INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Delaunay Triangulations for Moving Points

Pedro Machado Manhães de Castro — Olivier Devillers

N° 6750

October 2008

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6750--FR+ENG

Delaunay Triangulations for Moving Points

Pedro Machado Manhães de Castro , Olivier Devillers

Thème SYM — Systèmes symboliques
Projets Geometrica

Rapport de recherche n° 6750 — October 2008 — 22 pages

Abstract: This paper considers the problem of updating efficiently a Delaunay triangulation when vertices are moving under small perturbations. Its main contribution is a set of algorithms based on the concept of vertex tolerance. Experiment shows that it is able to outperform the naive *rebuilding* algorithm in certain conditions. For instance, when points, in two dimensions, are relocated by Lloyd's iterations, our algorithm performs about several times faster than *rebuilding*.

Key-words: tolerance, Delaunay triangulation, moving points

Triangulation de Delaunay pour des Points qui Bougent

Résumé : Ce travail considère le problème de la mise à jour de manière efficace d'une triangulation de Delaunay où les sommets subissent des petites perturbations. La contribution principale de ce travail est un ensemble d'algorithmes basés sur le concept de la tolérance d'un sommet. L'experimentation montre qu'il parvient a dépasser la performance de l'algorithme naif de reconstruction dans certaines conditions. Par exemple, quand les points sont soumis, aux itérations de Lloyd, en dimension deux, notre algorithme termine plusieurs fois plus rapidement que la reconstruction.

Mots-clés : tolérance, triangulation Delaunay, points mobiles

1 Introduction

Delaunay triangulation of a set of points is one of the most famous data structures produced by computational geometry. Two main reasons explain this success: −1− computational geometers eventually produce efficient algorithms to compute it [6, 36], and −2− it has many practical uses such as meshing for finite elements methods [16] or surface reconstruction from point clouds [7].

For several applications the data are moving and thus the triangulation evolves with time. It arises for example when meshing deformable objects [10], or in some algorithms relocating the points by variational methods [2, 38].

We first recall that the *Delaunay triangulation* $DT(\Omega)$ of a set Ω of n points in $\mathbb{R}^d$ is a *simplicial complex* such that no point in Ω is inside the circumsphere of any *simplex* in $DT(\Omega)$ [33, 4]. Several algorithms to compute the Delaunay triangulation are available in the literature. Many of them work in the *static* setting [19, 24], where the points are fixed and known in advance. There are also a variety of so-called *dynamic* algorithms [11, 13], in which the points are fixed but not known in advance and thus the triangulation is maintained under point insertions or deletions. If some of the points move continuously in $\mathbb{R}^d$ and we want to keep track of the modifications of the triangulation, we are dealing with *kinetic* algorithms [1, 34]. Finally, an important variation is when the points move but we are only interested in the triangulation at some discrete times, we call that context *timestamps relocation*.

When we are in the context of timestamps relocation, a simple and efficient method to consider is the following: for each timestamp we simply recompute the Delaunay triangulation of the set $\Omega(t)$ at timestamp t_i. We call this algorithm *rebuilding*. Note that this algorithm does not take any previous work into account, thus for timestamp t_i it does not benefit from any possible correlation between $\Omega(t_j)$ and $\Omega(t_i)$, with $t_j < t_i$. If points are well-distributed, it could achieve an $O(kn\log(n))$ computation time, where n is the number of vertices on the triangulation and k is the number of distinct timestamps.

Although the rebuilding algorithm is naive and has a poor theoretical complexity, it stands for an algorithm hard to outperform when most of the points move, as observed in previous work [34]. In this paper, we propose to compute for each point a safety zone where the point can move without changing its connectivity in the triangulation. Several experiments conducted on synthetic and practical data show added value of the method, in particular for mesh smoothing [3] where the points are converging to a final position.

2 Fundamentals

2.1 Certificates and Tolerances

A *predicate* is a function on a set of primitives which returns one value in a discrete set of possible results. In this paper, we consider the usual case where we compute a numerical value called the *predicate discriminant* and, the result of the predicate is the sign of this value. When one or more predicates are used to evaluate whether a geometric data structure is valid or not, we call each one of those predicates a *certificate*. Hereafter, a certificate is *valid* when it is positive.

Given any certificate $C : \mathcal{A}^m \rightarrow \{-1, 0, 1\}$ acting on a m-tuple of points $\zeta = (\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_m) \in \mathcal{A}^m$, where $\mathcal{A}$ is the space where the input lies in. We define the *tolerance* of C with respect to ζ, namely $\epsilon_C(\zeta)$ or simply $\epsilon(\zeta)$ when there is no ambiguity, the largest displacement applicable to $\mathbf{z}$ in ζ without invalidating C. More precisely, the tolerance can be stated as follows:

$$\epsilon_C(\zeta) = \inf_{\substack{\zeta' \\ C(\zeta')\leq 0}} dist_H(\zeta, \zeta'), \tag{1}$$

where $dist_H(\zeta, \zeta')$ is the Hausdorff distance between two finite sets of points.

By abuse of notation, $\mathbf{z} \in \zeta$ means that $\mathbf{z}$ is one of the points of ζ. Let $\mathcal{X}$ be a finite set of m-tuples of points in $\mathcal{A}^m$, then the *tolerance of an element* $\mathbf{e} \in \mathcal{A}$ with respect to a given certificate C and $\mathcal{X}$, namely $\epsilon_{C,\mathcal{X}}(\mathbf{e})$ or simply $\epsilon(\mathbf{e})$ when there is no ambiguity, can be defined as follows:

$$\epsilon_{C,\mathcal{X}}(\mathbf{e}) = \inf_{\substack{\zeta \ni \mathbf{e} \\ \zeta \in \mathcal{X}}} \epsilon_C(\zeta). \tag{2}$$

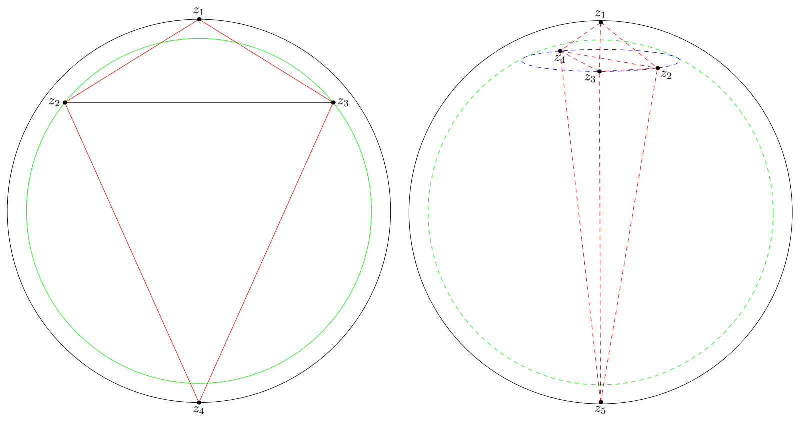
2.2 Delaunay Triangulations: Certificate and Tolerance

A non-self-intersecting triangulation lying in $\mathbb{R}^d$ can be checked to be Delaunay using the *empty-sphere certificate*, which states that, for each facet of the triangulation, the hypersphere through the $d+1$ vertices of a simplex on one side does not contain the vertex on the other side. Therefore, this certificate is applied to any $d+2$ distinct points $\mathbf{z}_1(x_{1\,1},..,x_{1\,d}),\ldots,\mathbf{z}_{d+2}(x_{d+2\,1},x_{d+2\,d})$ of the triangulation which belong to the same pair of incident cells. We call such a pair a *bi-cell* in the sequel. For the interior facets, the certificate is the sign of the determinant of the following matrix:

$$\begin{pmatrix} x_{1\,1} & \ldots & x_{1\,d} & \sum_{i=1}^{n} x_{1\,i}^2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_{d+2\,1} & \ldots & x_{d+2\,d} & \sum_{i=1}^{n} x_{d+2\,i}^2 & 1 \end{pmatrix}. \tag{3}$$

Special cases at the boundary of the convex hull can be solved by adding a vertex at infinity to virtually triangulate the outside of the convex hull [41, 32]. For the sake of simplicity, special treatment at infinity will not be further detailed in this paper.

The tolerance involved in a Delaunay triangulation is the *tolerance of the empty-sphere certificate* acting on any bi-cell of a Delaunay triangulation. From Equation 1, it corresponds to the size of the smallest perturbation the bi-cell's vertices could undergo so as to become cospherical. This is equivalent to compute the hypersphere that minimizes the maximum distance to the $d+2$ vertices, or to compute half the width of the *d-annulus of minimum width* containing the vertices (See Figure 1).



(a) two dimensions (b) three dimensions

Figure 1: Minimum-width 2-annulus and 3-annulus respectively.

2.3 Safe regions

Let $\mathcal{T}$ be a triangulation lying in $\mathbb{R}^d$ and $\mathcal{B}$ a bi-cell in $\mathcal{T}$. The *interior facet* of $\mathcal{B}$ is the common facet of the two cells of $\mathcal{B}$. The *opposing vertices* of $\mathcal{B}$ are the remaining two vertices that do not belong to its interior facet. Two bi-cells $\mathcal{B},\mathcal{B}'$ are *neighbors* if they share a cell. If the interior facet and opposing vertices of $\mathcal{B}$ are respectively inside and outside (or on) a common hypersphere $\mathcal{S}$, we say that $\mathcal{B}$ verifies the *safety condition*. We call $\mathcal{B}$ a *safe bi-cell* and $\mathcal{S}$ its *delimiter*. If a vertex $\mathbf{z}$ belongs to the interior facet of $\mathcal{B}$, then the *safe region* of $\mathbf{z}$ with respect to $\mathcal{B}$ is the region inside the delimiter. Otherwise, the *safe region* of $\mathbf{z}$ with respect to $\mathcal{B}$ is the region outside the delimiter. The intersection of the safe regions of $\mathbf{z}$ with respect to each one of its adjacent bi-cells is the *safe region* of $\mathbf{z}$. Finally, if all the bi-cells of $\mathcal{T}$ are safe bi-cells, then we call $\mathcal{T}$ a *safe triangulation*. When a triangulation is a safe triangulation, we say that it verifies the *safety condition*.

It is clear that a safe triangulation is equivalent to a Delaunay triangulation, since:

- Each delimiter can be shrunk in such a way that it touches the vertices of the interior facet, and thus defining an empty-sphere passing through the interior facet of its bi-cell.
- The *empty-sphere* property of Delaunay triangulation facets defines itself an empty-sphere passing through the interior facets of the bi-cells. Those empty-spheres are delimiters.

Proposition 1. *Given a Delaunay triangulation $\mathcal{T}$, if its vertices move arbitrarily yet inside their safe regions, then $\mathcal{T}$ remains Delaunay.*

It is a direct consequence of the equivalence between safe and Delaunay triangulations, since if the vertices remains inside their safe regions, then $\mathcal{T}$ remains a safe triangulation, and hence a Delaunay triangulation.

Among all possible delimiters of a bi-cell, we define the *standard delimiter* as the median hypersphere of the d-annulus with the inner-hypersphere passing through the interior facet and the outer-hypersphere passing through the opposing vertices. Both median hypersphere and d-annulus are unique. We call the d-annulus, the *generator of the standard delimiter.*

In order to compute the generator of the standard delimiter of a given bi-cell $\mathcal{B}$, one needs to compute the center of a d-annulus. This task can be done by intersecting the bisector of its opposing vertices and $d-1$ bisectors between vertices of its interior facet. The intersection of d nonparallel hyperplanes is a standard linear system with d variables with an unique solution which is the center of the annulus.

2.4 The tolerance of a vertex

Let $\mathcal{D}(\mathcal{B})$ be the delimiter of a given bi-cell $\mathcal{B}$. Then, for a given vertex $\mathbf{z} \in \mathcal{T}$, we define:

$$\tilde{\epsilon}(\mathbf{z}) = \inf_{\substack{\mathcal{B} \ni \mathbf{z} \\ \mathcal{B} \in \mathcal{T}}} dist_H(\mathbf{z}, \mathcal{D}(\mathcal{B})). \tag{4}$$

We have that $\tilde{\epsilon}(\mathbf{z}) \leq \epsilon(\mathbf{z})$, since the delimiter generated by the minimum-width d-annulus of the vertices of a bi-cell $\mathcal{B}$ maximizes the minimum distance of the vertices to the delimiter. If we consider the standard delimiter of a bi-cell as its delimiter, we have the equality in Equation 4.

For a given triangulation $\mathcal{T}$, the tolerance of each of its vertices can be computed by successively computing the tolerance of each bi-cell, and keeping the minimum value on each of its vertices. As a result, we have an $\epsilon(\mathbf{z})$ associated to each vertex $\mathbf{z} \in \mathcal{T}$, representing the maximum distance it could reliably move without breaking any Delaunay rules.

With this in mind, we define the *tolerance region* of $\mathbf{z}$ as the ball centered at the location of $\mathbf{z}$ with radius $\epsilon(\mathbf{z})$. That is the biggest ball centered at the location of $\mathbf{z}$ and contained inside its safe region.

We can always extend the tolerance region to be the entire safe region, but its shape is substantially more complex as it is defined by the intersection of several hyperspheres and complement of hyperspheres. See Figure 2 for an illustration.

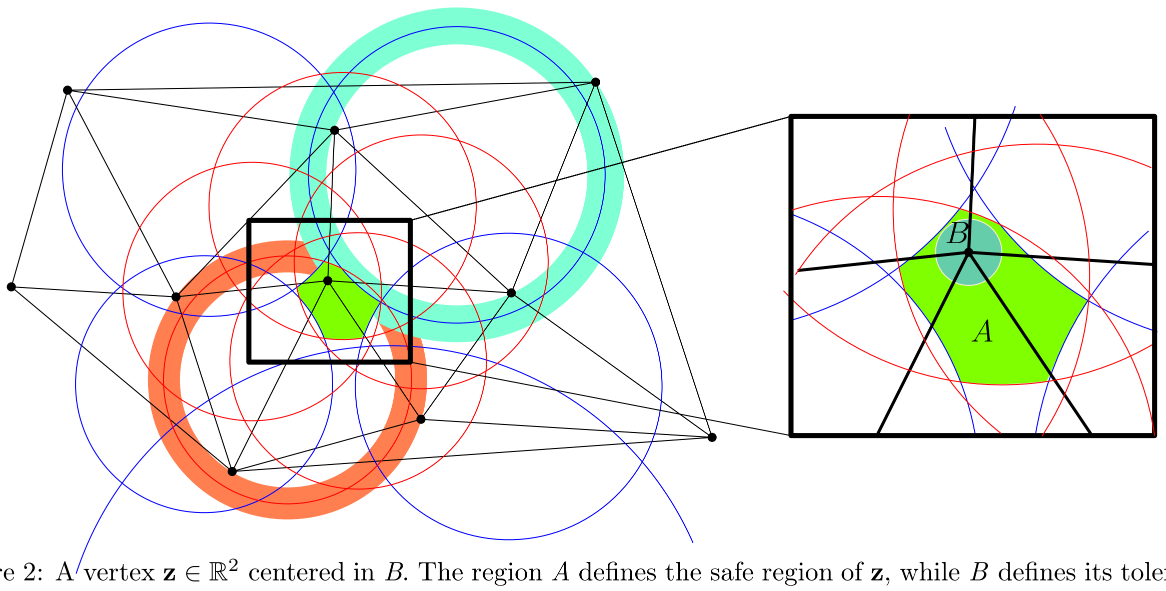


Figure 2: A vertex $\mathbf{z} \in \mathbb{R}^2$ centered in B. The region A defines the safe region of $\mathbf{z}$, while B defines its tolerance region.

3 Delaunay Maintenance Algorithms

3.1 Placement

Another naive updating algorithm, significantly different from rebuilding, is the *placement* algorithm. It consists of, iterating over all relocated vertices, taking each vertex and walking to the cell containing its new position, inserting a vertex at the new position, and, finally removing the old vertex from the triangulation. Each placement requires one removal, one insertion and one point location per relocated point. As the displacements of such moving points are supposed to be small, those point locations are usually fast. In favorable configurations with small displacements and constant local complexity, localization, insertion, and deletions take constant time per point, leading to $O(n)$ complexity per timestamp, which is theoretically better than the $O(n \log n)$ complexity of rebuilding. In practice however, the cost of deletions is the most expensive and hence rebuilding the whole triangulation is faster. On the other hand, the computational complexity of the placement algorithm depends mostly on the number of moving points whereas rebuilding does not.

There are a number of algorithms which requires to compute the next location of a vertex one by one, updating the Delaunay triangulation after each relocation [2, 38, 39]. Despite its computational cost, as placement algorithm is dynamic, unlike rebuilding, it remains suitable for such applications. Another relevant side effect of the static nature of rebuilding is that some significant overhead is necessary to preserve a certain order on accessing the vertices after a relocation, which may be useful for applications that reference the vertices of the triangulation externally and use the displacement algorithm as a black-box.

3.2 Improving Placement for Small Displacements

In two dimensions, a small modification of the placement algorithm leads to a substantial acceleration, by a factor of two. This modification consists of flipping edges when a vertex displacement does not inverse the orientation of any of its adjacent triangles. The key idea is to avoid as many removal operations as possible. In three dimensions, repairing the triangulation is more involved [37]. However, we conjecture that this same optimization can be done in that case as well, by generalizing the incremental topological flipping analysis from Edelsbrunner and Shah [17].

Furthermore, a weaker version of this improvement consists of using placement only when at least one topological modification is needed; otherwise the coordinates of the vertex are simply updated. Naturally, this additional computation leads to an overhead, though experiments show that it pays off when displacements are small enough. Moreover, when this optimization is combined with the algorithms described in the next subsection, our experiments show evidences that it is definitely a good option.

3.3 Algorithms Based on Tolerance

We redesigned the placement algorithm so as to take into account the tolerance region of each relocated vertex. In practice, the algorithms proposed are capable of correctly decide whether a vertex displacement requires an update of the connectivity or not, so as to trigger the trivial update condition. It is dynamic, preserving all benefits from placement compared to rebuilding.

The first algorithm takes into account the tolerance region of a vertex. It will denoted by *tolerance algorithm*, and it works as follow:

Data structure. Consider a triangulation $\mathcal{T}$, where for each vertex $\mathbf{z} \in \mathcal{T}$ we associate two point locations: $\mathbf{f_z}$ and $\mathbf{m_z}$. We will call them the *fixed position* and the *moving position* of a vertex. The fixed position will be useful to *fix* a reference position for a moving point. The moving position of a given vertex is its actual position, and will change at any time it is relocated. Initially, the fixed positions and moving positions are equal. We call $\mathcal{T}_\mathbf{f}$ and $\mathcal{T}_\mathbf{m}$ the embedding of $\mathcal{T}$ with respect to $\mathbf{f_z}$ and $\mathbf{m_z}$ respectively. For each vertex, we store two numbers: $\epsilon_\mathbf{z}$ and $D_\mathbf{z}$ (as in **D**isplacement). These numbers represent the tolerance value of $\mathbf{z}$ and the distance between $\mathbf{f_z}$ and $\mathbf{m_z}$ respectively.

Pre-computations. Compute the Delaunay triangulation $\mathcal{T}$ of the initial set of points Ω, and for each vertex, let $\epsilon_{\mathbf{z}} = \epsilon(\mathbf{z})$ and $D_{\mathbf{z}} = 0$. The *updating algorithm* performs as follows for each vertex displacement:

```
Input: A triangulation $\mathcal{T}$ after the pre-computations, a vertex $\mathbf{z}$ of $\mathcal{T}$ and its new location $\mathbf{p}$.
Output: $\mathcal{T}$ updated after the relocation of $\mathbf{z}$.
begin
    $(\mathbf{m_z}, D_\mathbf{z}) \leftarrow (\mathbf{p}, dist(\mathbf{f_z}, \mathbf{p}))$ ;
    if $D_\mathbf{z} < \epsilon_\mathbf{z}$ then we are done;
    else
        insert $\mathbf{z}$ on a queue $\mathcal{Q}$;
        while $\mathcal{Q}$ is not empty do
            let $\mathbf{h}$ be the head of $\mathcal{Q}$;
            $(\mathbf{f_h}, \epsilon_\mathbf{h}, D_\mathbf{h}) \leftarrow (\mathbf{m_h}, \infty, 0)$ ;
            move $\mathbf{h}$ with placement algorithm;
            foreach new created bi-cells $\mathcal{B}$ do
                $\epsilon' \leftarrow$ half the width of the standard delimiter of $\mathcal{B}$ ;
                foreach vertex $\mathbf{w} \in \mathcal{B}$ do
                    if $\epsilon_\mathbf{w} > \epsilon'$ then
                        $\epsilon_\mathbf{w} \leftarrow \epsilon'$;
                        if $\epsilon_\mathbf{w} < D_\mathbf{w}$ then insert $\mathbf{w}$ into $\mathcal{Q}$;
                    end
                end
            end
        end
    end
end
```

Algorithm 1: *tolerance algorithm*

The algorithm is shown to terminate as each processes vertex $\mathbf{z}$ gets a new displacement value $D_\mathbf{z}$ equals to 0 and thus smaller or equal to $\epsilon_\mathbf{z}$. At the end of this algorithm, all vertices are guaranteed to have their $D_\mathbf{z}$ smaller or equal to their $\epsilon_\mathbf{z}$. In such a situation, from Property 1, $\mathcal{T}_\mathbf{m}$ is the Delaunay triangulation of the moving positions. The tolerance algorithm has the same complexity as the placement algorithm. If all points move, the total number of calls to the placement algorithm is guaranteed to be smaller than $O(n)$.

The second algorithm takes into account the safe region of a vertex. It will denoted by *safe region algorithm*, and it works as follow:

Data structure. Consider a triangulation $\mathcal{T}$, where for each vertex $\mathbf{z} \in \mathcal{T}$ we associate two points: $\mathbf{f_z}$ and $\mathbf{m_z}$, as in the tolerance algorithm. For each bi-cell, we will associate a point $\mathbf{c}_\mathcal{B}$ and a value $r_\mathcal{B} > 0$, representing the center and the radius of its standard delimiter respectively.

Pre-computations. Compute the Delaunay triangulation $\mathcal{T}$ of the initial set of points Ω, and for each bi-cell compute its standard delimiter.

Consider the following sub-routine to check whether a point $\mathbf{p}$ is inside the safe region of a vertex $\mathbf{z} \in \mathcal{T}$:

```
Input: A point $\mathbf{p}$ and a vertex $\mathbf{z} \in \mathcal{T}$.
Output: True, if $\mathbf{p}$ is inside the safe region of $\mathbf{z}$, and false otherwise.
begin
    foreach bi-cell $\mathcal{B}$ containing $\mathbf{z}$ do
        if {$\mathbf{m_z}$ belongs to the interior facet of $\mathcal{B}$ then
            if $dist(\mathbf{m_z}, \mathbf{c}_\mathcal{B}) > r_\mathcal{B}$ then
                return false ;
            end
        else
            if $dist(\mathbf{m_z}, \mathbf{c}_\mathcal{B}) < r_\mathcal{B}$ then
                return false ;
            end
        end
    end
    return true;
end
```

The *updating algorithm* performs as follows for each vertex displacement:

```
Input: A triangulation $\mathcal{T}$ after the pre-computations, a vertex $\mathbf{z}$ of $\mathcal{T}$ and its new location $\mathbf{p}$.
Output: $\mathcal{T}$ updated after the displacement of $\mathbf{z}$.
begin
    $\mathbf{m_z} \leftarrow \mathbf{p}$;
    if $\mathbf{m_z}$ is inside the safe region of $\mathbf{z}$ then
        we are done;
    else
        insert $\mathbf{z}$ on a queue $\mathcal{Q}$;
        while $\mathcal{Q}$ is not empty do
            let $\mathbf{h}$ be the head of $\mathcal{Q}$;
            $\mathbf{f_h} \leftarrow \mathbf{m_h}$ ;
            move $\mathbf{h}$ with placement algorithm;
            foreach new created bi-cells $\mathcal{B}$ do
                compute and store $\mathbf{c}_\mathcal{B}$ and $r_\mathcal{B}$ in respect to $\mathcal{T}_\mathbf{f}$ ;
                foreach vertex $\mathbf{w} \in \mathcal{B}$ do
                    if $\mathbf{m_w}$ belongs to the interior facet of $\mathcal{B}$ then
                        if $dist(\mathbf{m_z}, \mathbf{c}_\mathcal{B}) > r_\mathcal{B}$ then
                            insert $\mathbf{w}$ into $\mathcal{Q}$;
                        end
                    else
                        if $dist(\mathbf{m_z}, \mathbf{c}_\mathcal{B}) < r_\mathcal{B}$ then
                            insert $\mathbf{w}$ into $\mathcal{Q}$;
                        end
                    end
                end
            end
        end
    end
end
```

Algorithm 2: *safe region algorithm*

Analogously, from Property 1, at the end of the algorithm, $\mathcal{T}_\mathbf{m}$ is Delaunay. We can expect that while the tolerance algorithm take shorter time to verify if a point $\mathbf{m_z}$ is inside its tolerance region, the safe region algorithm accepts bigger vertex displacements without calling placement.

If we relocate the vertices with a convergent scheme, we can thus run rebuilding for the first few timestamps until the points are more or less stable, and then run the algorithms described in this section. Also, a natural idea is to replace the region test on the aforementioned algorithm with the combination of both; in the case of failure of the tolerance test, we call the safety test. Section 5.2 shows the behavior of those algorithms with more details.

Another point concerns robustness issues. Computing the tolerance values using floating point computation may, in some special configuration, yields to rounding errors and wrong evaluation of ϵ_z. The algorithm ensures that T_f, the embedding at fixed position, is always correct while the embedding T_m at moving positions may be incorrect. Having a certified correct Delaunay triangulation for T_m can be ensured by using certified lower bound of ϵ_z as described in Section 4.

4 Exactness and static filtering

4.1 Floating Point Arithmetic

Due to robustness issues of the algorithms described in Section 3, several questions on the number type usage arise. For example, if D is stored using a finite precision floating point number type, one could find an example of exact D value such that its computation using floating point arithmetic would lead to a wrong comparison result with a tolerance value due to rounding errors. Conversely, if we compute them using an exact number type [20], it will lead to a prohibitive time overhead. Even if those values are computed as an interval number type [8], which is significantly faster than exact number type, the negative impact on the execution time remains significant. We came up with a solution which uses the concept of *static filtering*.

Although, the difference between the real value and the computed value of an arithmetic expression, the *absolute error*, depends on the input, we may find ways such that it can be upper-bounded in such a way that it depends less or even not at all on the input, but just on the arithmetic expression. When this dependence can be solved by assuming that each input is bounded by a constant L, and so needing only to compute one upper-bound for the whole expression, we call this *static filtering*. In the other hand, when, at each intermediate computation, the error of the expression is evaluated, we call this *dynamic filtering* (i.e. interval arithmetics). And, in this case, several error computations are required. Of course, as near as you are from static filtering, less is the number of computations required to certify your final expression. As we will see, divisions and square root operations require computing intermediate error expressions in order to have certified error bounds.

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely-used standard for floating-point computation, and is followed by many CPU and FPU implementations. The standard defines formats for representing floating-point numbers and special values together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions. The default rounding mode is the *"Round to Nearest"* which, naturally, consists of rounding to the nearest representable value, and if the number fails midway, it is rounded to the nearest value with an even least significant bit. Exact rounding is required for the four basic operations $(+, -, \times, /)$ and for square root operation as well. This rounding mode leads to absolute errors smaller than $ulp(x)/2$, with $ulp(x)$ being the *units in the last place* of x (for standard `double` it is 2^{-53}). For an easy-to-read survey on floating point arithmetic fundamentals, please refer to [21].

We will call $f(x)$ the floating point computed for the expression x. We compute α_x such that $|f(x)-x| \leq \alpha_x$. Since we are comparing x with a given value as part of the algorithm described in Section 3, we can simply compare the given value with $x + \alpha_x$ instead. The new value is no longer exact, and is a certified upper-bound for the exact value.

If $f(a)$ and $f(b)$ are the resulting floating point evaluations with their absolute error α_a and α_b respectively, then the value of α_{a+b} or α_{a-b} is at most

$$|f(a \pm b) - (a \pm b)| \leq \alpha_a + \alpha_b + |a \pm b| \frac{ulp(1)}{2}, \tag{5}$$

Besides, the maximum value of $\alpha_{a \times b}$ is at most

$$|f(a \times b) - (a \times b)| \leq \alpha_a |b| + \alpha_b |a| + |a||b| \frac{ulp(1)}{2}. \tag{6}$$

Unfortunately, for the division and the square root operation, we cannot bound its floating point value because the inverse of the divisor (or square root of the divisor) cannot be bounded *a priori*. However, for the division operation, we can take the corresponding divisor value on-the-fly and test it against some previously set conditions, acting more like a semi-static filtering. Then, the maximum value of $\alpha_{\frac{1}{a}}$, assuming $A = |f(a) - \alpha_a| \leq |a|$ and $\alpha_a < A \leq |f(a)|$, is at most:

$$\left| f\left(\frac{1}{a}\right) - \left(\frac{1}{a}\right) \right| \leq \frac{\alpha_a}{A^2} + \left|\frac{1}{A}\right| \frac{ulp(1)}{2}. \tag{7}$$

The equation above depends on A and α_a. If the above-mentioned condition $\alpha_a < A \leq |f(a)|$ does not hold, the filter fails. Moreover, if $f(a)$ is a non-degenerated floating point expression with $a \geq 0$, then the value of $\alpha_{\sqrt{a}}$ is at most

$$|f(\sqrt{a}) - \sqrt{a}| \leq \frac{\alpha_a}{2\sqrt{A}} + \sqrt{a} \frac{ulp(1)}{2}. \tag{8}$$

Recall that $|f(a)| - \alpha_a \leq |a| \leq |f(a)| + \alpha_a$, and so $|a|^{-1/2} \leq (|f(a)| - \alpha_a)^{-1/2}$. Than we can calculate the absolute errors of the square root operations on-the-fly wherever it appears.

Pure static filtering is at least twice as fast as interval arithmetic [12, 29].

4.2 Tolerance Certified Computation

We could find an *a priori* error expression for the squared distance between two points, we will call α_D, since it contains any divisions or square root operations, whereas for the tolerance error expression, we will call α_ϵ, we made it as less dependent on the input as possible. Following, we have the α_D and α_ϵ computations for two and three dimensions.

Two dimensions. Our code for the computation of the distance between two points:

```
/** given two vertices
 *  v_i (x_i, y_i), i = 1, 2,
 *  compute the squared distance between the two vertices.
 */
squared_distance(v_1, v_2)
{
  a := x_1 - x_2;
  b := y_1 - y_2;
  c := a * a;
  d := b * b;
  e := c + d;
  return e;
}
```

We will call $\mu = ulp(1)/2$. Let τ_z and α_z be a bound for the maximum absolute value and error of the variables z respectively, and $L = 2^k$ an upper-bound on the inputs. When two variables z and z' have the same bounds, we denote these common bounds by $\alpha_{\{z,z'\}}$ and $\tau_{\{z,z'\}}$. Remind from Section 4 the error bound formulas for the following operations: $+, -, \times, /, \sqrt{}$. We assume that the inputs are exact.

And so:

variables	τ	α
$\{a, b\}$	$\leq 2L$	$\leq 2L\mu$
$\{c, d\}$	$\leq 4L^2$	$\leq 2\alpha_{\{a,b\}}\tau_{\{a,b\}} + \tau^2_{\{a,b\}}\mu \leq 8L^2\mu + 4L^2\mu \leq 12L^2\mu$
$\{e\}$	$\leq 8L^2$	$\leq 2\alpha_{\{c,d\}} + 2\tau_{\{c,d\}}\mu \leq 24L^2\mu + 8L^2\mu \leq 32L^2\mu$

Our code for the computation of the tolerance:

```
/** given a Delaunay triangulation's bi-face B with vertices
 *  v_i (x_i, y_i), i = 1, 2, 3, 4,
 *  compute the square of half the width of the annuli
 *  where one circle passes through v_1, v_4 and the other through v_2, v_3
 */
squared_half_annulus_width(v_1, v_2, v_3, v_4)
{
  /* One bisector */
  a_1 := 2 * (x_2 - x_4);
  b_1 := 2 * (y_2 - y_4);
  c_1 := ((x_4)^2 + (y_4)^2) - ((x_2)^2 + (y_2)^2);
  /* The other bisector */
  a_2 := 2 * (x_1 - x_3);
  b_2 := 2 * (y_1 - y_3);
  c_2 := ((x_3)^2 + (y_3)^2) - ((x_1)^2 + (y_1)^2);
  /* The intersection between them */
  d := (a_1 * b_2) - (b_1 * a_2);
  n_x := (b_1 * c_2) - (b_2 * c_1);
  n_y := (a_2 * c_1) - (a_1 * c_2);
  /* The coordinates of the center of the annulus */
  den := 1 / d;
  x_annulus := n_x * den;
  y_annulus := n_y * den;
  /* The smallest annulus squared radius */
  r_2 := (x_2 - x_annulus)^2 + (y_2 - y_annulus)^2;
  /* The biggest annulus squared radius */
  R_2 := (x_3 - x_annulus)^2 + (y_3 - y_annulus)^2;
  /* The squared in-circle tolerance */
  w := (r_2 + R_2 - 2 * sqrt(r_2 * R_2))/4;
25:  return w;
26:}
```

Analogously, we compute α_ϵ as follow:

variables	τ	α
$\{a_1, b_1, a_2, b_2\}$	$\leq 4L$	$\leq 4L\mu$
$\{c_1, c_2\}$	$\leq 4L^2$	$\leq 12L^2\mu$
$\{d\}$	$\leq 32L^2$	$\leq 4\alpha_{\{a_1,\ldots\}}\tau_{\{a_1,\ldots\}} + 4\tau^2_{\{a_1,\ldots\}}\mu \leq 64L^2\mu + 64L^2\mu \leq 128L^2\mu$
$\{n_x, n_y\}$	$\leq 32L^3$	$\leq 2(\tau_{\{c_1,c_2\}}\alpha_{\{a_1,\ldots\}} + \tau_{\{a_1,\ldots\}}\alpha_{\{c_1,c_2\}}) + 4\tau_{\{a_1,\ldots\}}\tau_{\{c_1,c_2\}}\mu \leq 2(16L^3\mu + 48L^3\mu) + 64L^3\mu \leq 192L^3\mu$

Since we will use d as divisor, we will take $U = f(d) - \alpha_d$, the actual computed d. And we will be able to find, with Equation 7 upper-bounds for the error of den. By doing that, we skip from the pure static filtering approach to a semi-static approach, since α_{den} is computed on-the-fly.

$\alpha_{den} \leq \frac{128L^2\mu}{U^2} + \left(\frac{1}{U}\right)\mu$, $\tau_{den} \leq \frac{1}{U}$.

From the seventeenth line of the α_ϵ computation code on, as the number of terms grows fast thus impacting the performance, we will use the equations in Subsection 4.1 to dynamically evaluate an upper-bound for the absolute error.

Three dimensions. Our chosen code for α_D:

```
/** given two vertices
 *  v_i (x_i, y_i, z_i), i = 1, 2,
 *  compute the squared distance between the two vertices.
 */
1: squared_distance(v_1, v_2)
2: {
3:   a := x_1 - x_2;
4:   b := y_1 - y_2;
5:   c := z_1 - z_2;
6:   d := a * a;
7:   e := b * b;
8:   f := c * c;
9:   g := d + e + f;
10:  return g;
11:}
```

variables	τ	α
$\{a, b, c\}$	$\leq 2L$	$\leq 2L\mu$
$\{d, e, f\}$	$\leq 4L^2$	$\leq 2\alpha_{\{a,b,c\}}\tau_{\{a,b,c\}} + \tau^2_{\{a,b,c\}}\mu \leq 8L^2\mu + 4L^2\mu \leq 12L^2\mu$
$\{g\}$	$\leq 12L^2$	$\leq 3\alpha_{\{d,e,f\}} + 5\tau_{\{d,e,f\}}\mu \leq 36L^2\mu + 20L^2\mu \leq 56L^2\mu$

Our code for the computation of the tolerance:

```
/** given a Delaunay triangulation's bi-face B with vertices
 *  v_i (x_i, y_i, z_i), i = 1, 2, 3, 4, 5
 *  compute the square of half the width of the annuli
 *  where one circle passes through v_1, v_5 and the other through v_2, v_3, v_4
 */
1: squared_half_annulus_width(v_1, v_2, v_3, v_4, v_5)
2: {
3:   double sa = (x_1)^2 + (y_1)^2 + (z_1)^2;
4:   double sb = (x_2)^2 + (y_2)^2 + (z_2)^2;
5:   double sc = (x_3)^2 + (y_3)^2 + (z_3)^2;
6:   double sd = (x_4)^2 + (y_4)^2 + (z_4)^2;
7:   double se = (x_5)^2 + (y_5)^2 + (z_5)^2;
8:   /* One bisector with d_1 := -d_1 */
```

```
  double a_1 = 2*(x_1 - x_5);
  double b_1 = 2*(y_1 - y_5);
  double c_1 = 2*(z_1 - z_5);
  double d_1 = sa - se;
  /* The second bisector with d_2 := -d_2  */
  double a_2 = 2*(x_2 - x_4);
  double b_2 = 2*(y_2 - y_4);
  double c_2 = 2*(z_2 - z_4);
  double d_2 = sb - sd;
  /* The third bisector with d_3 := -d_3  */
  double a_3 = 2*(x_4 - x_3);
  double b_3 = 2*(y_4 - y_3);
  double c_3 = 2*(z_4 - z_3);
  double d_3 = sd - sc;
  /* The intersection between them */
  double m01 = a_1*b_2 - a_2*b_1;
  double m02 = a_1*b_3 - a_3*b_1;
  double m12 = a_2*b_3 - a_3*b_2;
  double x01 = b_1*c_2 - b_2*c_1;
  double x02 = b_1*c_3 - b_3*c_1;
  double x12 = b_2*c_3 - b_3*c_2;
  double y01 = c_1*a_2 - c_2*a_1;
  double y02 = c_1*a_3 - c_3*a_1;
  double y12 = c_2*a_3 - c_3*a_2;
  double z01 = a_1*b_2 - a_2*b_1;
  double z02 = a_1*b_3 - a_3*b_1;
  double z12 = a_2*b_3 - a_3*b_2;
  double d = m01*c_3 - m02*c_2 + m12*c_1;
  double x012 = x01*d_3 - x02*d_2 + x12*d_1;
  double y012 = y01*d_3 - y02*d_2 + y12*d_1;
  double z012 = z01*d_3 - z02*d_2 + z12*d_1;
  double den = 1/d;
  /* The coordinates of the center of the annulus */
  double x_annulus = x012 * den;
  double y_annulus = y012 * den;
  double z_annulus = z012 * den;
  /* The smallest annulus squared radius */
  r_2 := (x_2 - x_annulus)^2 + (y_2 - y_annulus)^2 + (z_2 - z_annulus)^2;
  /* The biggest annulus squared radius */
  R_2 := (x_1 - x_annulus)^2 + (y_1 - y_annulus)^2 + (z_1 - z_annulus)^2;
  /* The squared in-circle tolerance */
  w := (r_2 + R_2 - 2 * sqrt(r_2 * R_2))/4;
  return w;
}
```

variables	τ	α
$\{sa, sb, sc, sd, se\}$	$\leq 3L^2$	$\leq 8L^2\mu$
$\{a_1, b_1, c_1, a_2, b_2, c_2, a_3, b_3, c_3\}$	$\leq 4L$	$\leq 4L\mu$
$\{d_1, d_2, d_3\}$	$\leq 6L^2$	$\leq 2\alpha_{\{sa,\ldots\}} + 2\tau_{\{sa,\ldots\}}\mu \leq 16L^2\mu + 6L^2\mu \leq 22L^2\mu$
$\{m01, m02, m12, x01, x02, x12, y01, y02, y12, z01, z02, z12\}$	$\leq 32L^2$	$\leq 4\alpha_{\{a_1,\ldots\}}\tau_{\{a_1,\ldots\}} + 4{a_1}^2\mu \leq 64L^2 + 64L^2 \leq 128L^2\mu$
$\{d\}$	$\leq 384L^3$	$\leq 3(\alpha_{\{m01,\ldots\}}\tau_{c_3} + \alpha\{a_1,\ldots\}\tau_{\{m01,\ldots\}}) + 8\tau_{\{a_1,\ldots\}}\tau_{\{m01,\ldots\}}\mu$ $\leq 3(512L^3\mu + 128L^3\mu) + 1,024L^3\mu \leq 2,944L^3\mu$
$\{x012, y012, z012\}$	$\leq 576L^4$	$\leq 3(\alpha_{\{m01,\ldots\}}\tau_{\{d_1,\ldots\}} + \alpha\{d_1,\ldots\}\tau_{\{m01,\ldots\}}) + 8\tau_{\{d_1,\ldots\}}\tau_{\{m01,\ldots\}}\mu$ $\leq 3(768L^4\mu + 576L^4\mu) + 1,536L^4\mu \leq 5,568L^4\mu$

Again, since we will use d as divisor, we will take $U = f(d) - \alpha_d$, the actual computed d. And we will be able to find, with Equation 7 upper-bounds for the error of *den*. By doing that, we skip from the pure static filtering approach to a semi-static approach, since α_{den} is computed on-the-fly.

$\alpha_{den} \leq \frac{2{,}944L^3\mu}{U^2} + \left(\frac{1}{U}\right)\mu$, $\tau_{den} \leq \frac{1}{U}$.

From the forty-second line of the α_ϵ computation code on, as the number of terms grows fast thus impacting the performance, we will use the equations in Subsection 4.1 to dynamically evaluate an upper-bound for the absolute error.

5 Experimental Results

This section investigates, through several experiments, the size of the vertex tolerances and the width of the standard delimiter generators in two and three dimensions. We discuss the performance of the algorithms described in Section 3.3 on several data sets.

We used for our experiments a Pentium 4 at 2.5 GHz with 1GB of memory, running Linux (version 2.6.23 Kernel). The compiler used is g++4.1.2; all configurations being compiled with -DNDEBUG -O2 flags[1]. We are using *CGAL* 3.3.1 along with an *exact predicate inexact construction* kernel [9, 18].

5.1 Lloyd Iterations

A centroidal Voronoi tessellation is a Voronoi tessellation [4, 30, 5] whose generating points are the centroids (centers of mass) of the corresponding Voronoi regions [15]. Applications of centroidal Voronoi tessellation include image compression, quadrature, and cellular biology, to name a few. There are several approaches to determine centroidal Voronoi tessellations, classified as either probabilistic [23, 26] or deterministic [28, 22, 25, 40]. One deterministic method is the well-known *Lloyd's iterations* [28, 35, 14, 31]. Given a set of points, Lloyd's iterations optimize their placement by moving them to the centroid of their Voronoi region with respect to a given density function, up to convergence.

For each iteration of Lloyd's method, we must recompute the Delaunay triangulation of the points. Each iteration can be considered as a distinct timestamp.

In two dimensions, we consider four different density functions: $\rho_1 = 1$, $\rho_2 = x^2 + y^2$, $\rho_3 = x^2$ and $\rho_4 = \sin^2\left(\sqrt{x^2+y^2}\right)$. And we run the Lloyd's iterations to obtain centroidal Voronoi tessellations according to them.

Hereafter, we denote the set of the width of the standard delimiter generator of the bi-cells by $\mathcal{W}$, and the set of the vertex tolerances by $\epsilon(V)$. The average of a set S of numbers is denoted by $avg(S)$.

Our experiments show that during the process, while the average displacement is going to zero, $avg(\mathcal{W})$ and $avg(\epsilon(V))$ converge to around $33-40\%$ and $5-10\%$ of the average point density respectively (see Figure 4). We run the Lloyd's iterations during 1000 iterations which is necessary to generate satisfactory results (see Figure 3).

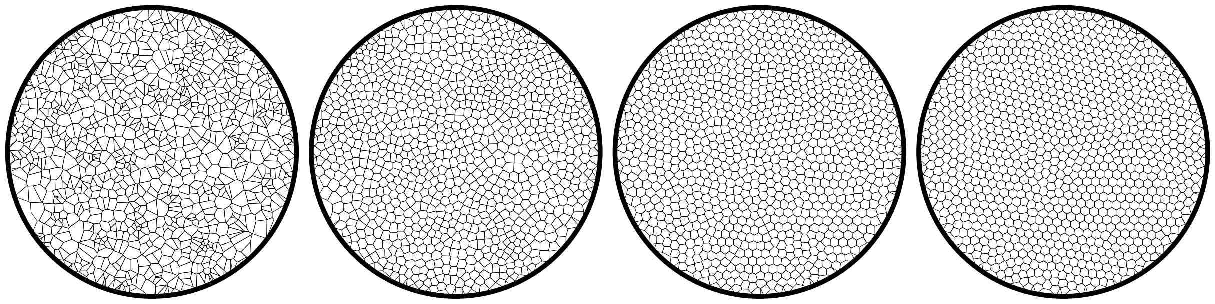
In three dimensions, we run the Lloyd's iterations on a point set of about 13.000 points in a ball. This experiment is referenced to as *slloyd*. The average tolerance of bi-cells remains about 30% of the point density, but due to the higher degree of a vertex in three dimensions, the average tolerance of the vertices converge to a much smaller value (of about 1%).

We also run a dual version of the Lloyd's algorithm, first called *optimal Delaunay triangulation* (*ODT*), which is shown to generate fewer *slivers* (flat tetrahedra, which impacts negatively on the stability of computations in simulations) [27, 39]. As this algorithm requires moving the points one by one in sequence, we cannot use the rebuilding scheme. We run it on a set of about 13.000 points in a sphere (*snodt*) and on a surface mesh of a human body of about 8.000 points (*man*).

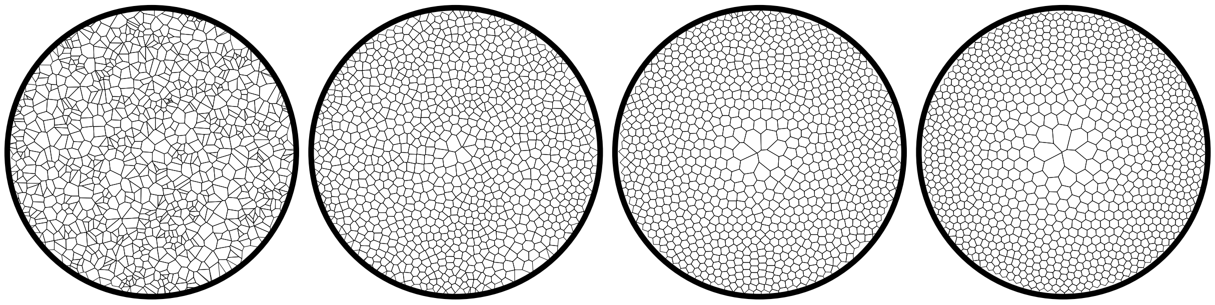
Figure 5 shows *man* at different iterations of the process. Close-ups on the head visually indicates that 100 iterations are clearly not sufficient to get an high quality mesh (as confirmed by less visual quality measures).

Experiments show a good convergence of the average tolerance of bi-cells to a reasonable value and of the average tolerance of vertices to a smaller value, although these tolerances converge slower than in two dimensions (see Figure 6).

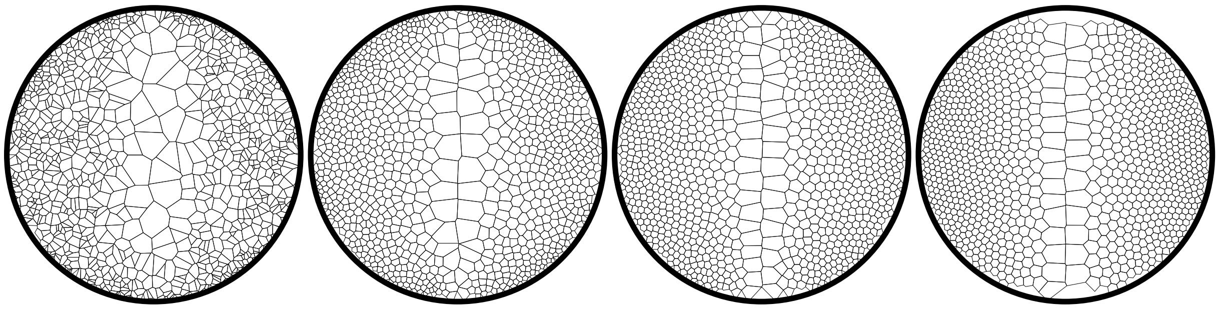
Lloyd iterations add some structure to the points as its way to convergence, increasing the tolerances compared to a random situation. As a point of comparison we present in Figure 7 some distributions and



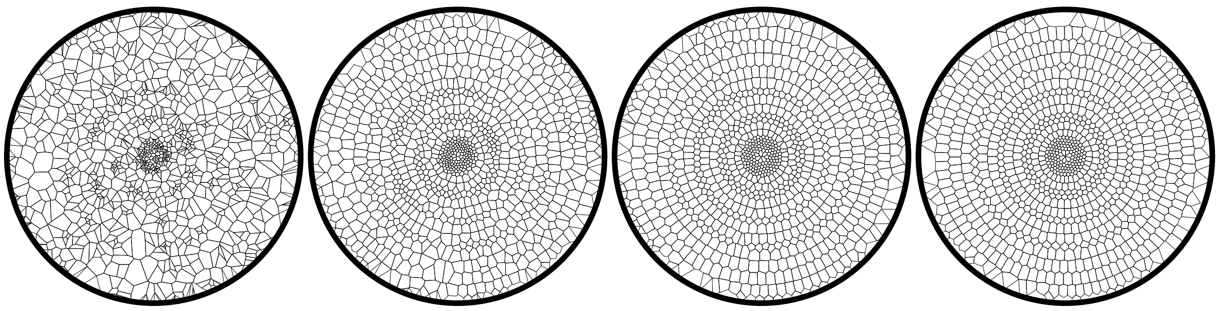
(a) $\rho = 1$



(b) $\rho = x^2 + y^2$



(c) $\rho = x^2$

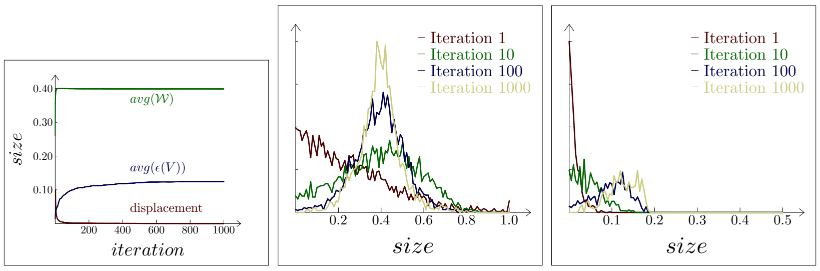


(d) $\rho = \sin^2\left(\sqrt{x^2+y^2}\right)$

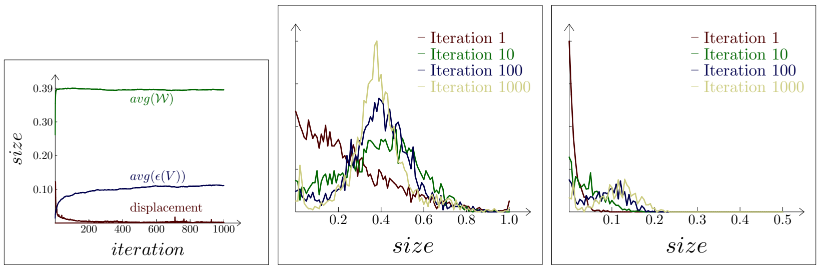
Figure 3: Each column correspond to the 1st, 10th, 100th and 1000th iteration.

experimental measures of average tolerances for random point distribution. We observe that the ratio between $avg(\mathcal{W})$ and $avg(\epsilon(V))$ fits the number of bi-cells per vertex.

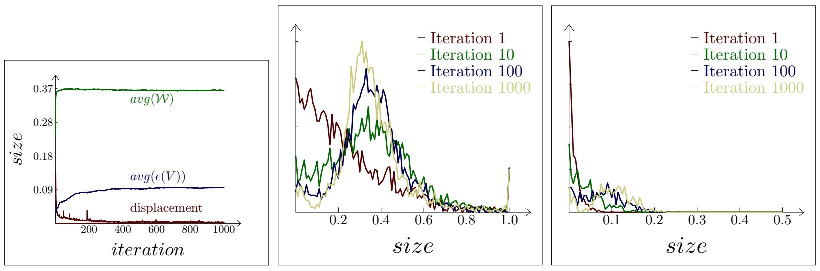
[1]release mode with compiler optimizations enabled.



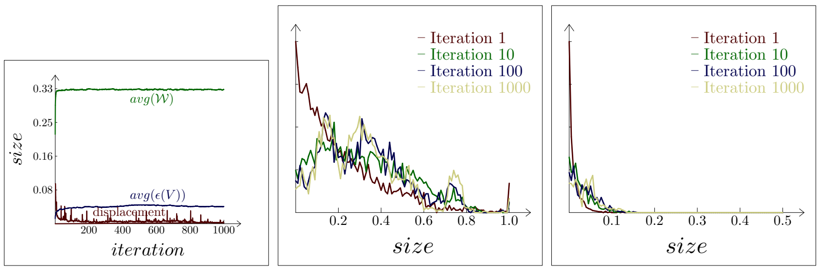
(a) $\rho = 1$



(b) $\rho = x^2 + y^2$



(c) $\rho = x^2$



(d) $\rho = \sin^2\left(\sqrt{x^2 + y^2}\right)$

Figure 4: The experiment consists of 1.000 points, sampled in a circle uniformly with their respective ρ, submitted to the Lloyd's method. The size of the circle is chosen such that there is one point per unity of area in average. In the first column, we have $avg(\mathcal{W})$ and $avg(\epsilon(V))$ versus the number of iterations. In the second and third column, we have $\mathcal{W}$ and $\epsilon(V)$ distributions on the iterations 1, 10, 100 and 1000 respectively.

5.2 Results

This section discusses the computation times behavior of rebuilding, placement and the tolerance algorithm for the input set introduced in previous section. We will denote the tolerance algorithm by T, the safe region algorithm by S, and the algorithm which combines S and T by S+T.

For a random point distribution and random displacements of length δ, Figure 8 shows how the computation times of the algorithms are related to the percentage of vertices moving inside their tolerance region. Typically most of the displacements do not produce any topological change on the triangulation, even if some of them are slightly outside their tolerance region.

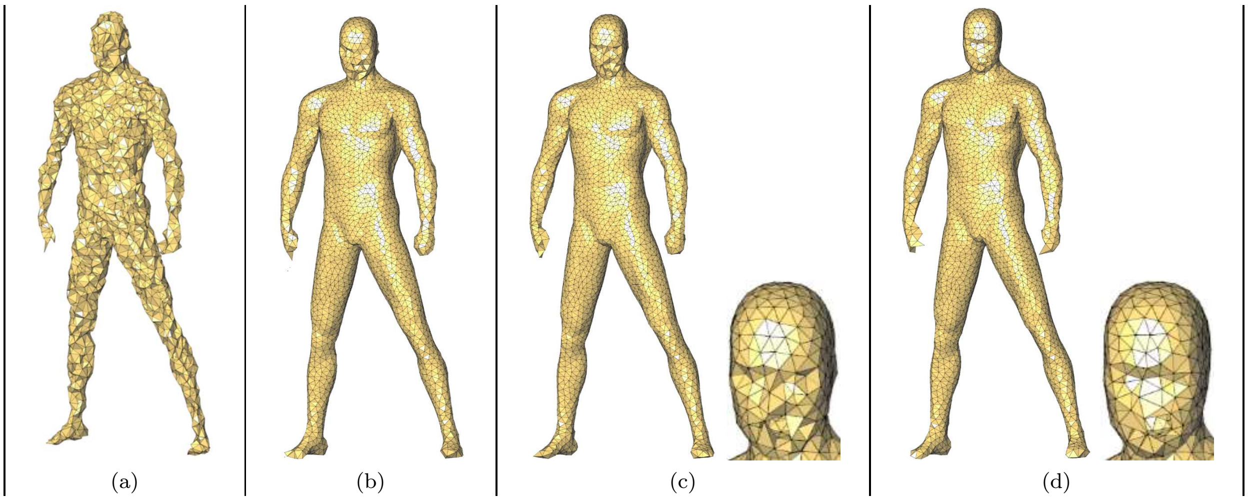
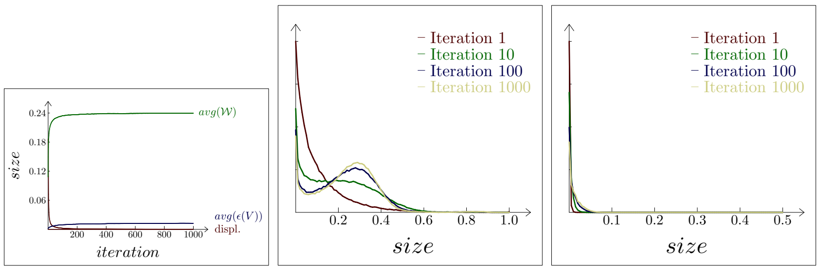
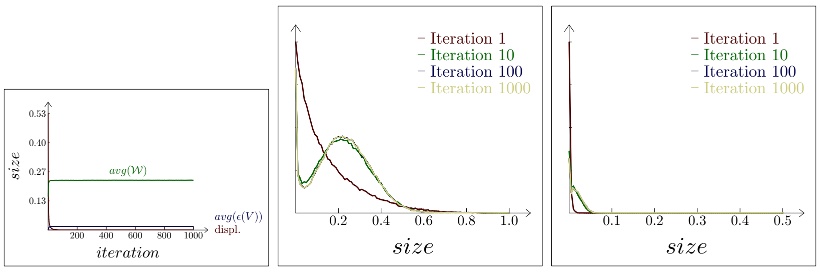


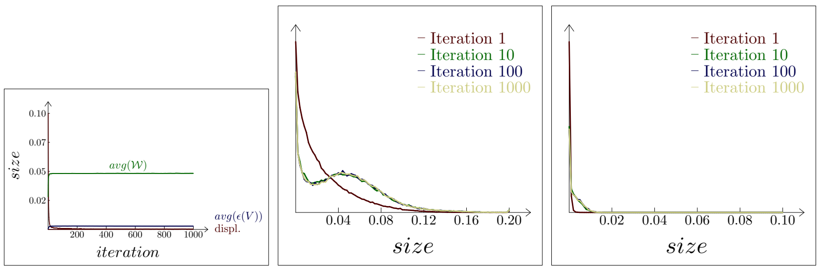
Figure 5: In (a) *man* initially; In (b), (c) and (d) *man* after 10, 100 and 1000 iterations respectively. In (c) and (d), we also have a zoom of the head. Here as well, we can notice the improvements on the Voronoi cells shape.



(a) Sphere with Lloyd's method

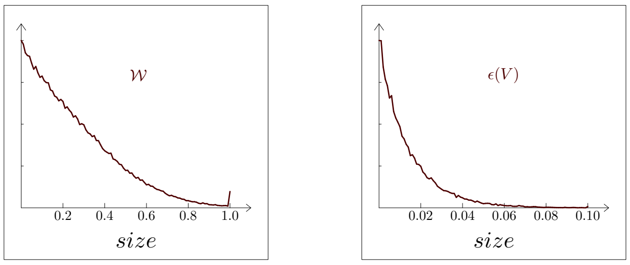


(b) Sphere with the modified Lloyd's method

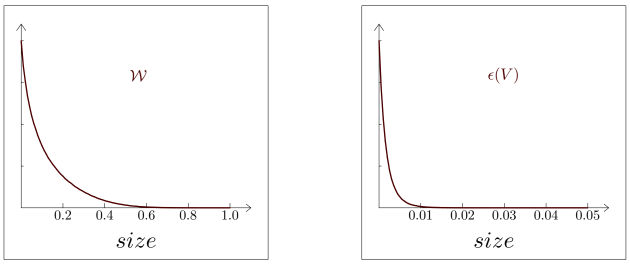


(c) *man*

Figure 6: In the first column, the corresponding $avg(\mathcal{W})$ and $avg(\epsilon(V))$ versus displacement size graph. In the second column, we have the distributions for $\mathcal{W}$ and $avg(\epsilon(V))$ on the iterations 1, 10, 100 and 1000 respectively. Remark: in (c), the size goes from 0 to 0.2 because points in *man* have a sparse distribution in the volume of reference.



(a) Two dimensions



(b) Three dimensions

$\rho = 1$	**2D**	**3D**
$avg(\mathcal{W})$	0.255	0.126
$avg(\epsilon(V))$	0.014	0.002
average max. annuli width at vertices	0.610	0.742
average number of adjacent bi-cells	12.0	66.3

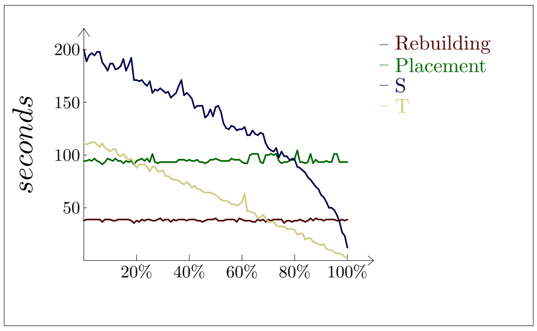
(c) Two dimensions vs. Three dimensions

Figure 7: In (a) and (b) we have the histograms in two and three dimensions of $\mathcal{W}$ and $\epsilon(V)$ respectively. Let *an adjacent bi-cell of a vertex* be any bi-cell of the triangulation containing that vertex. Then in table (c): *average max. annuli width at vertices* means the average, on all the vertices, of the biggest value among all the annuli width of the adjacent bi-cells of a vertex.

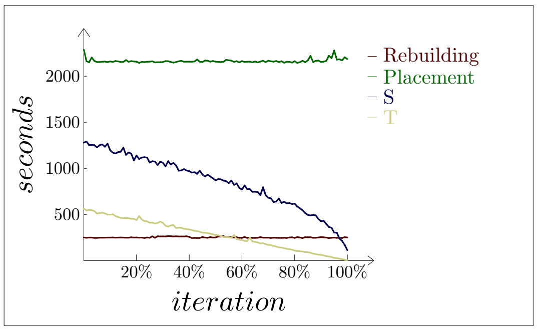
As shown by Figure 8, if displacements are small enough, and only around 25% of the vertices have displacement above tolerance in two dimensions (55% in three dimensions), then T is competitive with rebuilding. In three dimensions, the improvements for small displacements described in Section 3.2 are shown sufficient to outperform placement. Moreover, placement is twice slower and around nine time slower than rebuilding in two and three dimensions respectively. In extreme configurations, our algorithm outperforms rebuilding by a factor of 17 (125 in three dimensions). However those configurations are artificial and very unlikely to happen for real data sets. Algorithm S takes approximately 4 times (55 in three dimensions) more than T to check whether a vertex is inside its allowed region. Note that, from Figure 8, Algorithm S is not able to outperform rebuilding by more than a factor of three.

As just shown, the performance of T depends on the amount of displacements remaining inside the tolerance region. An idea of what this percentage represents in terms of distances can be shown within the context of random walk. Figure 9 shows that when displacement magnitudes are around 20% of $avg(\epsilon(V))$ of the input set, T is still able to outperform rebuilding in both two and three dimensions. If we refer to the numbers in Figure 7c, and take 20% of them, we can see that these displacement sizes represent around 1/350 of the unity of distance (1/2500 in three dimensions). In three dimensions, if all displacements magnitudes are lower or equal to $avg(\epsilon(V))$, Algorithm T is more than three time faster than placement.

We now discuss the performance of the algorithms for Lloyd iterations on all inputs described in Section 5.1, considering a thousand of iterations. For those inputs, we implement in addition a small variation of the tolerance algorithm, suggested in Section 3.3, which consists of rebuilding for the first few iterations, and, swapping to T. We denote this algorithm by $R{+}T$. The swapping criterion can be more or less heuristic, however, Figure 8 shows that when 75% of the vertices remain inside their tolerance (55% in three dimensions), the performance of Algorithm T is the same as rebuilding. This percentage can be used as a swapping criterion. Pragmatically, we

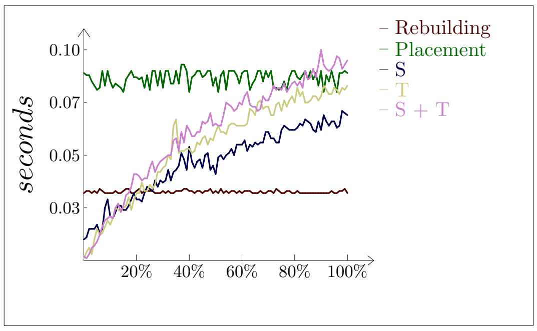


(a) Two dimensions

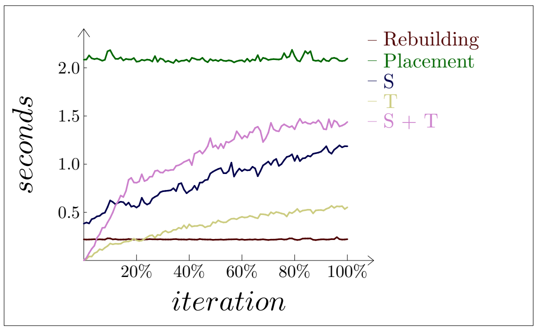


(b) Three dimensions

Figure 8: The experiment consists of 10.000 points in a circle (in two dimensions) or sphere (in three dimensions) and 1.000 iterations of small displacements. The x-axis represents the percentage of displacements size of a vertex smaller than its tolerance. Although placement has its own advantage, it remains very expensive in three dimensions.



(a) Two dimensions



(b) Three diemensions

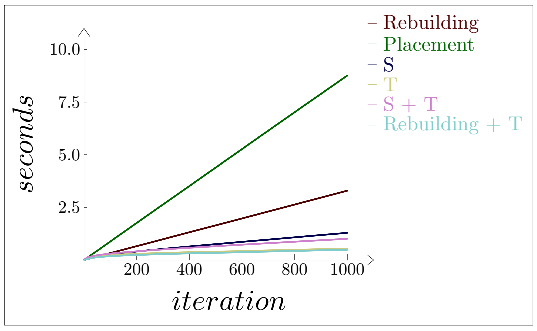
Figure 9: The experiment consists of 1.000 random points uniformly distributed in a circle (respectively sphere), such that there is on average one point per unity of area (respectively volume). Points are submitted to 10 iterations of random walk with δ equals to a percentage of $avg(\epsilon(V))$ of the input set (so this result can be applied for any other density value). The x-axis represents this percentage.

sample 40 random vertices at each group of four iterations, and we compute their tolerance. We then compare their tolerance with their displacement size, and check if at least 75% of those vertices have their tolerance larger than their displacement size. In this case, we swap the algorithms.

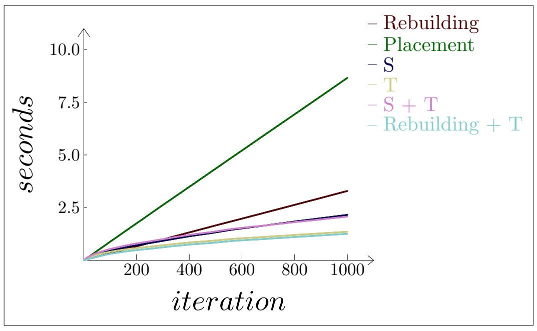
Figure 10 shows that, in two dimensions, all the algorithms S, T, S+T, rebuilding + T outperform both rebuilding and placement (see Table 1 for the precise speedup factors). In three dimensions, they outperform placement in any configuration. This enhancement is relevant for applications requiring to move vertices one at a time and for dynamic triangulations, which cannot be achieved by rebuilding (Section 3). However, to outperform rebuilding is harder in three dimensions, because the removal operation is even slower and the number of bi-cells containing a given point is five times larger than in two dimensions. In spite of that, T still outperforms rebuilding in synthetic and real data sets (*snodt* and *man*). T cannot perform with *slloyd*, as well as it does with the other inputs, because of the *slivers* produced by running a pure Lloyd iteration (tolerances are sensitive to *slivers*). In both two and three dimensions, when we go further on the number of iterations, Algorithm T becomes faster (see Table 1). As shown by Figure 5, the number of iterations highly impacts the quality of the final result. This result enables a novel possibility: Going further on the number of iterations.

In Table 1, we can clearly verify that the strategy of considering the safe region of vertices and its derivations is not worthwhile. Both the costs of updating the delimiters and checking whether a vertex is in a particular region or not, are more expansive than the tolerance based algorithms. If we look carefully at the behavior of Algorithm S+T in Figure 10, we can verify that Algorithm S+T almost does not improve Algorithm S. Therefore, in our input data, much of the vertices that escape its tolerance region, also escape its safe region. This fact reduces the interest of using safe regions.

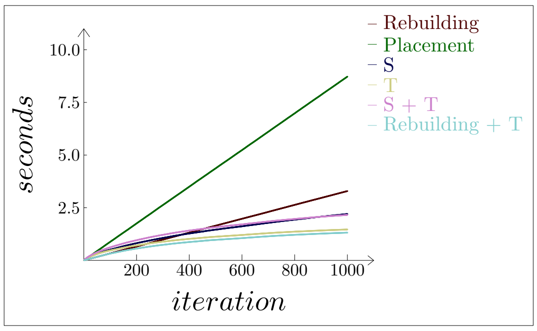
Finally, in two dimensions, the gain by combining rebuilding with Algorithm T is not significant. In three dimensions, it can lead to more or less good speed-ups on the original algorithm performance.



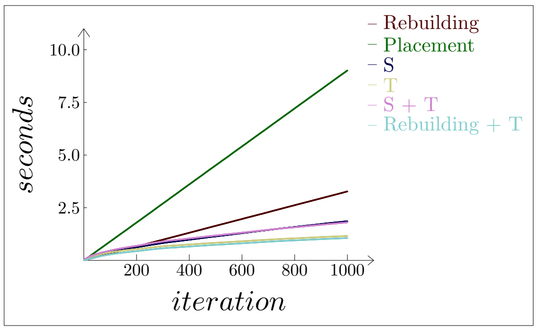
(a) $\rho = 1$



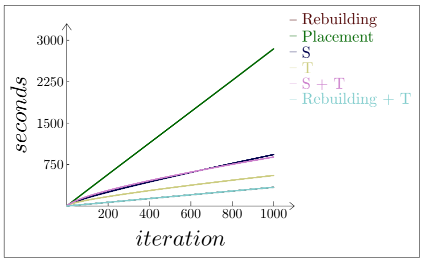
(b) $\rho = x^2 + y^2$



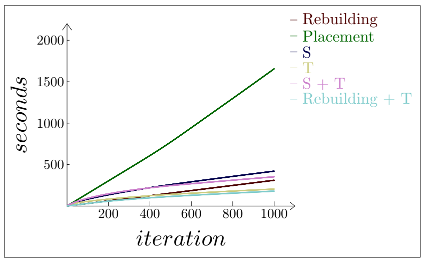
(c) $\rho = x^2$



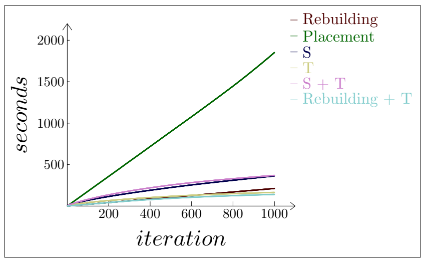
(d) $\rho = \sin^2\left(\sqrt{x^2+y^2}\right)$



(e) *slloyd*



(f) *snodt*



(g) *man*

Figure 10: The figure above represents the computation time vs. the number of iterations. We can see rebuilding, placement, Algorithm S, T, S+T and rebuilding + T against the input sets described in Section 5.1. In (e), we cannot see the performance of rebuilding, because it outperforms algorithm T everywhere. When that happens, rebuilding + T is the rebuilding algorithm itself.

6 Concluding Remarks

In this paper, we deal with the problem of updating Delaunay triangulations for moving points. We introduce notion of the tolerance and safe region of a vertex in this context. We use these concepts to develop a dynamic algorithm that works as a filter, avoiding unnecessary insert and remove operations. We conduct several experiments to showcase the behavior of the algorithm in a variety of data sets. Finally, we end up with

Table 1: The numbers below represent the speedup factor of an algorithm with respect to rebuilding, for a given input. Each number greater than one will be in boldface. The raws in green represent static algorithms and the raw in red represent dynamic algorithms.

	$\rho = 1$	$\rho = x^2 + y^2$	$\rho = x^2$	$\rho = \sin^2\left(\sqrt{x^2+y^2}\right)$	*sloyd*	*snodt*	*man*
rebuilding	1,00	1,00	1,00	1,00	1,00	1,00	1,00
placement	0,38	0,38	0,38	0,36	0,12	0,19	1,00
S	**2,55**	**1,53**	**1,50**	**1,76**	0,36	0,73	0,58
T	**6,10**	**2,43**	**2,24**	**2,82**	0,61	**1,52**	**1,29**
S+T	**3,27**	**1,56**	**1,52**	**1,81**	0,38	0,88	0,57
rebuilding+T	**6,80**	**2,62**	**2,50**	**3,08**	1,00	**1,72**	**1,51**

Table 2: The numbers below represent the speedup factor of an algorithm with respect to rebuilding running from the 500th to the 1000th iteration, for a given input. Each number greater than one will be in boldface. The raws in green represent static algorithms and the raw in red represent dynamic algorithms.

	$\rho = 1$	$\rho = x^2 + y^2$	$\rho = x^2$	$\rho = \sin^2\left(\sqrt{x^2+y^2}\right)$	*sloyd*	*snodt*	*man*
rebuilding	1,00	1,00	1,00	1,00	1,00	1,00	1,00
placement	0,38	0,38	0,37	0,36	0,12	0,18	0,15
T	**11,50**	**3,97**	**4,79**	**5.03**	0,76	**2.42**	**2.25**

an algorithm suitable when the magnitude of the displacement keeps decreasing while the tolerances keep increasing. Such configurations translate into convergent schemes, e.g. Lloyd's iterations itself. In this case, in two dimensions, the algorithm presented performs up to six times faster than *rebuilding*. In three dimensions, rebuilding the whole triangulation at each timestamp can be faster than our algorithm, but our solution is dynamic, and outperforms previous dynamic solutions. Such a dynamic algorithm is required for some meshing techniques.

Future works include extending this algorithm to *constrained* and *regular* triangulations. Another direction of research is to elaborate upon optimization schemes for the updating tolerances operation, e.g. finding lower bounds for the tolerances, which are faster to compute. Also, finding several different balls included in the safe region, which are bigger than the tolerance region. Furthermore, heuristics to attribute an appropriate order to the input moving points set, aiming at avoiding unnecessary tolerance updates, may enhance the performances in several situations. Displacement-adaptive delimiters could also be considered, as the method mainly outfits convergent relocation sequences.

Acknowledgements

The authors wish to thank the *ANR Triangles*[2], contract number `ANR-07-BLAN-0319`, and region PACA for their support. Also, the authors thank Pierre Alliez and Jane Tournois for the three-dimensional data set.

References

[1] G. Albers, Leonidas J. Guibas, Joseph S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points. *Internat. J. Comput. Geom. Appl.*, 8:365–380, 1998.

[2] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. *ACM Transactions on Graphics*, 24:617–625, 2005. SIGGRAPH '2005 Conference Proceedings.

[3] N. Amenta, M. Bern, and D. Eppstein. Optimal point placement for mesh smoothing. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 528–537, January 1997.

[4] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.

[2]http://www-sop.inria.fr/geometrica/collaborations/triangles/

[5] David Avis and Binay K. Bhattacharya. Algorithms for computing d-dimensional Voronoi diagrams and their duals. In Franco P. Preparata, editor, *Computational Geometry*, volume 1 of *Adv. Comput. Res.*, pages 159–180. JAI Press, Greenwich, Conn., 1983.

[6] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, and Mariette Yvinec. Triangulations in CGAL. *Comput. Geom. Theory Appl.*, 22:5–19, 2002.

[7] Jean-Daniel Boissonnat and Monique Teillaud, editors. *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, 2006.

[8] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 165–174, New York, NY, USA, 1998. ACM.

[9] Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Stefan Schirra, and Sylvain Pion. *2D and 3D Kernel*, 3.2 edition, 2006. CGAL User and Reference Manual.

[10] Jean-Baptiste Debard, Romain Balp, and Raphaëlle Chaine. Dynamic Delaunay Tetrahedralisation of a Deforming Surface. *The Visual Computer*, page 12 pp, August 2007.

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

[12] Olivier Devillers and Franco P. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete Comput. Geom.*, 20:523–547, 1998.

[13] Luc Devroye, Christophe Lemaire, and Jean-Michel Moreau. Expected time analysis for Delaunay point location. *Comput. Geom. Theory Appl.*, 29:61–89, 2004.

[14] Qiang Du, Maria Emelianenko, and Lili Ju. Convergence of the lloyd algorithm for computing centroidal voronoi tessellations. *SIAM J. Numer. Anal.*, 44(1):102–119, 2006.

[15] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, 1999.

[16] E. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge, 2001.

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

[18] Efi Fogel and Monique Teillaud. Generic programming and the CGAL library. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

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

[20] Herman Geuvers, Milad Niqui, Bas Spitters, and Freek Wiedijk. Constructive analysis, types and exact real numbers. *Mathematical. Structures in Comp. Sci.*, 17(1):3–36, 2007.

[21] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[22] J. Hartigan and M. Wong. Algorithm as136: A k-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979.

[23] Susumu Hasegawa, Hiroshi Imai, Mary Inaba, Naoki Kato, and Jun Nakano. Efficient algorithms for variance-based k-clustering. In *First Pacific Conference on Computer Graphics and Applications*, 1993.

[24] C. Hazelwood. *A Divide and Conquer Approach to D-dimensional Triangulations*. PhD thesis, Department of Computer Science, University of Texas at Austin, 1988.

[25] P. S. Heckbert. Color Image Quantization for Frame Buffer Display. *ACM Computer Graphics (ACM SIGGRAPH '82 Proceedings)*, 16(3):297–307, 1982.

[26] Mary Inaba, Naoki Katoh, and Hiroshi Imai. Applications of weighted voronoi diagrams and randomization to variance-based k -clustering (extended abstract). In *Symposium on Computational Geometry*, pages 332–339, 1994.

[27] Xiang-Yang Li and Shang-Hua Teng. Generating well-shaped delaunay meshed in 3d. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–37, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[28] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[29] Guillaume Melquiond and Sylvain Pion. Formal certification of arithmetic filters for geometric predicates. In *Proc. 17th IMACS World Congress on Scientific , Applied Mathematics and Simulation*, 2005.

[30] A. Okabe, B. Boots, and K. Sugihara. Spatial tessellations: Concepts and applications of voronoi diagrams, wiley. *NEW*, York, 1992.

[31] Rafail Ostrovsky, Yuval Rabani, Leonard J. Schulman, and Chaitanya Swamy. The effectiveness of lloyd-type methods for the k-means problem. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 165–176, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Sylvain Pion and Monique Teillaud. *3D Triangulations*, 3.2 edition, 2006. CGAL User and Reference Manual.

[33] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, October 1990.

[34] D. Russel. *Kinetic Data Structures in Practice*. PhD thesis, Stanford University, 2007.

[35] M J Sabin and R M Gray. Global convergence and empirical consistency of the generalized lloyd algorithm. *IEEE Trans. Inf. Theor.*, 32(2):148–155, 1986.

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

[37] Richard Shewchuk. Star splaying: an algorithm for repairing delaunay triangulations and convex hulls. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 237–246, New York, NY, USA, 2005. ACM.

[38] Jane Tournois, Pierre Alliez, and Olivier Devillers. Interleaving delaunay refinement and optimization for 2d triangle mesh generation. In *Meshing Roundtable conference proceedings*, pages 83–101, 2007.

[39] Jane Tournois, Pierre Alliez, Camille Wormser, and Mathieu Desbrun. Quality isotropic tetrahedron meshing with optimal delaunay triangulations, 2008.

[40] S. Wan, S. Wong, and P. Prusinkiewicz. An algorithm for multidimensional data clustering. *ACM Transactions on Mathematical Software*, 14(2):153–162, 1988.

[41] Mariette Yvinec. *2D Triangulations*, 3.2 edition, 2006. CGAL User and Reference Manual.

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

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

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