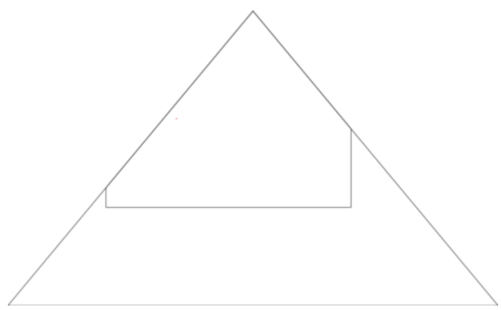
CSE 306 Project 2 report, Dimitrije Zdrale, BX 24

1. Polygon clipping

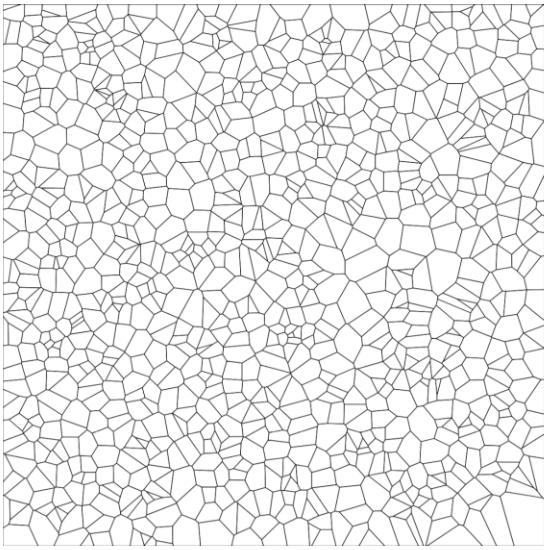
I started the project work by making a simple polygon clipping algorithm, known to us as Sutherland-Hodgman polygon clipping. However, before I started this, I needed to implement basic object classes, such as polygons, vectors, edges, as well as their important functions (containing/inside, intersection, vector operations). After these were done, I could implement the algorithm thanks to the lecture notes. To test it, I made a simple square on the background, and I cut it with a regular triangle, as seen on the image below:



Square clipped by a triangle

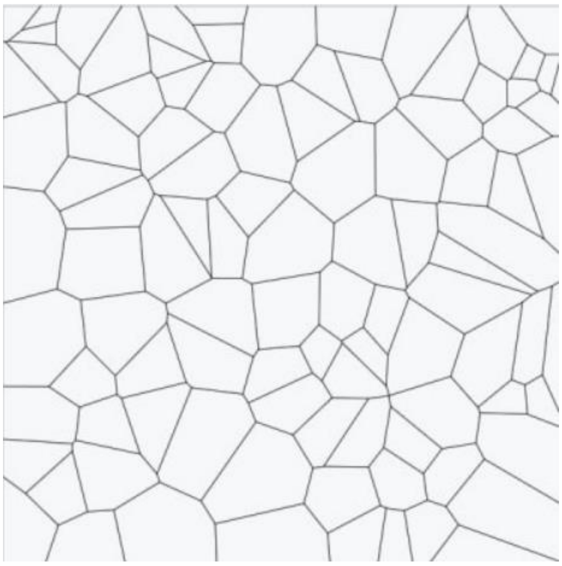
2. Voronoi diagram

With the clipping algorithm in place, I proceeded to do the Voronoi diagram implementation. The first step was to define new intersection and containment functions for my vectors and edges, as the M parameter needed to be modified to M'. This also required a new type of vector, which I named WeightedVector, however, I didn't have it as a separate class while I was doing the Voronoi diagram, but rather had a "weight" parameter in the vector class (later on this produced some confusion so I made the WeightedVector). After the new functions were done, I had to implement the Voronoi diagram generation. Firstly, I initialized "n" points randomly on the background, and then iterated over them, passing the current iteration point and the std::vector of all points to the clip_voronoi() function. The clip_voronoi() isn't much different than clip(), with the exception of using the new containment (inside_voronoi()) and new intersection functions (w_intersect), and sometimes using weighted vectors. When this was complete, I first made a Voronoi diagram. I obtained the following image:



1000 points

After playing with weight initialization, like setting them all to a constant value (0.1, 0.5, etc.), with the same number of points, I was getting images like the following one:



1000 points with constant weights

As you can see, some of the polygons got reduced/disappeared.
I decided to stop with the O(N^2) implementation and proceed with semi-discrete optimal transport.

3. Semi-discrete optimal transport and L-BFGS

This part was by far the most difficult for me, as I've spent the most time debugging and running it.

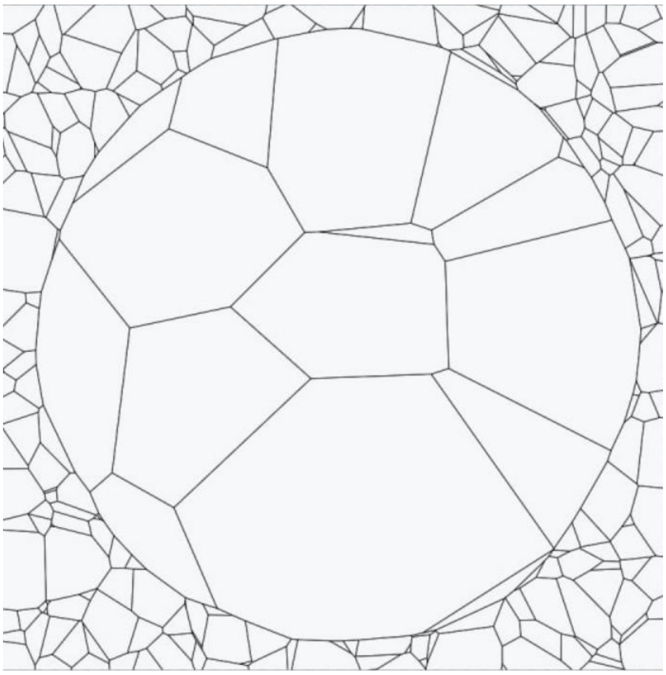
I began by obtaining the L-BFGS files from the provided github repository and linking them with my Makefile with the rest of the code. After this was done, I decided to change the sample.cpp provided in the repository to my needs.
Firstly, I had to decide on how the function will obtain all the necessary parameters. Since the points and the lambdas don't change during the gradient ascent, I decided

to pass them to the objective_function class instance. The lambdas are initialized as in the formula provided in the lecture notes, and I normalized them, as they should be a probability distribution. Points are generated in the same manner as in the Voronoi diagram. These two lists are then stored within the class and used later.

I then moved on to the evaluation function, the one where we implement the formulas.
First, I make a new set of points with updated weights, the x[i] from the library. These points were then used to generate another Voronoi diagram. Later on, for convenience (and after being told that the diagram generation should technically be in a separate class), I moved the diagram generation into a power_diag() function. After obtaining the power diagram of the current step, I proceeded to perform integral computation. Since we were given that f(x) = 1, after looking at the formulas, I realized that I could split the computation in the following manner: return
Fx = main_integral – weights*polygon_areas + lambdas*weights.
The main integral is done with triangulation. I will now explain every step of iteration over the points in the diagram.

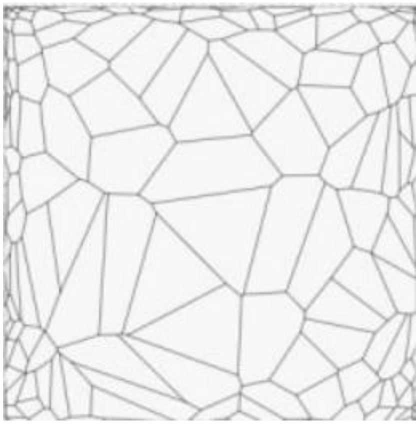
1) We compute the area of the current polygon using the provided formula (Shoelace/Gauss's formula) from the lecture notes.
2) We update the gradient with g[i] = area_of_polygon – lambdas[i]. Sign is flipped because we perform gradient ascent.
3) Next, I perform the triangulation operation. It is a function of the polygon class. Initially, I connected vertices to the "main point" of the polygon, the points[i]. However, after remembering the discussions on Slack, I decided to connect all vertices to the first one, vertex[0]. This function returns a std::vector of Triangle classes, which is, essentially, as simpler polygon class.
4) With the obtained triangles, I iterate over them and perform the area computation as specified in the lecture notes. I obtain the |T| by making a polygon out of the triangle vertices and computing it's area with the same function as from step 1).
5) This gives me the main_integral value for the certain point "i", so now, I increment the fx with (obtained value (integral_approx) – x[i] * area of the polygon + lambdas[i] * x[i]). As you can see, the exact split as explained above.
6) After we iterate over all the points "i", we return -fx as we want the gradient to ascend.

This is where the difficult part started for me. The lbfgs library failed after 2-5 iterations, and the results were bad, like the following:



So I started re-checking everything, as I really didn’t understand why this was happening.

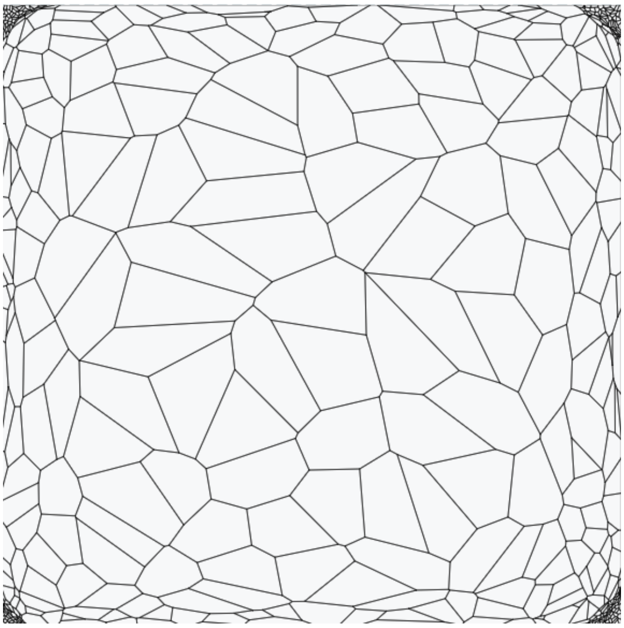
After a lot of debugging, code restructuring and re-writing many functions, the lbfgs started iterating, however, for a very long time.
I decided to try and run in with less points and see what happens. With 100 points, and after 1000-2500 iterations, depending on my initialization of m_x[i] (usually a constant), I would get images like the following:



100 points

I could now see that it begins to look like the image in lecture notes, but I needed more points. However, when running it on 1000 points, each iteration would take a long time (even though I tried parallelizing), and the algorithm wouldn’t stop even after 2500 iterations.
I did notice that fx didn’t change much, so I decided to set a hard upper-bound on the number of iterations. After some experimentation, I settled with 200.

With 200 iterations, the execution time was okay, and I got the following:



1000 points, 200 iterations

After all of the debugging I did, I was more than happy with this result. However, I didn't find out why my program needed so many iterations. It would terminate for 100 points with successful convergence, but for 1000 points it was too long.

4. Fluid dynamics

Firstly, I linked all the required files for saving frames with my project.

For the fluid animation, I made it as a class similar to the objective_function, with a step and run method.

Within the step, I perform all the necessary computations and updates to compute and apply the spring and gravitational forces, as well as update the position and velocity of the points and polygons. The objective_function is called with lambdas being the total volume divided by the number of points. The run function simply simulates the steps and prints our debugging statements. The points are also instantiated randomly.

I proceeded to play with many parameters, and then I merged all the collected frame images into gifs. Here are some of the results: youtu.be/e9emZEF8Sws

(Contact me if the link doesn't work).

5. Structure and compilation

Due to Visual Studio issues with the last project, I decided to do this one on the lab machines via ssh, so the structure is different. All the .cpp and .h files are in their respective relevant folders (classes, lbfgs, functions), and everything is compiled with a Makefile. To run the project, you can go into the project folder on a Linux machine, type "make" in the terminal, and then "./fluid". There are simple tests for all the major methods in the main.cpp file, they just need to be uncommented.