CSE306 Assignment 1

pedro.cabral

June 2023

1 Introduction

We provide a 2D fluid solver in C++ using a semi-discrete optimal transport approach. Notably, we implement:

1. An implementation (without the use of any external libraries) of kd-trees for nearest neighbor search.
2. A Voronoi diagram solver using Voronoi Parallel Linear Enumeration. Our clipping is accelerated with the use of kd-trees as described in the lecture notes.
3. The same for Power diagrams.
4. An optimizer for the weights of power diagrams using an (external) LBFGS library.
5. We implement a semi-discrete optimal transport fluid simulator using the features above.

2 Kd Trees

We implemented a Kd-Tree data structure inspired by the algorithm explained in a Computerphile video.

The kd-tree data structure is constructed with a set of N points. It has a method `std::vector <Vector> kNearestNeighbors( Vector P, int k )`[1] that takes a reference point `P` and the desired number of nearest neighbors `k`. For the nearest neighbor search, we used a Max-Heap data structure that keeps track of the k nearest neighbors found so far in the kd-tree traversal. When the current point in the traversal is at a smaller distance from our reference point than the root of the Max-Heap, we pop the root and insert his new point.

3 Voronoi Diagrams

In our current code, the Voronoi Diagram is implemented as a special case of the power diagram function with all weights equal to 0. We implement our Voronoi Diagram code in the file `gx_polygon.cpp`. The function `std::vector<Polygon> Voronoi(std::vector<Vector> points, Polygon bounding_box)` takes a set of points and a bounding box polygon enclosing all the points and outputs the cells corresponding to the voronoi diagram. When implementing the code, we noticed that the function failed when the points were highly structured (such as a regular $n \times n$ grid). Indeed, in such cases it happens frequently that points tested by the polygon clipping algorithm are exactly in the edges of the polygon. We found that the simplest solution was to dither the points by a small

[1] For simplicity, we do not include qualifiers in the function signatures in this report (such as `const` or the reference `&` symbol).

random amount (we used 10^{-12}). This way, the problem is fixed without resulting in any perceptible change to the output of the algorithm.

In order to show the result in Figure 1, we implemented a function `std::vector <Polygon> StandardizeCells( std::vector <Polygon> cells, Polygon bounding_box )` that shifts and scales the cells and the bounding box to fit the unit square. After that, we call the `save_svg` function.

As an experiment, we timed the voronoi diagram code with and without KdTree Acceleration. For $N = 10^5$ points, the code took $4.3s$ to run with KdTrees enabled and $8.07s$ without KdTrees.

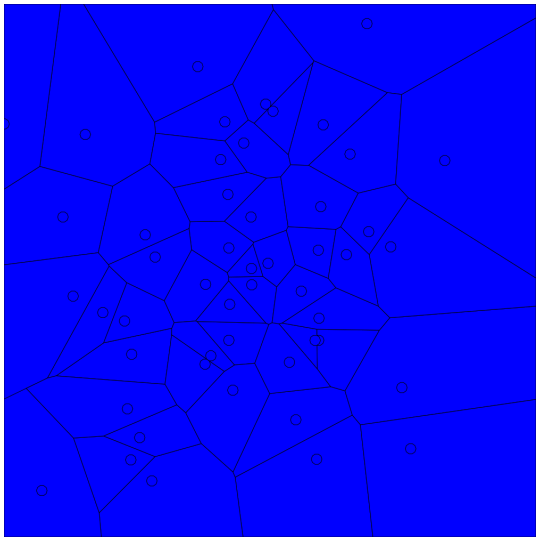


Figure 1: Voronoi Diagram of a set of 50 points.

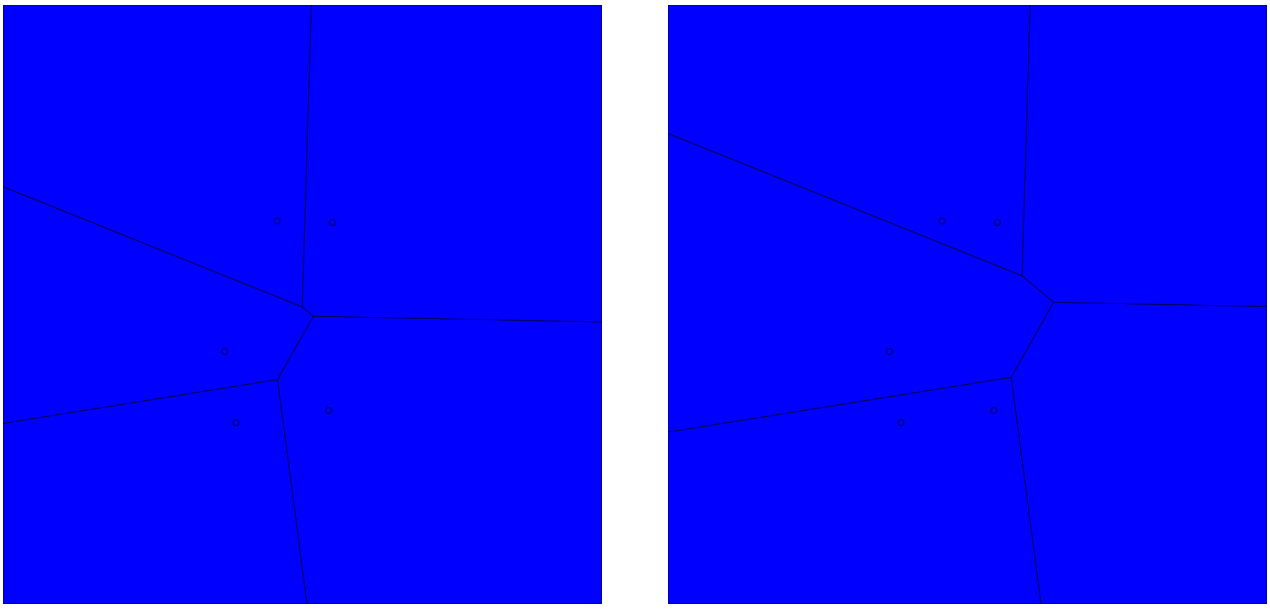
4 Power Diagrams

In this section, we explain the details behind the implementation of Power Diagrams. The file `gx_polygon.h` exposes the power diagram functionality through the `std::vector <Polygon> FastPowerDiagram(std::vector <Vector> sites, std::vector <double> weights, Polygon bounding_box)` function. However, most of the work happens inside the class `AcceleratedPowerDiagram`. In its constructor, this class initializes a 3D KdTree for the specified set of points (P_i). It takes the weights into account and sets the z coordinate of each point P_i to be of the form $\sqrt{M - w_i}$ for a large enough M.

The power cells are computed in parallel with an *omp for loop*. To compute the power cell corresponding to a point P_i, we start by clipping the bounding box with the $k = 20$ points in (P_j) closest to P_i. Notice that this search is performed in 3D even though we are constructing 2D power cells. After the clipping is completed, we have a polygon that we will refer to as the *candidate power cell*. We check the distance between P_i and each vertex of its candiate power cell, and we compare this value to the distance between P_i and the kth nearest neighbor (already computed). If the ratio of these two values is smaller than 4 (i.e. the kth nearest neighbor is far from P_i), we know that we can stop the algorithm and return the candidate power cell. Otherwise we double k and repeat. The procedure continues until it breaks, or until we check all N points.

For the polygon clipping algorithm, we use an adaptation of the Sutherland-Hodgman Polygon Clipping Algorithm (function `Polygon PolygonClip(Polygon subject, Polygon clip)`).

5 Optimization with LBFGS



(a) Before Optimization

(b) After Optimization

Figure 2: Power diagrams before and after optimization of weights. On the left we have all weights equal to 0. On the right we have the result of the LBFGS optimization with all parameters λ equal to each other.

In this section, we present our semi-discrete optimal transport code for transforming a uniform density into a weighted mixture of Dirac densities. More specifically, we consider a set of fixed weights $\lambda_1, \lambda_2, ..., \lambda_n$ together with a set of points $P_1, ..., P_n$ inside a bounding box $\mathcal{B}$ of area $|\mathcal{B}|$. We moreover assume that $\sum \lambda_i = |\mathcal{B}|$. Our optimization looks for weights $w_1, ..., w_n$ such that the power diagram of points (P_i) with weights (w_i) satisfies that the region $\text{Pow}_W(P_i)$ has area λ_i.

We use the LBFGS library to implement this optimization. We define an abstract class `ObjectiveLBFGS` in the file `ot.cpp` that serves as an interface to the lbfgs library. In the file `laguerre.cpp` we introduce the class `LaguerreSolver` that inherits from `ObjectiveLBFGS` and defines the function to be optimized and the gradient of the function. At every step of the optimization we compute the power diagram cells with the most updated weights (with functions `UpdateWeights, UpdateCells`), which in turn participate in the evaluation of the function and the gradient.

6 Fluid Simulation

We implement a fluid simulator using the optimal control code explained in the last section. Our approach consists in instantiating N particles of the fluid, each with position, velocity, and weight (x_i, v_i, w_i). Moreover, we also consider a weight corresponding to the air w_{air}. There are other parameters defined in the class `Fluid` such as the mass of each fluid particle m_i, the gravity vector g, the volume of the box occupied by the fluid, etc. We additionally implemented a `GradientAscent` function as an alternative to the LBFGS optimizer.

Each frame is computed from the previous one. In order to compute a given frame, we run the weight optimizer from the previous section through the class `FluidSolver` and obtain values for the (w_i) and w_{air}. Notice that `FluidSolver` also inherits from the abstract class `ObjectiveLBFGS`. After the weights are computed, we can generate the power cell corresponding to each particle (x_i, v_i, w_i). Assume that the centroid of the power cell corresponding to particle i is given by μ_i. The force acting on particle i consists of:

$$F_i = \left(\frac{\mu_i - x_i}{\epsilon^2} \right) + m_i g$$

We then update the velocity and the position of the particle using Euler's scheme:

$$v_i \leftarrow v_i + dt \frac{F_i}{m_i}$$
$$x_i \leftarrow x_i + dt v_i$$

where dt was set to 0.002.

Finally, notice that some particles may get outside of the bounding box. We that happens, we use our `BounceParticlesBack` routine. As an example, assume that a particle crosses the bottom side of the bounding box. We reflect the particle back into the box, changing the y coordinate sign of both the velocity and the position of the particle. Moreover, we scale these by a factor `k_elasticiy` that is smaller than 1. This way, the particles lose energy as time progresses.

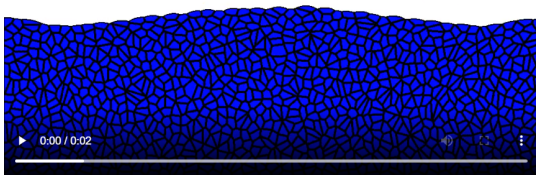


Figure 3: Fluid simulation video (see repository) with 1000 particles.