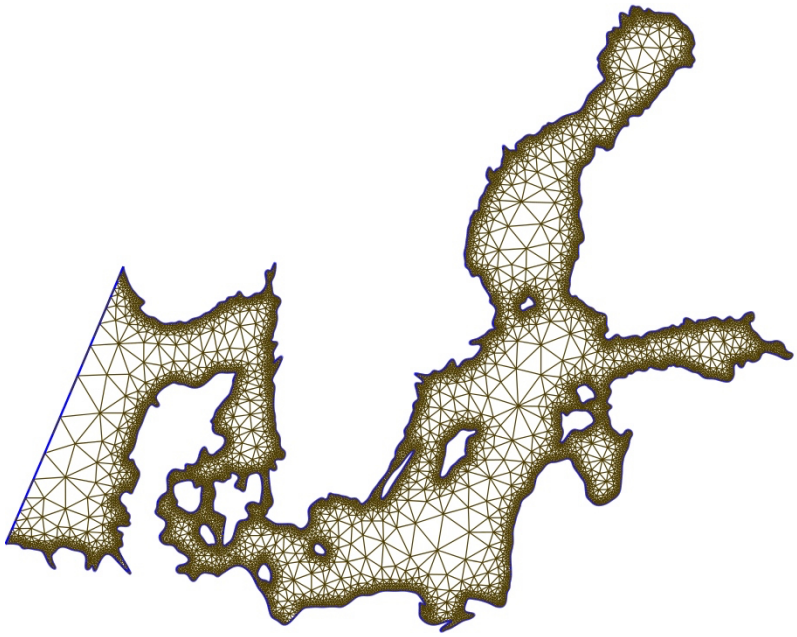
Université Catholique de Louvain

MECA2170

Numerical Geometry

Delaunay's triangulation

Report



CERCKEL Arnaud
STEVENS Nicolas

Professeurs : Vincent Legat & Jean François Remacle

16 décembre 2014

Abstract

This report studies the Delaunay's triangulation, more explicitly the Watson's algorithm depicted in [2], and comes to explain and illustrate our implantation written in the code :

C code	name of the file	description
Launch code	`main.c`	launch the code
Main code	`watson.c`, `watson.h`	implementation of the Watson's algorithm
Robust library	`robustFunctions.c`	see section "robustness"
GUI	`glfem.c`, `glfem.h`, `graphic.c`	implementation of the graphic interface
Matlab code		
Write points code	`generatePoints.m`	function to write random points file

We first present the the whole algorithm and the data structures related to it. Then we present the architecture of our codes. We go ahead with a deep insight in the robustness question. Finally we present the performance results of our code.

1 Delaunay's triangulation : Watson's algorithm

Watson's algorithm [2] is a randomized incremental algorithm : we first build a "big triangle" containing all the points, then we add point by point maintaining the Delaunay's properties at each step. The input is an array of points containing the x and y coordinate of each point. The output is a file containing the n triangles (this is n lines and 3 column containing indices of the nodes).

Data's structures

We have seven data's structures as depicted at figure 1 and implemented in `watson.h`.

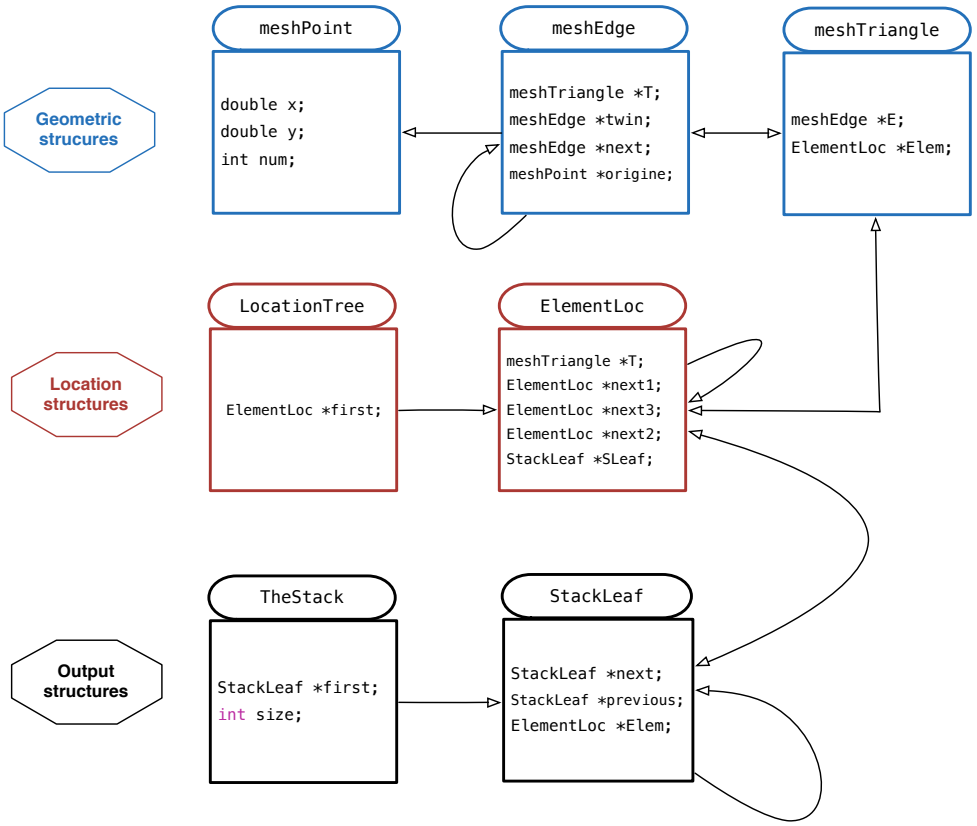


FIGURE 1 – Data structures of our code.

We divided them into three categories :

— the geometric structures contain all the informations about the mesh itself : the nodes, the triangles and the points. Note that each triangle has its own edges ordered counter-clockwise ; therefore, a triangle only need to know one of its edge, from which it can reach the other edges, points... The structure of the edges is the one which makes the links between points and triangle and also between neighbour triangles (as `meshEdge` contain a pointer toward its `twin` which contain a pointer toward its `meshTriangle`).

— the location structure, more precisely the location tree, denoted later as $\mathcal{T}$. This structure works as follow : its first element is the "big triangle" ; then when we add a point p_r, we create new triangles which are the `next` elements of the "parent" element in the tree. So each `ElementLoc` has three pointer toward the next `ElementLoc`, a pointer toward the triangle it represent and a pointer toward its potential element in the stack.

— the stack structures, denoted later as $\mathcal{S}$, which contains at each step, the leaves of the location tree $\mathcal{T}$. This structure is useful because it allows us to easily return the leaves of the the tree (so the final mesh, the output of the algorithm) at the end of the algorithm. It is also useful to plot the evolution of the mesh (cf. the GUI).

Watson's algorithm

Our algorithm can be written as follow :

Algorithm 1 Watson-DelaunayTriangulation($\mathcal{P}$)

Input A set of n points $\mathcal{P}$
Output An array of Delaunay's triangles
Initialise the "big triangle" by choosing three extreme points
Initialise the data's structures (the location tree $\mathcal{T}$ and the stack $\mathcal{S}$)
perform a random permutation on $\mathcal{P}$
for $r \leftarrow 1$ **to** n **do**
    Find triangle $p_ip_jp_k \in \mathcal{T}$ containing p_r
    if p_r lies in the interior of triangle $p_ip_jp_k$ **then**
        Add edges from p_r to p_i, p_j and p_k and create three new triangles. Add them to the structure $\mathcal{T}$ and actualise $\mathcal{S}$.
        Legalize edge (p_r, $\overline{p_ip_j}$,triangle $p_ip_jp_r$,$\mathcal{S}$)
        Legalize edge (p_r, $\overline{p_jp_k}$,triangle $p_rp_jp_k$,$\mathcal{S}$)
        Legalize edge (p_r, $\overline{p_kp_i}$,triangle $p_ip_rp_k$,$\mathcal{S}$)
    else(p_r lies on an edge of triangle $p_ip_jp_k$, say p_ip_j)
        Add edges from p_r to p_k and to p_l (the opposite node in the adjacent triangle $p_ip_jp_l$) and create four new triangles. Add them to $\mathcal{T}$ and actualise $\mathcal{S}$.
        Legalize edge (p_r, $\overline{p_ip_l}$,triangle $p_ip_rp_l$,$\mathcal{S}$)
        Legalize edge (p_r, $\overline{p_lp_j}$,triangle $p_rp_jp_l$,$\mathcal{S}$)
        Legalize edge (p_r, $\overline{p_jp_k}$,triangle $p_rp_jp_k$,$\mathcal{S}$)
        Legalize edge (p_r, $\overline{p_kp_i}$,triangle $p_ip_rp_k$,$\mathcal{S}$)
return $\mathcal{S}$

where the step "legalize edge" corresponds to :

Algorithm 2 Legalize Edge(p_r, $\overline{p_ip_j}$, triangle $p_ip_jp_r$,$\mathcal{S}$)

Let $p_ip_jp_k$ be the adjacent triangle to $p_ip_jp_r$. Test if p_k lies inside the oriented circle define by $p_ip_jp_r$. If it is the case, $\overline{p_ip_j}$ is illegal
if $\overline{p_ip_j}$ is illegal **then**
    create a new edge $\overline{p_rp_k}$ and create two new triangles associated. Add them to the structure $\mathcal{T}$ and actualise $\mathcal{S}$.
    Legalize edge (p_r, $\overline{p_ip_k}$,triangle $p_rp_ip_k$,$\mathcal{S}$)
    Legalize edge (p_r, $\overline{p_kp_j}$,triangle $p_rp_jp_k$,$\mathcal{S}$)

More explicitly for the initialisation of the "big triangle" :

```
thePoint[0] = meshPointCreate(minX - 10*(maxX-minX),minY-10*(maxY-minY),0);
thePoint[1] = meshPointCreate(maxX + 10*(maxX-minX),minY-10*(maxY-minY),1);
thePoint[2] = meshPointCreate((maxX+minX)/2,maxY+10*(maxY-minY),2);
```

Our implementation is in the files `watson.c`. This algorithm is expected to run in $O(n \log n)$ time in most cases. $O(n^2)$ in "worst" cases (see [2]). We will go back on this later.

2 Architecture of the code

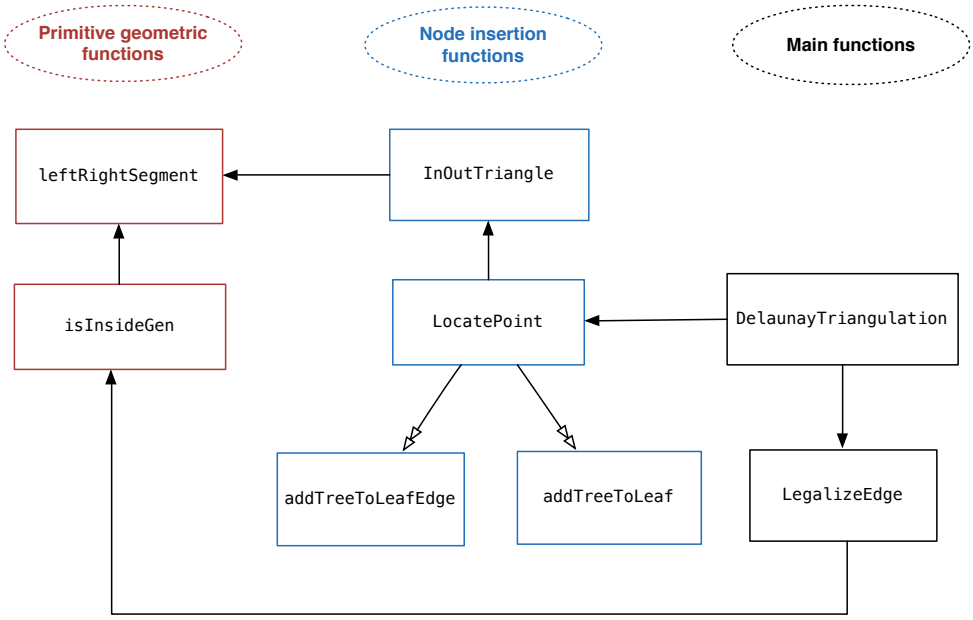


FIGURE 2 – Main architecture of the code. The simple arrows mean "use this function" and the double arrows mean "have as consequences".

Figure 2 depicts the global architecture of our code. We have basically 4 types of functions :

— the primitive **geometric functions** :

1. `leftRightSegment` test whether a point lies on the right or on the left of a segment ;
2. `isInsideGen` test whether a point lies inside or outside of a circle (it calls `leftRightSegment` because we need to have an oriented circle).

It is important to notice that even if these functions seem trivial, they are the core of our code which determine the correctness of the code as we base all our processes on the answer of these functions. These are the only functions where we actually make computation (see section "Robustness" for more informations).

— the **insertion functions** which performs

1. the localisation step (the critical operation is `leftRightSegment` which is required to know in which triangle the adding point is) ;
2. the adding step : it adds it to the tree structure. This ere mainly manipulate pointer and create new structure's elements.

— the **main functions** :

1. `DelaunayTriangulation` which is the main function performing the Delaunay's triangulation ;
2. `LegalizeEdge` which performs the pivot operation (the critical operation is `isInsideGen` which determine whether the pivot is performed or not).

— the **maintenance functions** (useful but not interesting function, which is why they are not figured on figure 2) :

1. the functions to create the structures (`meshPointCreate`, `meshEdgeCreate`...).
2. the stack functions useful to manipulate the stack (`DeleteStackElement`...)

— the graphical interface functions (in `glfem.c`).

3 Robustness

As said previously, the "critical" operations are the primitive geometric operations as they determine the correctness of the code. Even if the questions of "in/out circle" and "left/right segment" might seem obvious, the problems of floating numbers which can occur are sensitive issues. In this section, we present why these problems arise and how they are handled.

One may define ([4]) the robustness as the ability of an algorithm to produce the correct result for some perturbation of the input. As we will see, rounding errors in floating point arithmetic are such perturbations.

3.1 Floating number

We first have a look at how does the computer store and handle problems. As a computer does not have infinite precision, the real numbers are stored in the computer as *floating numbers* ([3],[4]). A floating number has a base B, a fixed mantissa length l and an exponent e :

$$\pm \underbrace{d_0.d_1d_2\cdots d_{l-1}}_{l} * B^e.$$

It is *normalized* if $d_0 = 0$. As numbers are represented as finite floating point, rounding error may occur. This lead to numerical error. Let a be a number and a_f its floating number representation. We define the *absolute error* as $|a - a_f|$ and the *relative error* as $|a - a_f|/|a|$. The relative error is bounded by the *machine epsilon*, this is $\frac{B}{2}B^{-l}$. The way the machine makes the rounding operation is defined by the IEEE standard 754.

These floating numbers can be stored in *simple precision* (32 bit word, this is $e \in [-126, 127]$; `float` in C) or *double precision* (64 bit word, this is $e \in [-1022, 1023]$ and $l = 53$; `double` in C).

Let's give an example of the dramatic effect these rounding might involve. Let three number $a = 5.6543723$, $b = 5.654371$ and $c = 0.000001$. And their floating number representation if mantissa length is $l = 6 : a_f = 0.565437 \cdot 10^1$, $a_f = 0.565437 \cdot 10^1$ and $c = 0.1 \cdot 10^{-5}$. we have :

$$a - b - c = 3 \cdot 10^{-07}$$
$$a_f - b_f - c_f = -0.1 \cdot 10^{-5}$$

Where these two simple expression does not even have the same sign !

3.2 Primitive geometric operations and determinant

In numerical geometry, many primitive operation (as the including in a circle or the orientation problem) consist of evaluating the sign of a determinant ([1], [6]).

Let $\overrightarrow{ab}$ be an oriented segment and c a point. Determine whether c lies on the left or on the right of $\overrightarrow{ab}$ can be solve by evaluating the sign of the following determinant :

$$\det \begin{pmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{pmatrix} \tag{1}$$
$$\Leftrightarrow \det \begin{pmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{pmatrix} \tag{2}$$

if this determinant is positive, c lies on the right of $\overrightarrow{ab}$, if negative, on the left and if null, on the segment.

And let abc be an oriented circle (meaning that a, b and c are points on the circle, occurring in counter-clockwise order), and d a point. Determine whether d lies inside or outside the circle abc can be solve by

evaluating the sign of the following determinant :

$$\det \begin{pmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{pmatrix} \tag{3}$$

$$\Leftrightarrow \det \begin{pmatrix} a_x-d_x & a_y-d_y & (a_x-d_x)^2+(a_y-d_y)^2 \\ b_x-d_x & b_y-d_y & (b_x-d_x)^2+(b_y-d_y)^2 \\ c_x-d_x & c_y-d_y & (c_x-d_x)^2+(c_y-d_y)^2 \end{pmatrix} \tag{4}$$

If the determinant is positive, d lies inside the circle, if negative, outside and if null, on the circle.

So we see that these primitive operation correspond to evaluate the sign of a determinant. As we have see before, in floating point representation, rounding error may occur. And as we have seen in an example, this might even change the sign of an expression. So evaluate these determinant is a critical issue.

A first idea to reduce the numerical error is to compute it in a numerical "robust" way. The determinant above are already a "not so bad" way to make these geometric tests (we might have build more "hard-coded" manner less robust). Moreover, it turns out ([5]) that formulation (2) and(4) are better in terms of error propagation that (3) and (1). So this are the first "non-robust" (but acceptable) methods that we implemented in our functions `leftRightSegment` and `isInsideGen` (if they are called with `ROBUST=0`).

But these method remains non-robust and are not design to resist to extreme cases. This are cases where the evaluation of the sign of the determinant might fail. The next sections introduce methods to deal with these issues. Notice that the fact that these methods are non-robust does not mean that they are bad. Indeed the limit cases we are handling now occurs a very few times so the non-robust method work well most of the time.

3.3 Magic epsilon

A very common idea ([4]) would be to use a bound saying "*if something is closed to zero, it is zero*". This is, setting a ϵ_{magic} and say if $a - b < \epsilon_{magic} \Rightarrow a = b$. This might solve some problems and work in a lot of cases. But this approach might fail in some extreme cases and moreover, under this ϵ_{magic}, the equality is not transitive. So we drop this idea.

3.4 Clarkson's method

We do not spend much time on this method as we did not implement it ; we just briefly quote it for our most interested and zealous readers.
The idea of Clarkson's method, developed in [1], is to evaluate the sign of the determinant of A, using a wise preconditioning matrix which reduce the numerical errors. Clarckson's preconditioning applies a variant of Gram-Scmidt reduction which is why it is sometimes called reorthogonalization method.

3.5 Exact arithmetic

A promising idea ([1], [4], [5]) would be to work with exact numbers, this is *exact arithmetic* define ([8]) as :
In computer science, arbitrary-precision arithmetic [...] indicates that calculations are performed on numbers whose digits of precision are limited only by the available memory of the host system. This contrasts with the faster fixed-precision arithmetic [...] which typically offers between 8 and 64 bits of precision.

For instance, exact arithmetic with *rational* numbers can be done by storing two integer numbers (the numerator and denominator). Exact arithmetic for *real* numbers is a little more intricate. On way to figure out how exact arithmetic works is to think of an arithmetic expression as an *expression tree* ([4]) : the leaves are numbers and the internal nodes are operations $+, *, -, /$.
Arbitrary precision values are expressed as $x = x_n + \cdots + x_2 + x_1$ (where $x_i > x_{i-1}$). We call $\oplus$, $\ominus$ and

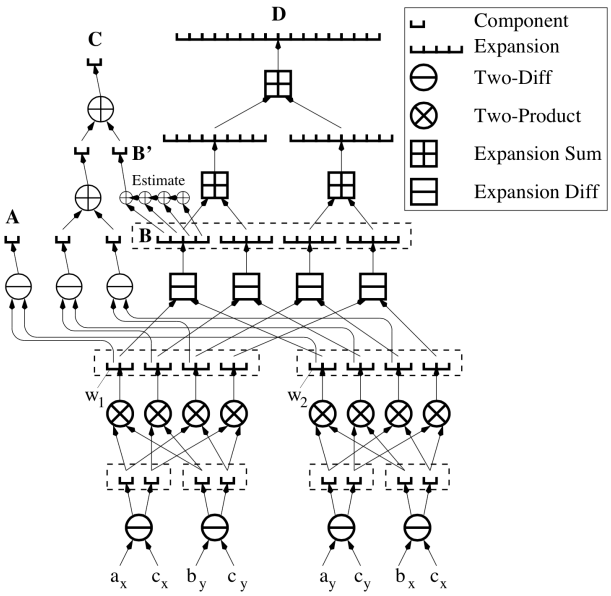


FIGURE 3 – Adaptive calculations used by the 2D orientation test. Sources : [5]

$\otimes$ the p-bit floating point addition, subtraction and multiplication with exact rounding. For instance, the orientation problem described above can be expressed as 3.

Obviously exact arithmetic uses more memory than classical floating point arithmetic. It is also slower (significantly slower).

Libraries in C++ implement such exact arithmetic ; for instance LEDA or LD (Fortune and Van Wyk).

3.6 Arithmetic filters

The main inconvenient of exact arithmetic is that it is significantly slower than classical floating point arithmetic. Furthermore, we only need to use exact arithmetic for some "limit" cases. An obvious idea would be to use exact arithmetic only in some critical situations. The implementation of this method is called *arithmetic filters* ([4], [5]). The idea of floating point filters is :

— compute the expression using classical floating numbers arithmetic ;
— compute a bound of the error of the floating point computation and compare it with the absolute value of the expression compute before.
— if the error is smaller than the "exact value" of the expression and its approximation in floating point have the same sign, so our approximation is acceptable ;
— if the error is bigger we have no guarantee that the sign of the floating point approximation and the exact value is the same. In this case, we use exact arithmetic computation to re-evaluate the expression.

Fortune and Van Wyk implemented such arithmetic filter.

3.7 Implementation

In our code, we use an implementation developed by J. R. Shewchuk ([5] for the paper, [6] for the code) available in open source (downloaded in the file `robustFunctions.c`). He implemented methods using exact computation and arithmetic filters (slightly improve using computations of increasingly accuracy, cf. results A, B, C or D of figure 3, and many other heuristics...). In our functions `isInsideGen` and

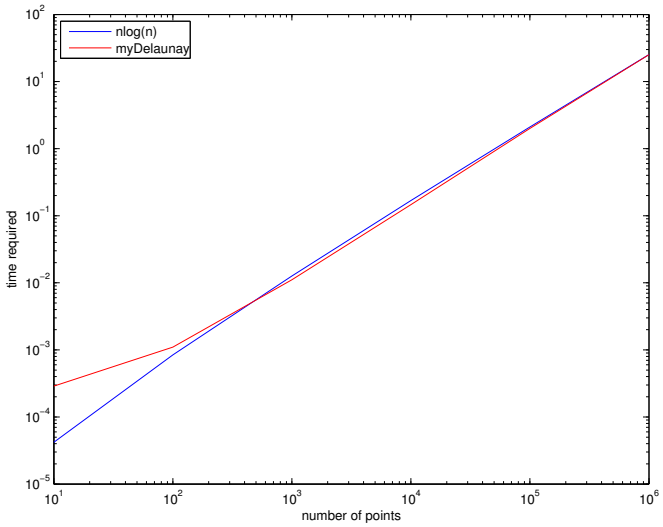


FIGURE 4 – Computation time (red line) for randomly chosen points from $1e1$ to $1e6$ and theoretical time (blue line) $n \log n$.

	10^1	10^2	10^3	10^4	10^5
memory (bytes)	10,208	151,504	1,624,304	16,513,168	165,296,208

TABLE 1 – Heap memory in use at exit.

`leftRightSegment` we either make our non-robust implementation (if the input variable `ROBUST=0`) or we call the robust Shewchuk's functions `incircle` and `orient2d` (if the input variable `ROBUST=1`).

4 Performances results

Let's now have a look at the result of our implementation. Generating a set of points with Matlab from 10^1 to 10^6, we compute the Delaunay Triangulation on those point and measure the time required. The implementation is the first script written in `time.m` and the result is shown in figure 4. The blue line represents the theoretical result $Cn\log(n)$ and the red line is the observed times. Adjusting C based on the mean of the times for 10^2 to 10^6, the computation time for 10^1 is too short to be relevant, we obtain a good fit. It illustrates that the Watson algorithm is usually in $O(n\log(n))$. The result of our Delaunay triangulation on a random set of 10 points is shown in figure 6a.

As explained earlier we have two types of implementations, a robust one and a less robust one. The table 2 shows the exact times for both implementation and the ratio between them. The authors of the robust one speak about an average of 30% slowdown when using their implementation. We see that in our case for a sufficient data set we are about 20% slower, quite close from the time expected.

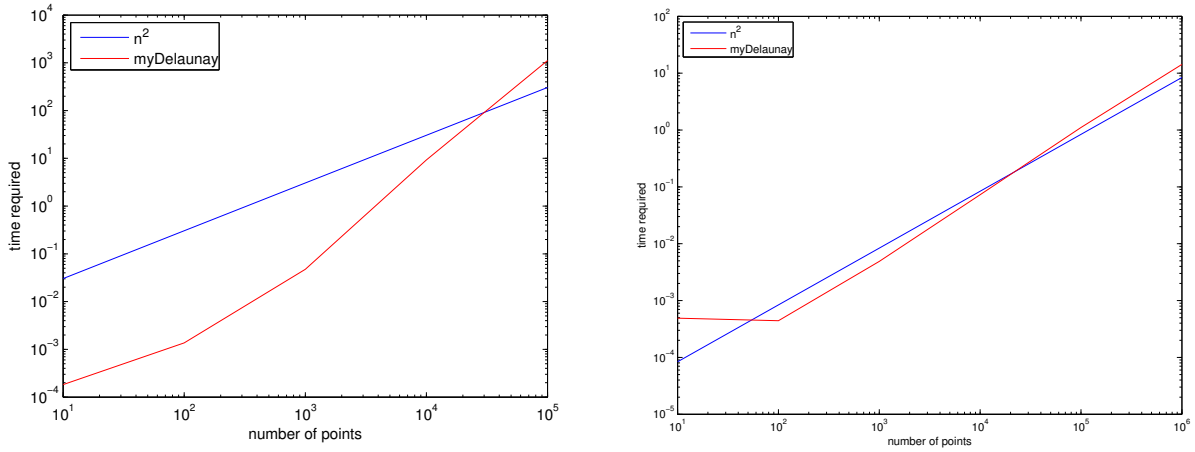
Moreover, comparing the time for $1e6$ points with the implementation suggested in [7], we have both around 20 seconds. We don't have the same computer but it shows that we have the same order of magnitude.

The space complexity is theoretically in $\mathcal{O}(n)$. Using valgrind we measure the heap memory use at exit. It could seem as we don't think about freeing the memory but the structures we require are used during the whole process and so we can only free at the end. In consequence it's not really strategical and it allows us to easily measure the final space complexity, which is the maximal one. The memory at exit is shown in table 1. We see that we have almost the same result as the the theoretical one. Each time we multiply by 10 we increase the memory used by a factor 10.

We also implement a script to generate points based on a worst case scenario, `generatePointLimite.m`.

	10^1	10^2	10^3	10^4	10^5	10^6
not robust	0.0102	0.0007	0.0062	0.1086	1.5392	20.2006
robust	0.01030	0.0011	0.0096	0.1333	1.9299	25.0801
ratio	0.9903	0.6364	0.6458	0.8147	0.7976	0.8054

TABLE 2 – Comparison of time between the use of the not robust and robust implementations.



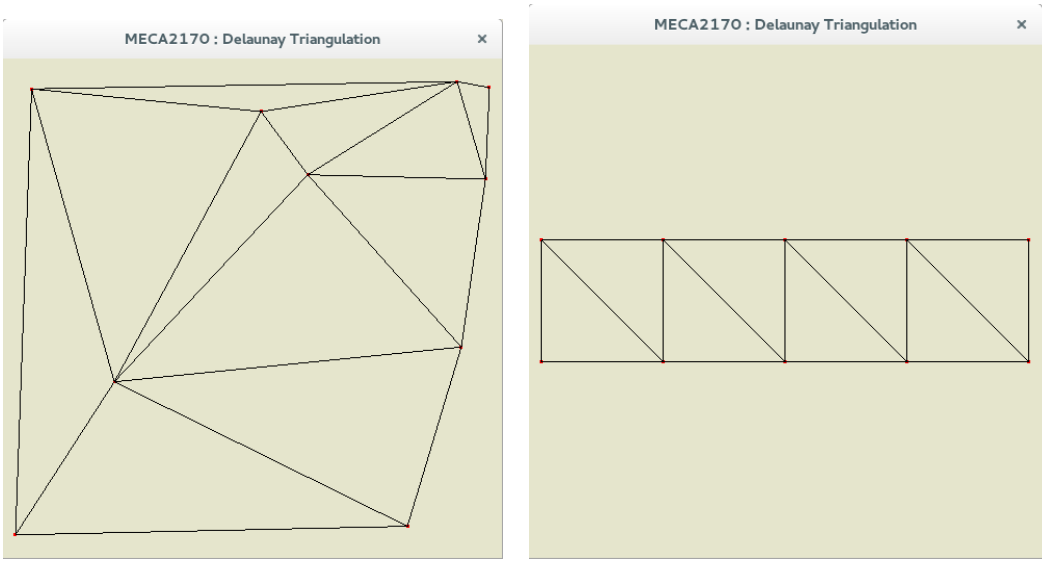
(a) Ordered (only until 10^5)

(b) Randomly mixed

FIGURE 5 – Computation time (red line) for points from $1e1$ to $1e5$ choosen in a worst case scenario and theoretical time (blue line) n^2.

This scenario is a set of points forming a sequence of squares. It's considered as a worst case scenario because the points are are always exactly on the circle made of the 3 other points of the square. So the algorithm has a risk of falling into a infinite loop checking a point changing the triangle and then doing it again. In theory the worst case scenario can be in $O(n^2)$ with the Watson algorithm. We see the importance of randomisation in this case. Indeed as shown in figure 5a, if we do not randomise the data we are far worse than the theoretical result and if we randomise it, figure 5b, we are then close to the theoretical result. An example of Delaunay triangulation on a sequence of squares is shown in figure 6b.

A way of improvements would be in the randomised worst case scenario. Indeed in this case we have sometimes some segmentation fault due to reasons we didn't get the time to investigate. Note that if we receive a random set of points or an ordered worst case scenario, our algorithm seems like working all the time.



(a) 10 random points.

(b) 10 points based on a worst case scenario.

FIGURE 6 – OpenGL representations of some Delaunay Triangulations.

Références

[1] Hervé Brönnimann and Mariette Yvinec. Efficient exact evaluation of signs of determinants. *Algorithmica*, 27(1) :21–56, 2000.

[2] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational geometry*. Springer, 2000.

[3] Vincent Legat. *Methodes numériques*. SICI, 2012.

[4] Stefan Schirra. *Robustness and precision issues in geometric computation*. MPI Informatik, Bibliothek & Dokumentation, 1998.

[5] Johnathan Richard Shewchuk. *Robust adaptive floating-point geometric predicates*. 1996.

[6] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust predicates for computational geometry.

[7] Jonathan Richard Shewchuk. Triangle.

[8] Jonathan Richard Shewchuk. Wikipedia, the free encyclopedia, arbitrary-precision arithmetic.