Assignment 2 : Free-surface 2D Fluid Solver Using Incompressible Euler's Equations

Garance Perrot

June 2024

The goal of this assignment is to implement a free-surface 2D fluid solver using incompressible Euler's equations. The final result is a video (link here) of a fluid simulation with fluid particles colored in blue and moving quite realistically in the frame.
Legend for abbreviations:
- N: size of point set
- rt: runtime of image generation
- it: number of iterations performed by LBFGS
- L: Left, R: Right
- LN: lecture notes

Lab 6 : Polygon Class & Voronoï Diagram

We use the same Vector class as in the first assignment, but fixing the third coordinate to 0 as we are in 2D. We create a class Polygon which is a vector of vertices (their order matters to define the edges), as well as a class Voronoi that gives the Voronoi diagram of a set of points.

To display a simple shape, we simply add vertices to a polygon and placing them accordingly. Then the function *save_svg()* (provided in the course) creates an svg file from the polygon.

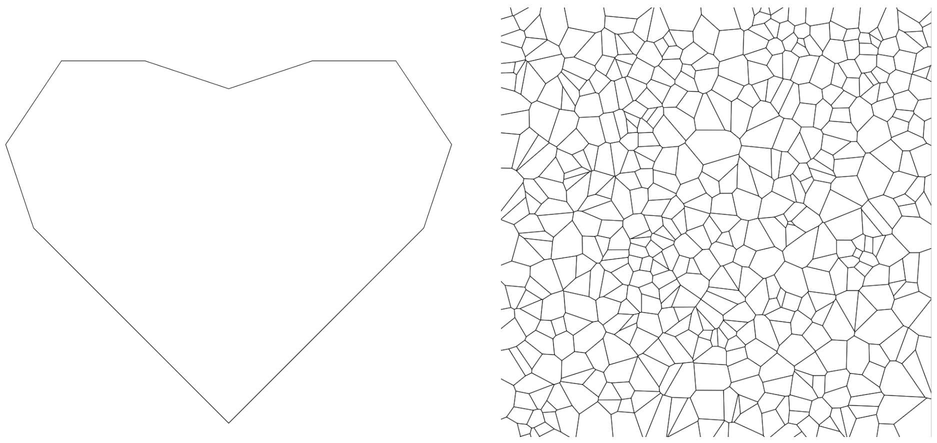

Figure 1: *L: A simple heart shape made by a Polygon of 13 vertices (rt: 0s)*
R: A Voronoï diagram (N = 500, rt: 0.178s in parallel VS 0.249s not in parallel)

To display the Voronoi diagram of a random set of N points, we create the Voronoi cell of each point separately (each cell can be processed in parallel which improves runtime). For each point, the Voronoi cell, represented as a Polygon, is initialized to a square (defined anticlockwise). Then, iterating over all the other points of the set, we clip the current cell with the bisector of the two points by applying the Sutherland Hodgman algorithm (slides p.16). To remove one half-space, we define a new empty polygon, iterate over its edges and check the two points's position with respect to the bisector. A point is inside if it satisfies the definition of the Voronoi cell (slides p.61), i.e by comparing normed distances, otherwise it is outside. If applicable, we compute the intersection point with the bisector and add it to the polygon for subsequent edge-clipping operations. In the end, we return the clipped polygon.

LAB 7 : Power Diagram & Optimal Transport With LBFGS

To extend the Voronoi diagram into a (Laguerre) Power diagram, we operate a few changes in the Sutherland-Hodgman algorithm. We change slightly the position of the middle point M, keep the same intersection P and change the definition of "X inside" to make it correspond to Power diagram properties by adding a weight contribution to the comparison of norms.

Initializing correctly the weights is crucial to compute a Power diagram. If the weights are all constant, the power diagram is similar to a Voronoi diagram as the weights cancel out in the comparison $||X - P_0||^2 - w_0 \leq ||X - P_i||^2 - w_i$, i.e checking if X is inside the Power Diagram cell. In order to initialize the weights randomly, we use a random engine (*std::default_random_engine*) from the <*random*> library instead of the *rand()* function from the <*cstdlib*> library in order to benefit from a uniform distribution. This way, the generated points are more evenly scattered, leading to smoother borders in the Power Diagram, as shown in Figure 2.

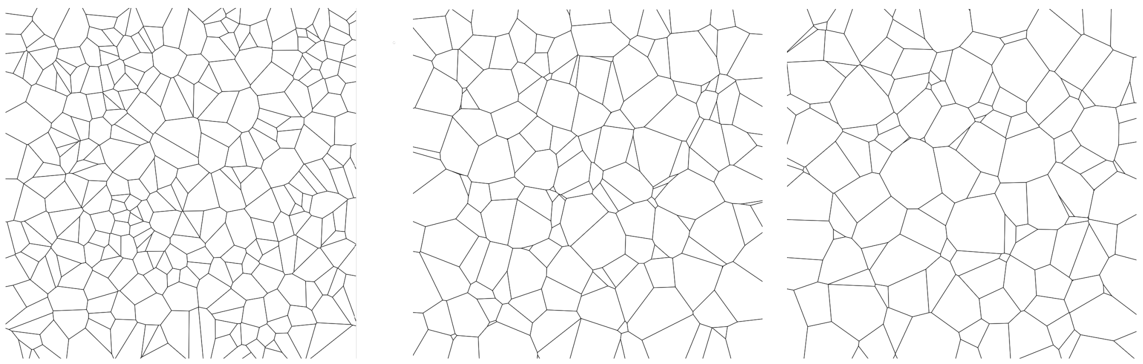

Figure 2: *L: A Power diagram with constant weights is similar to a Voronoi diagram (N = 300, rt: 0.064s, same result for weight = 1, 2, 100...).*
R: Power diagrams with random weights, the 2nd one with a uniform distribution (N = 20000, rt: 46.38s using rand() VS 43.70 using a random engine.

One should mention that a Power diagram and a Voronoi diagram of the same point set are not necessarily composed of the same number of cells. By essence, the Power diagram simplifies the partitioning by giving more influence to points with larger weights, resulting in fewer and potentially larger cells compared to the standard Voronoi diagram from a point set of the same cardinality.

The next step is to partition the space evenly between cells. We use the Semi-Discrete Optimal Transport method to find the optimal weights such that all the cell areas are equal to a constant λ (set to $\frac{1}{N}$). This problem is equivalent to maximizing a functional g (formula in *LN* p.102). In order to do so, we use the iterative LBFGS algorithm (Limited-Memory Broyden-Fletcher-Goldfarb-Shanno) which minimizes a continuous objective function, making sure that we switch the signs of the resulting function as our problem corresponds to gradient ascent for maximizing g.

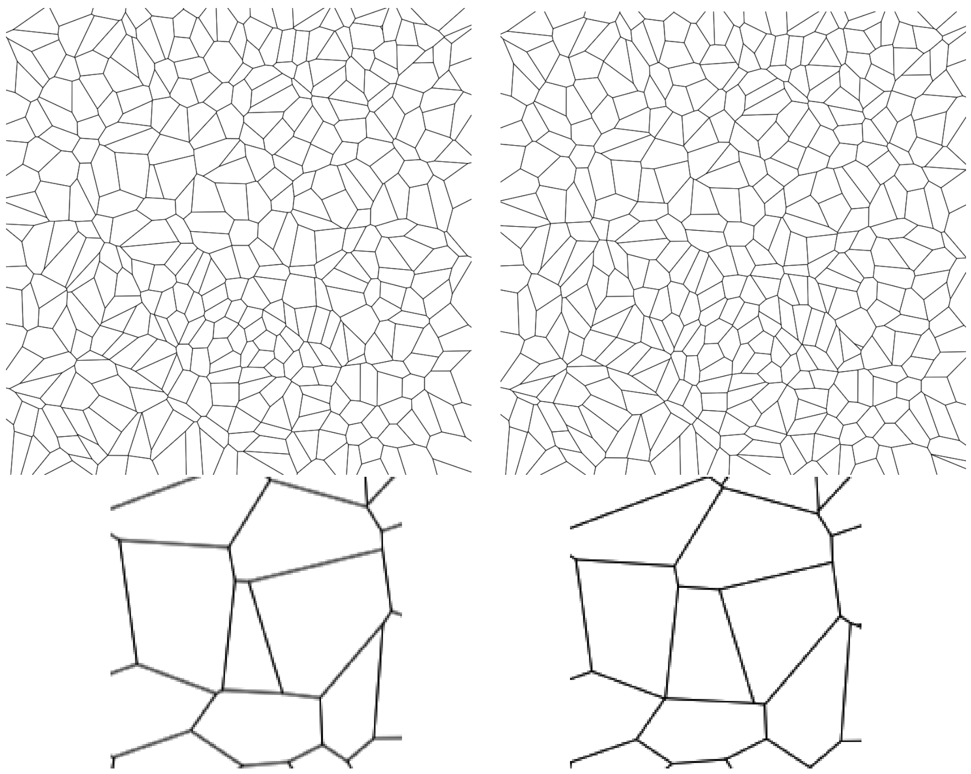

Figure 3: *Diagrams optimized using LBFGS with different parameters.*
(Top L: N=400, rt=1.322s, it=5, ϵ=0.01, ls=LBFGS_LINESEARCH_BACKTRACKING.
Top R : N=400, rt=6.181s, it=10, ϵ=1e-8, ls=LBFGS_LINESEARCH_BACKTRACKING_WOLFE).
Bottom : Zooms of the diagrams above.

It is important to set correctly the parameters for the LBFGS algorithm, such as the line-search algorithm, the maximum number of iterations allowed or the value of ϵ which is the convergence tolerance for the gradient norm. On Figure 3, the first diagram is optimized using a simple back-tracking linesearch with a very large tolerance and few iterations, and those elements can prevent the algorithm from converging to a good solution. In contrast, the second diagram relies on a more complex linesearch (it typically finds a more optimal step size) along with a lower tolerance and more iterations, allowing the optimization to perform better, as we observe that the area of cells tends to be more constant.

LAB 8: De Gallouet-Mérigot Incompressible Euler Scheme

Our next goal is to transform the cells into fluid particles. This is implemented by creating a class Fluid and calling a function *simulate()* that performs *nb_steps* times the Gallouët & Mérigot algorithm. Each time step consists in computing the current Optimal Transport diagram, then computing the vector fields of the points, adding forces (gravity and spring force that attracts a point to the centroid of its corresponding cell) to the velocities and then updating the positions of the points, making sure to bounce the particles back into the domain. We use the same parameters as in the slides p.17, namely $\epsilon = 0.004$ for the spring force, $m_i = 200$ the mass of a particle and $dt = 0.002$ for the time step. The *save_frame()* function provided in the course saves an image at each time step and allows to color the particles in blue.

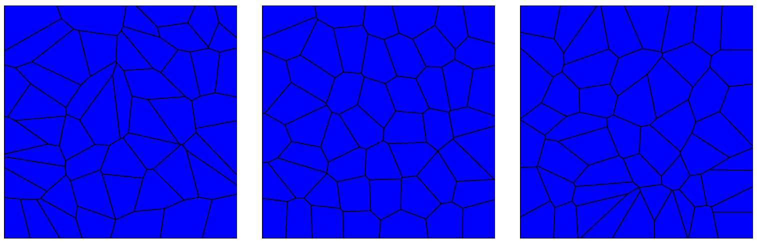

Figure 4: *Fluid particles moving, time steps 0, 40, 80 (N=50, rt=189.3s, avg_it=8, nb_steps=100).*

Next, in order to reshape the particles into circles, an additional clipping is performed for each cell during the computation of each Power diagram. We use a series of lines defined by points on a reference circle (line segments set to 20), otherwise the process is analogous to clipping by bisectors. Finally, we can divide the domain between air and fluid particles. During the LBFGS optimization, the desired area of fluid is set to 40% of the domain, the air fills the rest, and the estimated areas occupied by air and fluid particles are computed. All of these parameters are used for calculating the objective function and its gradient in order to ensure the correct repartition of particles across the domain. This corresponds to incompressibility, i.e the fluid's density remains constant throughout the simulation, even as it moves and deforms. The final result is a series of images of a fluid simulation that is combined into a video, realized using *ffmpeg* (20 frames/s).
Please let me know if you do not manage to open the video. After 159 iterations I lost my particles (white images) maybe due to memory overload in the code, but I did not manage to debug it.

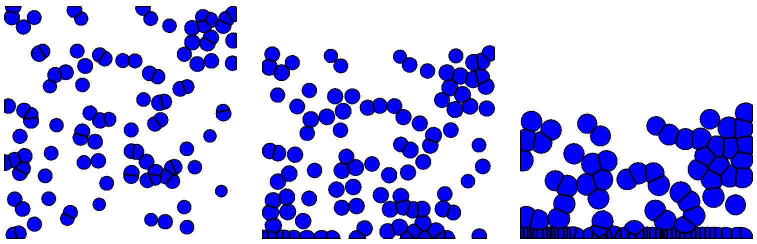

Figure 5: *Time steps 0, 80, 158 (N=100, rt=338.7s, avg_it=3, nb_steps=159, frame: 500×500).*

References

Similarly as for the first assignment, I got inspired a lot from live coding during the tds, LN, slides, but also did personal contributions and modifications. My goal was to successfully understand the concepts of the course as well as the role of each line of code in the overall process.