Homework 1 : Delaunay Triangulations

General instructions :

1. Form groups of three students. If not possible, groups of two are allowed. Groups of four are not permitted (please do not ask for exceptions).
2. You are allowed to use AI tools at any time. However, the evaluation of your project will be based not only on the performance of your implementation, but also on an interview of each group with all instructors. It is therefore absolutely essential that you both verify what the AI provides and understand it thoroughly, down to the smallest details.
3. The code will be graded on 20 points according to the structure detailed in this document. This grade is initially attributed to the group. The interview will allow us to objectify this grade and to adjust it individually, depending on each member's performance during the discussion. Adjustments may be made both downwards (in case of insufficient understanding) and upwards (in case of outstanding individual insight).
4. You must submit a short report of at most one page explaining how you organized your work. Be precise about "who did what". Each member of the group must sign this report.
5. **Important notice.** We will use all anti-plagiarism tools at our disposal. Any confirmed case of plagiarism will result in a grade of zero for both the group who copied and the group whose work was copied.

In this first assignment, we'll ask you to calculate the Delaunay triangulation of set $P = (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$ of n points $\mathbf{x}_i = (x_i, y_i)$ of the plane.

The deadline for your assignment is Halloween 2025.

1 Input/Output

The input for your code will be a file structured as follows. The first line of the file will be an integer n corresponding to the number of points to be triangulated, and the n following lines will list the x_i and y_i coordinates of the n points to be triangulated. The python script `genpts.py` (see Appendix A) can be used to generate a file of this type with an arbitrary number of points.

Your program `del2d.py` must be able to run non-interactively : we must be able to test it as follows :

```
python del2d.py -i pts.dat -o triangles.dat
```

The output file `triangles.dat` will be structured in a similar way to the input file. The first line of `triangles.dat` will contain an integer n_t designating the number of triangles in the Delaunay triangulation $DT(P)$ and the n_t following lines will each contain three integers t_{j1}, t_{j2}, t_{j3}, $j = 0, \ldots, n_t - 1$ indexed from 0 to $n_t - 1$ and designating the three vertices of the j triangle.

2 The Algorithm

Here we describe the various steps in the algorithm used to calculate this triangulation.

2.1 Data-structure

You can use the python class `TriangularMesh` which is available in Appendix A. This `HalfEdge` data structure is ideal for programming Delaunay triangulation.

2.2 Initialization

You'll use an iterative algorithm to calculate $DT(P)$. You need to initialize the algorithm with a simple triangulation consisting of two triangles that completely enclose the point cloud. You must therefore start by calculating the axis oriented bounding box of P i.e. calculate two points $(x_{\min}, y_{\min})$ and $(x_{\max}, y_{\max})$ that are such that $x_{\min} < x_i$, $\forall i$, $y_{\min} < y_i$, $\forall i$ etc. Four *infinite* points $\mathbf{x}_n = (x_{\min} - L, y_{\min} - L)$, $\mathbf{x}_{n+1} = (x_{\max} + L, y_{\min} - L)$, $\mathbf{x}_{n+2} = (x_{\max} + L, y_{\max} + L)$ and $\mathbf{x}_{n+3} = (x_{\min} - L, y_{\max} + L)$, $L \in \mathbb{R}_{>0}$ are used to define an initial triangulation made of two triangles. $(\mathbf{x}_n, \mathbf{x}_{n+1}, \mathbf{x}_{n+2})$ and $(\mathbf{x}_n, \mathbf{x}_{n+2}, \mathbf{x}_{n+3})$. small understanding question : why is using L useful ?

2.3 Incremental construction of $DT(P)$ using Bowyer-Watson

There are several ways of obtaining $DT(P)$. You can use Bowyer Watson's algorithm as described in the course. Let DT_i be the Delaunay triangulation comprising the first i points of the list P. We now want to add $\mathbf{x}_{i+1}$ to form DT_{i+1}. To do this, you need to calculate the Delaunay cavity C_{i+1}, which contains all the triangles of DT_i that violate the Delaunay criterion, i.e. all the triangles whose circumscribed circle contains $\mathbf{x}_{i+1}$.

This cavity is simply connected and star-shaped, and the new point $\mathbf{x}_{i+1}$ is part of the kernel of C_{i+1}. It is therefore very simple to triangulate C_{i+1} by connecting $\mathbf{x}_{i+1}$ to all the edges forming the border of C_{i+1}. This operation adds exactly two triangles to DT_i.

3 Interfacing C and Python

In the context of this first project, there are two functionalities that are particularly useful to obtain a robust and efficient code, but that are not necessarily available as ready-to-use Python packages. We will therefore ask you to interface (bind) some C code with your Python implementation. Note that for the more adventurous among you, it is also possible to implement the entire triangulation procedure directly in C and only bind the main function, in order to remain consistent with the requirements outlined in the previous section.

One convenient way to interface C code with Python is to compile a C source file into a shared library and then access its functions from Python using the `ctypes` module (it belongs to Python's standard library : link to documentation). This allows Python to call compiled C routines directly, combining the performance of C with the flexibility of Python. Below we provide a minimal working example : a simple C function is compiled into a shared library, and then loaded and called from Python.

C code (`mycode.c`)

```c
#include <stdio.h>

// A simple function that doubles an integer
int double_value(int x) {
    return 2 * x;
}
```

Compilation

On Linux or macOS :

```
gcc -shared -o libmycode.so -fPIC mycode.c
```

On Windows (MinGW) :

```
gcc -shared -o mycode.dll -fPIC mycode.c
```

Python code (`test.py`)

```python
import ctypes
import os

# Load the shared library
lib = ctypes.CDLL(os.path.abspath("libmycode.so"))  # "mycode.dll" on Windows

# Tell ctypes the function signature
lib.double_value.argtypes = [ctypes.c_int]
lib.double_value.restype = ctypes.c_int

# Call the C function
x = 7
result = lib.double_value(x)
print(f"C function returned: {result}")
```

4 To infinity and beyond

If your program delivers a correct result – even if it's slow, we'll test your implementation up to 1000 random points – your assignment will be successful and will allow you to have a *maximal grade* of 10/20. To improve your score, we suggest several avenues of improvement.

4.1 Removing infinite points (max +2 points)

The triangulation your program produces at this stage has $n+4$ points. The Delaunay triangulation $DT(P)$ is supposed to contain only the n points of P and not the four extra points we used to initialize the algorithm. We want $DT(P)$ to cover exactly the convex hull of P and eliminating all the triangles in your triangulation that contain one of the 4 *infinite* points $\mathbf{x}_{n+j}$, $j = 0, 1, 2, 3$ does not guarantee that $DT(P)$ will be obtained. You are therefore asked to supply exactly $DT(P)$.

4.2 Robustness (max +2 points)

If you do not use robust predicates, a number of point configurations may cause your code to fail. For a baseline grade of 10/20 we will only test on random point sets, but if, for example, you attempt to triangulate a cloud of points that are all co-circular, your implementation will break. To make your code robust, you must rely on so-called *robust geometric predicates*. These are not easy to implement, but C versions are available and can be bound to Python. The most widely used ones are those developed by Jonathan Shewchuk [2] (see the source code in C).

4.3 An implementation in $\mathcal{O}(n \log n)$ (max +4 points)

In the course, we saw that sorting the set P using space-filling curves allows to reduce the complexity of the algorithm. Space filling curves are 1-dimensional traversals of n-dimensional space where each point in a discrete space is visited once. These family of curves have been heavily researched in the past for their locality preserving properties and are especially useful for dimensionality reduction. We've seen how to sort points along a Hilbert curve. The fastest known implementation of Hilbert sorting in two and three dimensions was developed by our team [1](see this repository) and is written in C. Other implementations do exist, and you may either program one yourself in C, as demonstrated during the course, or even implement it directly in Python.

Once you've sorted your points, you can "walk" into DT_i from one of the triangles in the cavity C_i and quickly find the triangle in DT_i that contains $\mathbf{x}_{i+1}$.

4.4 Go Interactive (max +4 points)

Delaunay triangulations are beautiful graphic objects and it's natural to want to draw them. In this last part of the project you are asked to implement an interactive Python program that plots the Voronoi diagram of a set of points in two dimensions. The program should allow the user to add points on the fly, for example by clicking on the canvas, and the diagram must be updated immediately. Ideally, the program should also provide the option to activate Lloyd's algorithm so that the points are iteratively repositioned at the centroids of their Voronoi cells until convergence. In practice, you may limit the number of Lloyd iterations to a maximum value specified by the user, with a default value of typically ten iterations.

To achieve this, you are encouraged to make use of suitable Python libraries such as `matplotlib` with event handling, or web-oriented frameworks like `bokeh` and `plotly`. Once again, the bravest among you can try a more graphic-oriented library like OpenGL. Be warned that OpenGL is *really* harder than the usual plotting libraries, but your favourite chatbot might help you well. After all OpenGL is 33 years old and has extensive documentation and examples available on the web.

If you decide to "Go Interactive", your final submission should also consist of a short `README` explaining how to run the program. During the interview, you will present a brief demonstration illustrating both point insertion and Lloyd iterations.

Références

[1] Célestin Marot and Jean-François Remacle. One machine, one minute, three billion tetrahedra. *International Journal for Numerical Methods in Engineering*, 121(23) :5219–5235, 2020.

[2] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust predicates for computational geometry. *Discrete & Computational Geometry*, 18(3) :305–363, 1997.

A Source code for genpts.py

```python
import sys
import numpy as np

def get_points(nr_points, minx=0, maxx=1, miny=0, maxy=1):

    # random values between 0 - 1
    points = np.random.rand(2, nr_points)

    # map x and y values between minx - maxx, miny - maxy
    points[0, :] = np.interp(points[0, :], [0, 1], [minx, maxx])
    points[1, :] = np.interp(points[1, :], [0, 1], [miny, maxy])

    return points


if __name__ == '__main__':

    p =  get_points (int(sys.argv[1]))

    f = open(sys.argv[2],"w")

    f.write(sys.argv[1]+"\n")

    for i in range(int(sys.argv[1])) :
        f.write(str(p[0,i])+" "+str(p[1,i])+"\n")

    f.close()
```

B Source code for HalfEdge.py

```python
import sys
import math
#import gmsh
import numpy as np

class TriangularMesh:

    def __init__(self, vertices, triangles):
        self.vertices = []
        j = 0
        for v in vertices :
            self.vertices.append(Vertex(v[0],v[1],v[2],j,None))
            j = j+1

        self.faces = []
        self.halfedges = []
        edges = {}
        j = 0
        index = 0
        for t in triangles :
            indices = [index, index+1, index + 2]
            index = index + 3
            for i in range (3) :
```

```
                self.vertices[t[i]].halfedge = indices[i]
                self.halfedges.append(Halfedge(indices[(i+1)%3], None, indices[(i+2)%3],
                ↪  t[i], j, indices[i]))
                edges[(t[i], t[(i+1)%3])] = indices [i]
            self.faces.append(Face(j,indices[0]))
            j = j+1
        for e,ind1 in edges.items() :
            if (e[1],e[0]) in edges :
                ind2 = edges[(e[1],e[0])]
                self.halfedges[ind1].opposite = ind2
                self.halfedges[ind2].opposite = ind1



class Vertex:

    def __init__(self, x=0, y=0, z=0, index=None, halfedge=None):
        self.x = x
        self.y = y
        self.z = z
        self.index = index
        self.halfedge = halfedge

class Face:

    def __init__(self, index=None, halfedge=None):
        self.index = index
        # halfedge going ccw around this facet.
        self.halfedge = halfedge

class Halfedge:

    def __init__(self, next=None, opposite=None, prev=None, vertex=None,
                 facet=None, index=None):
        self.opposite = opposite
        self.next = next
        self.prev = prev
        self.vertex = vertex
        self.facet = facet
        self.index = index


```