of simplices in the final triangulation. Empirical results with existing implementations justify this; see Facello [15], but also section 5 here.

When adding points in a random order, we can no longer rely on point p_i to lie outside $\mathcal{D}_{i-1}$, and if p_i lies inside, then the enclosing tetrahedron needs to be identified. In [12], this point-location problem was solved using a directed acyclic graph storing the history of all flips performed. This history dag serves as the data structure for point location and yields to an algorithm which is elegant and optimal in the worst case. However, the dag needs $O(|\mathcal{D}|)$ additional storage, and this severely limits the usefulness of the algorithm in practice. Section 3.1 describes an alternative, which is efficient in practice yet only uses $O(1)$ additional storage.

3.1 A simple point-location scheme

`Detri 2.2` employs a simple point-location scheme; in lack of a better name, call it the "walk-through" method. We need some terminology to discuss the method and the problem. For an *oriented* triangle $a \in \mathcal{T}$, let a^+ denote its positive side. If a^+ does not lie outside $\mathcal{T}$, then there exists a tetrahedron t incident to a with $a^+ \cap t \neq \emptyset$; denote this by $t = t_{a^+}$. A triangle $a = \sigma_T \in \mathcal{T}$ is said to be *visible* from a point p, if $p \in a^+$ and the simplex $\sigma_{T \cup \{p\}}$ does not intersect any other triangle of $\mathcal{T}$. The point-location problem is now, to identify a triangle $a \in \mathcal{T}$ which is visible from a query point p, under the assumption that p is not a vertex of $\mathcal{T}$. More specifically, if p lies inside $\mathcal{T}$, then find a such that $p \in t_{a^+}$; otherwise, find a convex-hull triangle a that is visible from p. The following procedure does the job.

> Choose a random triangle $a \in \mathcal{T}$ such that $p \in a^+$;
> **loop**
>   **if** a is a convex-hull triangle and a^+ is outside $\mathcal{T}$ **then**
> p lies outside $\mathcal{T}$, and a is visible from p;
>   **exit loop;**
>   **else if** $p \in t_{a^+}$ **then**
> p lies inside $\mathcal{T}$, contained in tetrahedron t_{a^+};
>   **exit loop;**
>   **else**
>   let b be a face of t_{a^+} such that
>   - b has same orientation than a, and
>   - $p \in b^+$;
>   set $a = b$;
>   **end if**
> **end loop**

Let $(a_0, a_1, \ldots, a_\ell)$ be the sequence of oriented triangles a with $p_h \in a^+$ for which the above loop is executed. Here, $\mathcal{T}$ is a Delaunay triangulation, and the volume of $\bigcap_{\lambda=0}^{\ell} a_{\lambda}^{+}$ is strictly decreasing for increasing ℓ. Thus, the loop always terminates successfully. This simple idea of walking through the triangulation is not new, it is already mentioned in Bowyer [2], with a reference to Green and Sibson [19], who use it for computing Delaunay triangulations in the plane.

It is difficult to give a good estimate for the time complexity of the walk-through method. In the worst case, one obviously has to test all triangles of $\mathcal{T}$, and that is $O(n^2)$. A more realistic estimate is that the number of steps required to locate p is proportional to $n^{\frac{1}{3}}$ (imagine $\mathcal{T}$ with vertices on a cubic grid; see also [2, 19]). Empirically, it performs even better. This is especially so, if we choose the initial triangle from a random sample of size $K = c \cdot \log(n)$, for some constant c. Then, we choose a "good" starting triangle according to some fast heuristics. Our implementation selects the closest triangle with respect to the floating-point Euclidean distance function. In comparison to the long-integer arithmetic, necessary for the robust implementation of the geometric primitives (see section 3.3), this is computational inexpensive. With this, there is empirical evidence that the time complexity of the walk-through point-location scheme appears to be close to logarithmic in n. In particular, the average search length is roughly 1/2 of the average search length of the history-dag method, with only $O(1)$ additional space [26].

3.2 Flipping link facets

After adding a new point p_i to $\mathcal{D}_{i-1}$, and thus forming $\mathcal{T} = \mathcal{T}_i$, there will, in general, be triangles in $\mathcal{T}$ which are not locally Delaunay. However, these triangles are in some sense local around p_i. They are either incident to p_i, or part of the "link" of p_i. We call a triangle $a = \sigma_T \in \mathcal{T}$ a *link facet* of p_i if $\sigma_{T \cup \{p_i\}}$ is a tetrahedron in $\mathcal{T}$, and denote this by $a \in \mathcal{L}(p_i)$. To transform $\mathcal{T}$ to $\mathcal{D}_i$, it is sufficient to consider only link facets as possible candidates for flipping [12]; moreover, the sequence of flips is arbitrary, and the loop listed below will always terminate with $\mathcal{T} = \mathcal{D}$.

```
Assume L = L(p_i) and T to be the current triangulation;
while ∃ triangle a ∈ L such that a ∈ T and a is not locally Delaunay do
  remove a from L;
  if ∃ edge e of a such that e is transformable then
    flip edge e in T, and add up to 2 new link facets to L;
  else if a is transformable then
    flip triangle a in T, and add up to 3 new link facets to L;
  else
    "skip" a as nontransformable and add it to L again;
  end if;
end while
```

A geometric test for local Delaunayhood (the sphere test of section 3.3) has to be performed whenever a valid triangle a is selected from $\mathcal{L}$. Assume it is not locally Delaunay. The algorithm will try to "transform" a by either flipping one of its edges e, or by flipping the triangle a itself. Since it is possible to test whether an edge is transformable *without* any geometric tests, and since geometric tests are computational expensive, we check transformability for edges first; this reduces the overall number of tests.

If an edge e has degree 3, and does not lie on the boundary of the triangulation, then either all three incident triangles are locally Delaunay or all three triangles are not locally Delaunay. Here, e is an edge of a triangle $a \in \mathcal{L}$ which is not locally Delaunay. Thus, a constant-time test whether or not e has degree 3 decides whether e is transformable and an edge flip can be performed.

If none of the edges of a is transformable, a series of geometric tests (orientation tests; see section 3.3) is necessary to decide whether $t_{a^+} \cup t_{a^-}$ is convex, and thus triangle a is transformable. If so, a triangle flip is performed; if not, a is (temporarily) discarded or "skipped" as nontransformable.

Note that, whenever a flip is performed, $\mathcal{L}$ changes and new link facets have to be added to the list. Since there cannot be more than a constant number of new link facets after each flip, the total time required by the above loop is proportional to the number of flips performed. `Detri 2.2` implements $\mathcal{L}$ as a stack but for no reason other than convenience; it could as well be a queue, a priority queue, for example, ordered by the volume of the incident tetrahedra, or a set, allowing random access.[8] Also, when we remove triangles from the triangulation during the flipping process, we mark them as "deleted" and later check the marks when selecting triangles from $\mathcal{L}$. However, in order to avoid duplicates in $\mathcal{L}$, we also mark triangles as "on the list" when they are added to $\mathcal{L}$.

3.3 Geometric Primitives

Constructing Delaunay triangulations essentially requires two geometric tests. One test decides on which side of a plane, spanned by three points, a fourth point lies. The other test decides on which side of a sphere, spanned by four points, a fifth point lies. By the "side" of a sphere, we mean the inside or the outside.

Orientation test. Let $T = \{p_i, p_j, p_k\}$ and define h_T as the unique plane that contains all three points of T. This plane can be oriented if we replace the set T by the sequence T; for example, $T = (p_i, p_j, p_k)$. A point p_u lies on the *positive side* of h_T if the sequence (p_u, p_i, p_j, p_k) has positive orientation; otherwise, it is said to lie on the *negative side* of h_T. According to section 2, we get,

$$T = (p_i, p_j, p_k)\text{: } p_u \text{ lies on the positive side of } h_T \iff D_{4\times 4}(u, i, j, k) > 0,$$

and p_u lies on the negative side if the determinant is negative. Notice that the determinant equals zero if and only if the points lie on a common plane, in other words, the general position assumption is violated.

Sphere test. Given a set $T = \{p_u, p_i, p_j, p_k\}$ of four points, we need to decide whether a fifth point, p_v, lies inside or outside the sphere ∂b_T going through the points of T. The general position assumption says that the degenerate case, where

[8] And it would be interesting to see how different schemes influence the total number of flips.

p_v lies on ∂b_T, does not occur. The implementation of this test makes use of the lifting map from $I\!R^3$ to $I\!R^4$. In particular, a sphere ∂b with center $c = (\gamma_1, \gamma_2, \gamma_3)$, and radius ρ, is mapped to the hyperplane ∂b_U: $x_4 = \sum_{\ell=1}^{3}(2\gamma_\ell x_\ell - \gamma_\ell^2) + \rho^2$. This hyperplane has the property that $U \cap \partial b_U$ projected along the x_4-axis into the $x_1x_2x_3$-space yields ∂b. Moreover, a point p lies inside (outside) ∂b if and only if p_U lies vertically below (above) ∂b_U. The resulting formula assumes that each point $p_i \in S$ has a fourth coordinate $\pi_{i,4} = \sum_{j=1}^{3} \pi_{i,j}^2$.

$$T = \{p_u, p_i, p_j, p_k\}\text{: } p_v \text{ lies inside } \partial b_T \iff D_{4\times4}(u,i,j,k)D_{5\times5}(u,i,j,k,v) > 0$$

The first determinant, $D_{4\times4}(u,i,j,k)$, is a corrective term which is necessary because the sphere ∂b_T does not change if the points of T are permuted. The second determinant, $D_{5\times5}(u,i,j,k,v)$, expresses the fact that the lifting map transforms the sphere test in $I\!R^3$ to a orientation test in $I\!R^4$.

4 Robustness

The code is *robust* in the following sense: if S is in general position, then $\mathcal{D}$ is unique and `Detri` will compute it; if S contains degeneracies, then the computed $\mathcal{D}$ will be the Delaunay triangulation for a "slightly" perturbed point set $S(\varepsilon)$. This is achieved by a symbolic perturbation scheme called SoS, Simulation of Simplicity [10].

In a nutshell, SoS works as follows. For each point $p_i \in S$, its coordinates $\pi_{i,j}$ are replaced by polynomials in ε, namely $\pi_{i,j}(\varepsilon) = \pi_{i,j} + \varepsilon(i,j)$. The $\varepsilon(i,j)$ terms are themselves short polynomials in ε. Depending on the indices i and j, they are chosen in such a way that any determinant in the perturbed coordinates is nonzero; in other words, $S(\varepsilon)$ is in general position. Once a given point set is indexed, the perturbation is fixed; the ε itself is never specified, but is assumed to be arbitrarily small but positive.

Details of the method can be found in [10, 26]; here only so much: SoS can be seen as a systematic way to break ties. It effectively replaces the evaluation of any determinant D in the original coordinates by an evaluation of $D(\varepsilon)$, which is a polynomial in ε, whose coefficients are subdeterminants of D. Evaluating the sign of $D(\varepsilon)$ for given coordinates then means evaluating a sequence of subdeterminants of D, namely, D_0, D_1, …, D_s, the so-called *significant terms*, with $D_0 = D$ and $D_s = 1$. Because of the specific form of the perturbation, the first subdeterminant D_ℓ, with $0 \leq \ell \leq s$, which evaluates to nonzero, also determines on the sign of the ε-polynomial; ℓ is then called a particular *evaluation depth* of $D(\varepsilon)$.

The ε-polynomial $D_{4\times4}(i,j,k,v)(\varepsilon)$ implementing the orientation test has 15 significant terms, $D_{5\times5}(u,i,j,k,v)(\varepsilon)$ for the sphere test has 50. However, the probability, that a subdeterminant D_ℓ has to be evaluated, decreases drastically for increasing ℓ, and the actual number of significant terms of ε-polynomials is not really significant in practice (see section 5).

Remark. One important remark is necessary concerning the SoS implementation of the sphere test. In our implementation, SoS perturbs the sum of squares, $\sum_{j=1}^{3} \pi_{i,j}^2$, as if it were a forth coordinate, $\pi_{i,4}$. This "cheap trick" [14] is used for reasons of compactness and efficiency. However, since the algorithm mixes primitives in $I\!R^3$ and $I\!R^4$, inconsistencies might occur and some care is in order. Recall that the lifted points $p_i \in S_U$ lie on the paraboloid of revolution U: $x_4 = \sum_{\ell=1}^{3} x_\ell^2$ in $I\!R^4$. This is no longer true for $S_U(\varepsilon)$. In particular, $S_U(\varepsilon)$ is no longer necessarily a convex point set. Unfortunately, this happens to be the underlying assumption for the correctness of the algorithm. Fortunately, there is only one (special) case for which $S_U(\varepsilon)$ is not convex, namely, if S includes coinciding points $p_i = p_j$, for $i \neq j$; and it is easy to eliminate such points in a preprocessing step, without changing the overall time complexity to the worse. If, on the other hand, $p_i \neq p_j$, for all $i \neq j$, then $S_U(\varepsilon)$ is convex, because the unperturbed S_U is convex, and the ε of the perturbation is assumed to be arbitrarily small. Thus, the "cheap trick" works, and the sphere predicate can be implemented using SoS.

Also, it is only fair to mention, that symbolic perturbation is under heated debate in principle; see, for example, Seidel [31]. Moreover, there exist alternative perturbation schemes, which either are more general in their scope, do not require "cheap tricks" as above, or produce theoretically shorter sequences of subdeterminants; see Yap [39, 38], Knuth [23], or Emiris and Canny [13, 14], respectively. Still, the original SoS method of [10] is the most "implementable" scheme, and it works well for Delaunay triangulations.

Perturbation Artefacts. Note that SoS does not allow to "selectively" remove certain degeneracies, while retaining others; rather, it *fully* implements what theoreticians call the "general position" of the input. Moreover, the resulting triangulation is for $S(\varepsilon)$ rather than S. For nondegenerate configurations in S, this does not matter: simply ignore the ε-perturbation and use the original coordinates. However, for certain degeneracies in the data, some postprocessing might be desirable. Consider, for example, coplanar points on the boundary of the convex hull of S. The perturbation will move some of these points off the boundary, while some will remain on it. As a result, the points that were moved inside the convex hull, will be covered by "flat" tetrahedra in $\mathcal{D}$ of $S(\varepsilon)$. After ignoring the ε-terms of the coordinates, these tetrahedra will be total flat, so-called *volume-0 slivers*. Such slivers may occur quite frequently (for example, with points on a cubic grid more than 1/3 of the tetrahedra are volume-0 slivers) and this might be awkward for certain applications. Fortunately, they can be removed in a postprocessing step, using $O(1)$ time per sliver [26].

Another interesting "side effect" of the perturbation is that the resulting triangulation is guaranteed not to contain any volume-0 slivers inside the convex hull.[9] In other words, after the above postprocessing, which removes the slivers on the convex hull, the resulting Delaunay triangulation $\mathcal{D}$ of S will not contain any volume-0 tetrahedra, no matter what input is given.

[9] One can prove this using certain facts of how the SoS perturbation is implemented, in particular, the fact that, $\varepsilon(i, j_1) \gg \varepsilon(i, j_2)$ for $j_1 < j_2$.

5 Performance

This section takes a look at the performance of `Detri 2.2`. Actually, all the statistical data presented here is from a predecessor version of the code, `Detri 1.2`; however, the differences between the two versions concern mainly the interface; in terms of the core implementation both versions are identical. In the following, we will address the following questions about the program. What is the actual performance of the method? How efficient is the walk-through point-location scheme in practice? How large is the speedup due to randomization? What is the price for the exact arithmetic in SoS?

Hardware. The experiments were run on SiliconGraphics Indigo and Crimson workstations, with 50 Mhz R4000 RISC processors and 48 Mb and 192 Mb of main memory, respectively. The machines were chosen such that their main memory was always large enough for the data and page swapping was not a factor.

Categories of data. We gathered statistics for more than 40 different data sets, ranging from $n = 318$ to $n = 26304$. The data consist of points in $\mathbb{R}^3$, given by their x_1-, x_2-, and x_3-coordinates. We restrict ourselves to 32-bits integer coordinates, but only to compare the performance of the code with other implementations. `Detri` itself understands coordinates given in fix-point format since version `1.2`.

The performance of the algorithm depends somewhat on the distribution of the points. We tested the code on data from several different categories, as listed below. It is interesting to distinguish data according to *how* degenerate the point set is. This is, in some sense, measured by the mean depth of SoS evaluations. Data sets with a particular high mean depth can be considered as "highly degenerate." (Most degeneracies in our experiments are due to coplanarities.) Data sets with a maximum evaluation depth of 0 are in general position.

- r *Random points.* They usually do not display any degeneracies.
- C *Chaotic (or strange) attractors.* They are used in computer graphics to model natural phenomena, such as, smoke.[10] Chaotic attractors are essentially "very complex" surfaces with points rendered in a random fashion; see, for example, Pickover [27]. In fact, concerning degeneracies, they behave much like random points, with a mean depth of more or less 0.0; however, two of our chaotic attractors have a maximum depth 2.
- M *Molecular data.* These point sets consists of the coordinates of atom centers. They are input data of a "real-life" application in molecular modeling and protein folding.[11] The molecular data used in our experiments does not contain any degeneracies.
- S *Points on surfaces.* These include points generated randomly on certain surfaces, for example, on fractal terrains. Some degeneracies exist; maximum depth: 6, means: less then 0.06.
- V *Volumetric data.* These are (random) threshold samples of "voxel" data used in volume visualization, where each grid point (x_1, x_2, x_3) has assigned some value $f(x_1, x_2, x_3)$.[12] Volumetric data usually contains many degeneracies; maximum depth: 13, means: less then 0.1.
- G Geometric models. Such data sets consist of point coordinates derived from geometric models, with *all* the original connectivity information (that is, edges and triangles) removed. Such models can be found in public domain databases for solid modelers. SoS evaluation depths in

[10] The chaotic attractor data was generated with code of R. Hudson, UIC, and P. Haeberli, SGI.

[11] The geometric structure of many proteins is available in public domain databases.

[12] The volumetric test data is due to P. Moran, NCSA, UIUC.

Procedure			Share of total CPU cycles		Share of SoS primitives operations
delaunay			98%		
	search	(i.e., point-location)	10%		29% of positive
	add p_i to $\mathcal{D}_{i-1}$		2%		3% of positive
	flip		86%		
		get next facet in $\mathcal{L}$		60%	100% of in_sphere
		triangle-to-edge flip		23%	68% of positive
		edge-to-triangle flip		3%	

Table 1: CPU cycle distribution while executing Detri.

Procedure	Share of total CPU cyles
in_sphere	57%
positive	28%
$\sum$	85%

Procedure	Share of total CPU cyles
mul	39%
psub	13%
padd	4%
$\sum$	56%

Table 2: CPU cycle distribution for SoS and long-integer arithmetic.

our sample: maximum: 9, with means up to 0.13

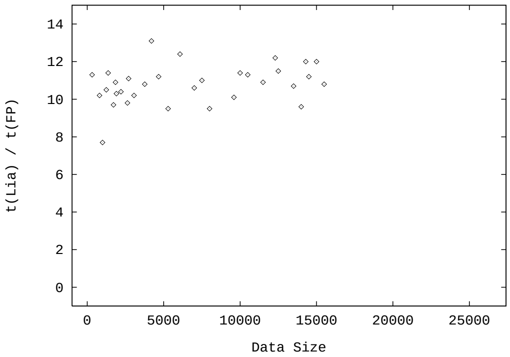
R *Range data.* It consists of random samples taken from images obtained with a laser range finder.[13] The data consists of surfaces of the form $x_3 = f(x_1, x_2)$, and contains many degeneracies; maximum depth: 11, means up to 0.6.

O *Other, miscellaneous data.* This category contains, for example, the real-life "universe" data, obtained from a simulation of the positions of galaxies within our universe,[14] or point sets on a perfect cubic grid; the latter can be considered a worst-case example in terms of degeneracies; maximum evaluation depth: 13, means of up to 0.98.

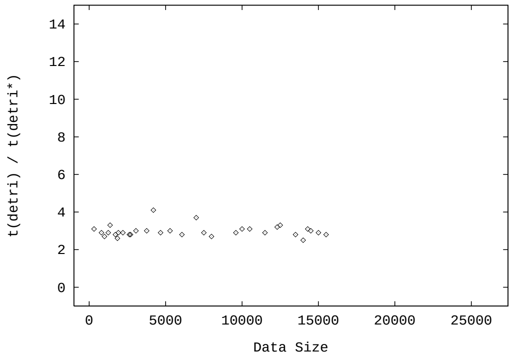
Experiments. The code of Detri 1.2 was run several times for each data set (sample size $N = 13$). The plots in this section show the sample means $\overline{Y}$ of the corresponding measurements. Their confidence intervals are described in relative terms as a percentage of $\overline{Y}$. The confidence level used was 90%. We say a sample mean has a *confidence interval* of $\pm H\%$ if the interval $[\overline{Y} - H\%, \overline{Y} + H\%]$ contains the real mean value μ with a probability of 90%.

CPU cycle distribution. The first question to ask is this: "Where do all the cycles go?" Table 1 lists the major procedures of the incremental-flip code and their relative share of the total number of CPU cycles. The percentages were obtained by run-time profiles generated with pixie.[15] Although, the counts will change slightly from data set to data set (and from one randomized run to the next), the variance is only marginal. Table 1 lists the CPU cycle distribution for a typical run.

[13] Range data source: F. Stein, Institute for Robotics and Intelligent Systems, USC.
[14] The "universe" is due to M. Dyksterhouse, Computer Science Department, UIUC.
[15] See SGI's Irix man pages, for example, under http://reality.sgi.com/cgi-bin/getman/.



(a) Observed slow-down factors of long-integer versus floating-point arithmetic with respect to determinant evaluations. Sample mean: 10.79 (±2.9%)



(b) Hypothetical slow-down factors of long-integer versus floating-point arithmetic with respect to the overall program. Sample mean: 2.98 (±3.1%)

Figure 3: Long-integer arithmetic in `Detri`.

It is not surprising that the majority of the time is spent in procedure `flip`, see section 3.2. Within this procedure, the tests controlling the loop are the most time-consuming. This is where the sphere tests are performed which check incoming triangles for local Delaunayhood.

Recall from section 3.3 how the sphere test can be expressed by two determinants, one 5-by-5 one 4-by-4. The actual implementation, however, manages to eliminate the evaluation of the 4-by-4 determinant. After all, it is only a corrective term to check for the orientation of the involved points, and it is possible to implement the incremental flip algorithm such that the orientation of the points, for which the `in_sphere` predicate is called, is always positive.

Another substantial amount of time is spent on performing orientation tests; they are needed for the point location and the triangle flips.[16] The implementation of the orientation test, the `positive` predicate, consists of evaluating a 4-by-4 determinant. Keep in mind that both `in_sphere` and `positive` are implemented with SoS, and that subdeterminants are occasionally evaluated in order to resolve degeneracies. However, degenerate cases occur with low probability, and the evaluation of k-by-k determinants, with $k < 4$, typically accounts for less than 0.5% of the total CPU time. Overall, SoS is responsible for roughly 85% of the CPU time; see table 2. The long-integer arithmetic implementing the exact determinant evaluation for SoS accounts for 56% of the CPU time.

Exact arithmetic. We see, that the largest chunk of CPU time is spent for the robust implementation of the geometric primitives; especially, in doing long-integer arithmetic. This is the real overhead of SoS. The code uses its own long-integer module, `Lia`, which is semidynamic and thus avoids the usual memory-allocation overhead. To this date, `Lia` appears to be the most efficient long-integer package that is freely available.

Empirical comparisons of SoS code versus fast but *nonrobust* floating-point imple-

[16]Recall that edge flips do not need any geometric tests.

mentations of the incremental-flip algorithm suggest a slow-down factor between 2.5 and 4.0, assuming input data with coordinates with no more than 32 bits. Moreover, consider the following experiment:

- Run `Detri` and store the parameters for all determinant evaluations.
- Input these to a different code evaluating all specified determinants using both long-integer as well as floating-point arithmetic.
- Time the difference.

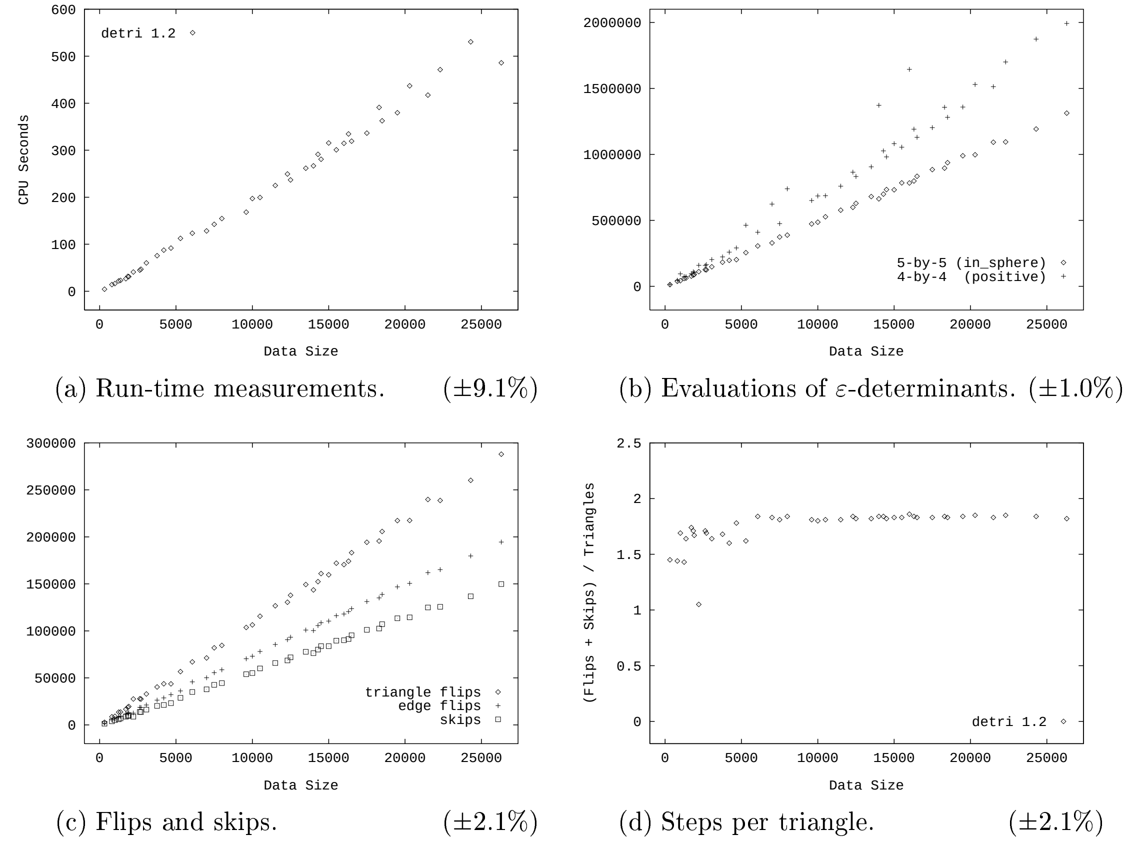
Let `t(detri)`, `t(Lia)`, and `t(FP)` denote the CPU time needed for the overall execution of `Detri 1.2`, and for the above long-integer and floating-point evaluations, respectively. The formula `t(detri*) = t(detri) - t(Lia) + t(FP)` then gives a rough estimate for the running time of a hypothetical floating-point version `detri*`, which (with some "magic") would produce the correct result. See figures 3 (a) and (b) for the corresponding slow-down factors, observed for the data sets in our sample with $n < 16$k.[17] The results show that long-integer arithmetic slows down the determinant evaluations by a factor of 11, which leads to an overall slow-down factor for `Detri` of roughly 3, when compared to the hypothetical floating-point implementation. We believe that this is an adequate price to pay for a compact and robust implementation of a rather involved geometric algorithm. A price, that could be reduced when even more efficient implementations for long-integer arithmetic would be available.

Performance. We now come to the the actual performance of `Detri 1.2`, and thus `Detri 2.2`, illustrated by the plots in figure 4. The plotted values constitute the sample means $\overline{Y}$ of the measurements in question. The corresponding confidence intervals are usually too small to be drawn in the plot. We simply state the worst confidence interval, with respect to the whole sample, in the captions of the figures.

Figure 4 (a) plots the actual run-time measurements of `Detri 1.2`. There is only a slight variance between the data of different categories. A high degree of degeneracy does not influence the total running time too much; this is in sharp contrast to the nonrandomized version of the algorithm. In any case, a positive correlation between the required CPU seconds and the number of necessary primitive operations (that is, the required evaluations of ε-determinants) is obvious; see figure 4 (b): this plot shows the total count of 5-by-5 and 4-by-4 minor evaluations. Note that all 5-by-5 determinants occur in `in_sphere` tests, but 4-by-4 minors are due to `positive` tests *and* to degenerate `in_sphere` tests. In particular, we observe a high number of evaluations of 4-by-4 determinants for volumetric and range finder data, that is, the highly degenerate data.

Plot (c) in figure 4 draws the number of triangle(-to-edge) and edge(-to-triangle) flips, including the number of "skips.". Adding up these numbers results in a run-time independent cost model for the incremental flip method. It measures the number of times the loop of the `flip` procedure is executed. Figure 4 (d) divides this number of "steps" of the algorithm by n_f, the number of triangles in the final Delaunay triangulation. We see that the performance of the algorithm is sensitive

[17] Disk space limitations restricted this particular experiment to $n \leq 16$k.



(a) Run-time measurements. (±9.1%)

(b) Evaluations of ε-determinants. (±1.0%)

(c) Flips and skips. (±2.1%)

(d) Steps per triangle. (±2.1%)

Figure 4: Performance of `Detri 1.2`.

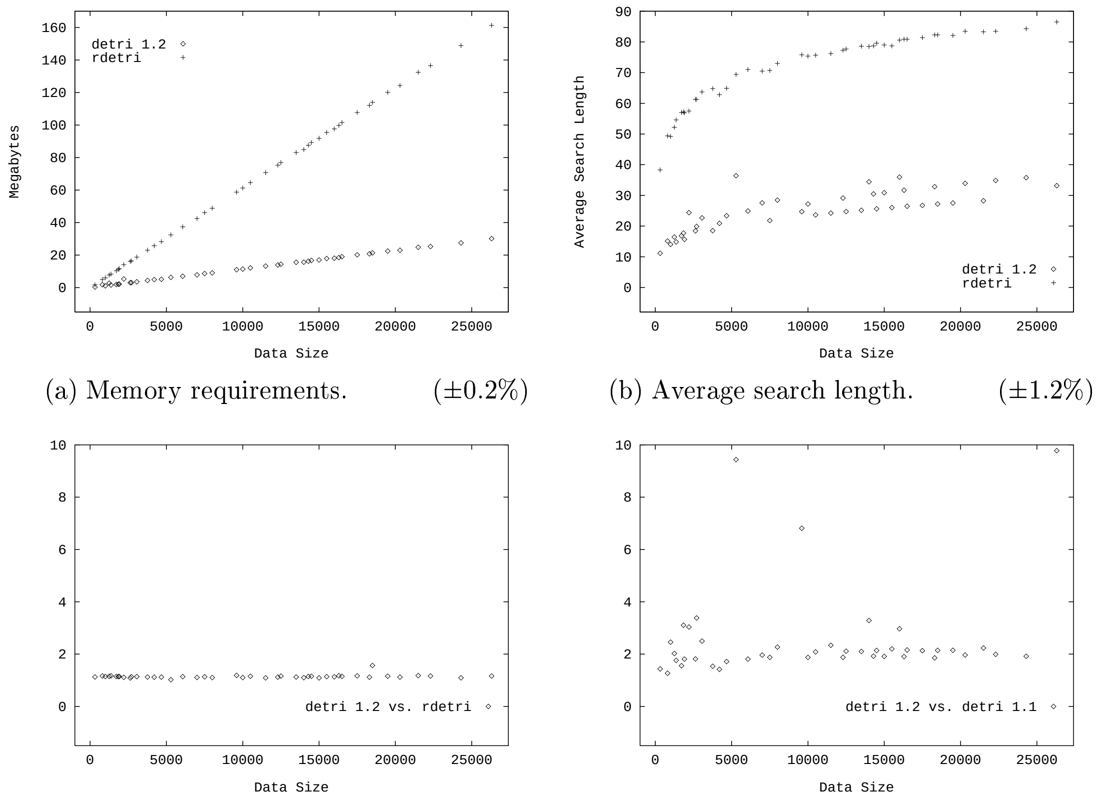
to the size of the output. Within the window of our sample data, we can say that the algorithm never needs more that $1.9n_f$ steps to construct the Delaunay triangulation, and n_f is the output size.

Comparing incremental-flip algorithms. How does our variant of the incremental flip algorithm compare with the other methods? In the following, `Detri 1.2` is compared with two other implementations:

- `Detri 1.1`, a *non-randomized* version of `Detri`, essentially implementing the original method of Joe [21], and
- `rdetri`, a code implementing the incremental-flip using the history dag as described in [12]; the code is due to Facello [15].

Figure 5 shows four plots comparing the three programs, which all use the same set of geometric primitives, implemented with SoS to cope with degeneracies, and the same data structure for the triangulation; they are thus perfectly well comparable. The major difference between our variant and the one of [12] is that they use a history dag to implement the point location. In contrast to point location with history dag, the walk-through method of section 3.1 does not use any extra memory. This explains the enormous difference in memory requirements between the two implementations; see figure 5 (a).[18] It is also twice as fast in locating points in the

[18] According to personal communications with Facello, a newer version of `rdetri` (which is now



(a) Memory requirements. (±0.2%)

(b) Average search length. (±1.2%)

(c) Speedup: `Detri 1.2` versus `rdetri`.

(d) Speedup: `Detri 1.2` versus `Detri 1.1`.

Figure 5: Comparing `Detri 1.2` with `rdetri` and `Detri 1.1`.

triangulation, provided we employ the "random sample" heuristic to find a starting triangle relatively close to the query point as discussed. Figure 5 (b) documents this by plotting the "average search length" counting the average number of `positive` tests required to locate a point with respect to one run.

For the data in our experiment, we see that the history dag always performs worse than the walk-through method. However, the latter seems to have a higher variance between the different categories of data. In the range below $n = 10$k, we have poorer performance for data with many points on the boundary of the convex hull and for data with a high count of degeneracies. Beyond $n = 10$k, three groups with different levels of average search length emerge. Random coordinates perform best, followed by chaotic attractors, the highly degenerated range sets of the range finder data perform the worst. A further investigation of the walk-through strategy and its performance would be interesting. The same is true for a better tuning of the size of the random sample to determine a "good" starting point for the search. The current implementation simply chooses the closest triangle out of $K = 15\log(n_i)$ random triangles, with n_i the number of points in the current triangulation. This procedure yields reasonable results. The detection of the "good" initial triangle usually accounts for no more that 15% of the time spent in the point-location procedure.

called `regtri`) could reduce the memory requirements by half; this is still roughly a factor of 2 overhead per triangle.

Of course, point-location itself is not responsible for more that 10% of the total running time of the program; see table 1. Therefore, the overall performance of `Detri 1.2` is only slightly better than the one of `rdetri`: on average by 14%. Figure 5 (c) plots the speedup factor `Detri` versus `rdetri`; the average factor in our sample is 1.14.

When comparing `Detri 1.2` with the non-randomized version `Detri 1.1`, we achieve a speedup of roughly 2; see figure 5 (d). Note even more dramatic speedup factors (beyond 6) for the highly degenerate volumetric data. The `1.1` version of `Detri` essentially implements Joe's original algorithm [21]. However, there is one difference. Rather than inserting the points sorted along the x_1-, or some other axis, `Detri 1.1` inserts them sorted along *some* arbitrary direction. We observed that Joe's algorithm performs particularly badly if the data points come in planes orthogonal to the chosen direction in which these points are to be sorted. By choosing this direction in an random fashion, this problem is eased.[19]

6 Input, Output, and Availability

`Detri` reads ASCII input files, where each data line i contains the three coordinates of point p_i. The coordinates can be integers or in a user-defined fix-point format; internally, they will always be converted to long-integers. The output is a binary file, containing the "triangle-edge" data structure of the triangulation. This data structure is described in detail in [26]. It is a simplification of Dobkin and Laszlo's edge-facet structure [7], specialized for triangulations without duals. It uses 36 bytes per triangle, and fully represents topology and order in a three-dimensional triangulation. The user has the option to either use the binary file of the data structure directly (code for the access functions is provided in the distribution below), or use a run-time option in `Detri` to dump a list of the triangulation's faces to an ASCII file.

The `C` source code is available via anonymous `ftp` from `cs.uiuc.edu`, directory `/pub/edels/geometry/`. This directory also contains a collection of data sets, as used in the experiments of section 5. Alternatively, the code can be downloaded as part of the three-dimensional Alpha shape software as it is distributed in `ftp.ncsa.uiuc.edu`, directory `/Visualization/Alpha-shape/alpha-2.2/`. The `Detri` distribution includes the above mentioned long-integer module, `Lia`, and the SoS implementation for the necessary set of geometric primitives.

[19] It should be noted that, in order to implement Joe's algorithm, we need one more primitive not yet mentioned in this text: the `smaller` predicate, which compares coordinates. See [10] for a SoS implementation of this predicate; it can then be modified such that it compares coordinates along a given direction (that is, it compares the distance from two points to a plane).

Acknowledgments

Version `2.2` and earlier versions of `Detri` were mainly developed while the author was completing his dissertation under Herbert Edelsbrunner at the Department of Computer Science, UIUC, where he had the oportunity to be part of the exciting alpha shape project [11]. Obviously, the author is deeply indebted to Herbert for all his support. He also wants to thank the Software Development Group at NCSA, for their encouragement and for providing access to their computing facilities.

References

[1] F. Aurenhammer. Voronoi diagrams — a survey of a fundamental data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[2] A. Bowyer. Computing Dirichlet tessellations. *Computer Journal*, 24(2):162–166, 1981.

[3] K. Q. Brown. Voronoi diagrams from convex hulls. *Information Processing Letters*, 9(5):223–228, 1979.

[4] P. Conti, M. Tomizawa, and A. Yoshii. Generation of oriented three-dimensional Delaunay grids suitable for the control volume integration method. *International Journal for Numerical Methods in Engineering*, 37(19):3211–3227, 1994.

[5] B. Delaunay. Sur la sphère vide. *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

[6] D. P. Dobkin. Computational geometry and computer graphics. *Proceedings of the IEEE*, 80(9):1400–1411, 1992.

[7] D. P. Dobkin and M. J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4(1):3–32, 1989.

[8] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6(4):343–367, 1991.

[9] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[10] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[11] H. Edelsbrunner and E. P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, 1994.

[12] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[13] I. Z. Emiris and J. Canny. A general approach to removing degeneracies. In *Proceedings of the 32nd Annual IEEE Conference on Foundations of Computer Science (FOCS 91)*, pages 405–413, 1991.

[14] I. Z. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proceedings of the 8th Annual Symposium on Computational Geometry*, pages 74–82, 1992.

[15] M. A. Facello. Implementation of a randomized algorithm for delaunay and regular triangulations in three dimensions. *Computer Aided Geometric Design*, 12(4):349–370, 1995.

[16] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[17] P. L. George and F. Hermeline. Delaunay's mesh of a convex polyhedron in dimension d. Application to arbitrary polyhedra. *International Journal for Numerical Methods in Engineering*, 33(5):975–995, 1992.

[18] N. A. Golias and T. D. Tsiboukis. An approach to refining three-dimensional tetrahedral meshes based on Delaunay transformations. *International Journal for Numerical Methods in Engineering*, 37(5):793–812, 1994.

[19] P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *Computer Journal*, 21(2):168–173, 1978.

[20] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *Transactions on Graphics*, 4(2):74–123, 1985.

[21] B. Joe. Construction of three-dimensional Delaunay triangulations using local transformations. *Computer Aided Geometric Design*, 10(4):123–142, 1989.

[22] B. Joe. Three-dimensional triangulations from local transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(4):718–741, 1989.

[23] D. E. Knuth. *Axioms and Hulls. Lecture Notes in Computer Science 606*. Springer-Verlag, Berlin, 1992.

[24] C. L. Lawson. Generation of a triangular grid with applications to contour plotting. Memo 9, Jet Propulsion Laboratory, Pasadena, California, 1972.

[25] C. L. Lawson. Software for C^1 surface interpolation. In J. R. Rice, editor, *Mathematical Software III*, pages 161–194. Academic Press, New York, 1977.

[26] E. P. Mücke. *Shapes and Implementations in Three-Dimensional Geometry.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Ubana, Illinois, 1993. Technical report UIUCDCS-R-93-1836.

[27] C. Pickover. A note on rendering 3-D strange attractors. *Computers and Graphics*, 12(2):263–267, 1988.

[28] V. T. Rajan. Optimality of the Delaunay triangulation in R^d. In *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 357–363, 1991.

[29] N. Sapidis and R. Perucchio. Combining recursive spatial decompositions and domain Delaunay tetrahedrizations for meshing arbitrarily shaped curved solid models. *Computer Methods in Applied Mechanics and Engineering*, 108(3/4):281–302, 1993.

[30] W. J. Schroeder and M. S. Shephard. A combined octree Delaunay method for fully-automatic 3-D mesh generation. *International Journal for Numerical Methods in Engineering*, 29(1):37–55, 1990.

[31] R. Seidel. The nature and meaning of perturbations in geometric computing. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science(STACS 94)*, volume 775 of *Lecture Notes in Computer Science*, pages 3–17, 1994.

[32] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier Mémoire: Sur quelques propriétés des formes quadratiques positives parfaites. *Journal für die Reine und Angewandte Mathematik*, 133:97–178, 1907.

[33] G. Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième Mémoire: Recherches sur les parallélloèdres primitifs. *Journal für die Reine und Angewandte Mathematik*, 134:198–287, 1908.

[34] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *Computer Journal*, 24(2):167–172, 1981.

[35] N. P. Weatherhill and O. Hassan. Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints. *International Journal for Numerical Methods in Engineering*, 37(15):2005–2039, 1994.

[36] J. P. Wright and A. G. Jack. Aspects of three-dimensional constrained Delaunay meshing. *International Journal for Numerical Methods in Engineering*, 37(11):1841–1861, 1994.

[37] F. F. Yao. Computational geometry. In R. A. Earnshaw and B. Wyvill, editors, *Handbook of Theoretical Computer Science, Volume A: Algorithms in Complexity*, pages 345–490. Elsevier Science Publishers, Amsterdam, 1990.

[38] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Journal of Computer and System Sciences*, 40:2–18, 1990.

[39] C. K. Yap. Symbolic treatment of geometric degeneracies. *Journal of Symbolic Computing*, 10:349–370, 1990.