Fast Delaunay Triangulation for Converging Point Relocation Sequences

Pedro Machado Manhães de Castro, Olivier Devillers

Fast Delaunay Triangulation for Converging Point Relocation Sequences

Pedro Machado Manhães de Castro[*]   Olivier Devillers[*]

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. Experiments show 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 several times faster than *rebuilding*.

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, and –2– it has many practical uses such as meshing for finite elements methods or surface reconstruction from point clouds.

For several applications the data are moving and thus the triangulation evolves with time. It arises for example when meshing deformable objects [4], or in some algorithms relocating the points by variational methods [1].

We first recall that the *Delaunay triangulation $DT(S)$* of a set S of n points in $\mathbb{R}^d$ is a *simplicial complex* such that no point in S is inside the circumsphere of any *simplex* in $DT(S)$. Several algorithms to compute the Delaunay triangulation are available in the literature. Many of them work in the *static* setting, where the points are fixed and known in advance. There are also a variety of so-called *dynamic* algorithms [5], 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 [8]. 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 $S(t)$ 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 $S(t_j)$ and $S(t_i)$, 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 [8]. 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 [2] where the points are converging to a final position.

2   Safe regions

Given any certificate $C : \mathcal{A}^m \to \{-1, 0, 1\}$ acting on a m-tuple of points $\zeta = (\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_m) \in \mathcal{A}^m$, 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. Hereafter, a certificate is *valid* when it is positive. Then, more precisely, the tolerance can be stated as follows:

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

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:

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

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.

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*.

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:

$$\tilde{\epsilon}(\mathbf{z}) = \inf_{\substack{\mathcal{B} \ni \mathbf{z} \\ \mathcal{B} \in \mathcal{T}}} dist_H(\mathbf{z}, \mathcal{D}(\mathcal{B})). \qquad (3)$$

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 that $\tilde{\epsilon}(\mathbf{z}) = \epsilon(\mathbf{z})$.

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 1 for an illustration. Details on the computations of such objects can be found in the extended version of this paper [3].
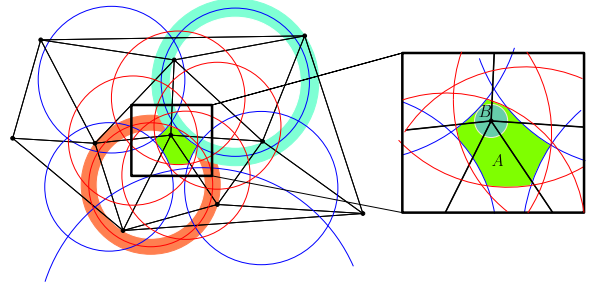

Figure 1: $\mathbf{z} \in \mathbb{R}^2$ the center of B. The region A is the safe region of $\mathbf{z}$, while B is its tolerance region.

3 Delaunay Maintenance Algorithms

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. Since the cost of deletions is very high, in practice, rebuilding the whole triangulation is faster. However, there is a number of algorithms which require to compute the next location of a vertex one by one, updating the Delaunay triangulation after each relocation. Placement algorithm is dynamic, unlike rebuilding, and 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.

We redesigned the placement algorithm so as to take into account the tolerance region of each relocated vertex. In practice, the algorithm proposed is 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 will be denoted by *tolerance algorithm*. It is dynamic, preserving all benefits from placement compared to rebuilding.

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 S, 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 $\mathbf{z}$.
$(\mathbf{m_z}, D_{\mathbf{z}}) \leftarrow (\mathbf{p}, dist(\mathbf{f_z}, \mathbf{p}))$;
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 generator 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

The algorithm is shown to terminate as each processed 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 $O(n)$. Discussion on robustness issues can be found in the extended version of this paper [3].

4 Experimental Results

A centroidal Voronoi tessellation is a Voronoi tessellation whose generating points are the centroids (centers of mass) of the corresponding Voronoi regions [6]. 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 or deterministic. One deterministic method is the well-known *Lloyd's iterations* [7]. 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.

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

In two dimensions, we consider four different density functions: $\rho_1 = 1$, $\rho_2 = x^2 + y^2$, $\rho_3 = x^2$ and $\rho_4 = \sin^2 \sqrt{x^2 + y^2}$. And we run the Lloyd's iterations to obtain centroidal Voronoi tessellations according to them. 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. We run the Lloyd's iterations during 1000 iterations, which is required to generate satisfactory results (see Figure 2).
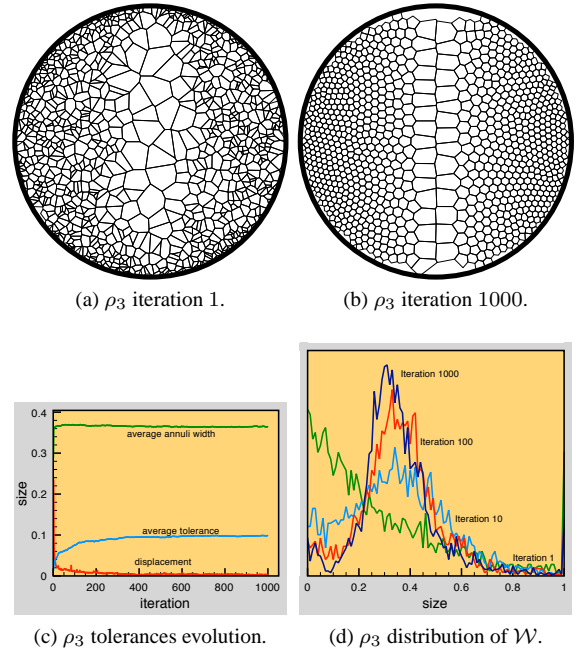

(a) ρ_3 iteration 1.   (b) ρ_3 iteration 1000.

(c) ρ_3 tolerances evolution.   (d) ρ_3 distribution of $\mathcal{W}$.

Figure 2: 1.000 points in a circle with densities ρ_1 and ρ_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) [9]. 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 3 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).
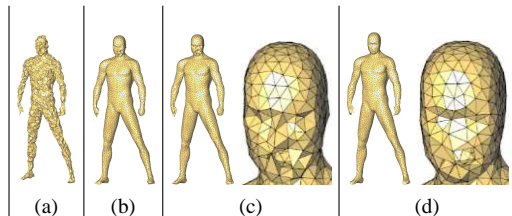

| (a) | (b) | (c) | (d) |

Figure 3: In (a) man initially; In (b), (c) and (d) man after 10, 100 and 1000 iterations respectively. (c) and (d), show close-ups on the head.

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.

For a random point distribution and random displacements of length δ, 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, 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.

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. 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. In three dimensions, if all displacements magnitudes are lower or equal to $avg(\epsilon(V))$, T is more than three time faster than placement.

We now discuss the performance of the algorithms for Lloyd iterations, considering a thousand of iterations. We implemented in addition a small variation of the tolerance algorithm, suggested in Section 3, which consists of rebuilding for the first few iterations, and, swapping to T. We denote this algorithm by $R+T$.

In two dimensions, T and $R+T$ outperform both rebuilding and placement. In three dimensions, they outperform placement in all configurations. This enhancement is relevant for applications requiring to move vertices one at a time and for dynamic triangulations, which cannot be achieved by rebuilding. However, to outperform rebuilding is harder in three dimensions, because the removal op-

eration 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, T becomes faster. As shown in Figure 3, 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. More details on experimental results are available in the full version of the paper [3].

5 Conclusion

This paper deals with the problem of updating Delaunay triangulations for moving points. We introduce the notion of the tolerance and safe region of a vertex in this context. We end up with 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.

References

[1] Alliez P., Cohen-Steiner D., Yvinec M., and Desbrun M. Variational tetrahedral meshing. *ACM Trans. on Graphics*, 24:617–625, 2005. SIGGRAPH '2005 Conf. Proc.

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

[3] De Castro P. M. M. and Devillers O. Delaunay triangulations for moving points. Research report, INRIA, 2008, http://hal.inria.fr/inria-00344053/.

[4] Debard J. B., Balp R., and Chaine R. Dynamic Delaunay Tetrahedralisation of a Deforming Surface. *The Visual Computer*, page 12 pp, Aug. 2007.

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

[6] Du Q., Faber V., and Gunzburger M.. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, 1999.

[7] Ostrovsky R., Rabani Y., Leonard J. Schulman, and Swamy C. The effectiveness of Lloyd-type methods for the k-means problem. In *FOCS '06: Proc. of the 47th Annu. IEEE Sympos. on Found. of Comp. Sci.*, pages 165–176, Washington, DC, USA, 2006. IEEE Computer Society.

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

[9] Tournois J., Alliez P., Wormser C., and Desbrun M. Quality isotropic tetrahedron meshing with optimal Delaunay triangulations, 2008.