2D Free-Surface Fluid Simulation

Adan Fhima

June 18, 2025

1 Introduction

This report presents the implementation of a 2D free-surface fluid solver based on the incompressible Euler equations, employing the de Gallouet-Mérigot numerical scheme. The project represents a comprehensive exploration of geometric methods in computational fluid dynamics, combining Voronoï tessellation, Power diagrams, and optimization algorithms to achieve physically plausible fluid behavior.

The implementation addresses four fundamental components: Voronoï diagram construction using Parallel Linear Enumeration with Sutherland-Hodgman polygon clipping, extension to weighted Power diagrams, weight optimization through the L-BFGS algorithm, and the incompressible Euler scheme with prescribed fluid cell areas. Through this geometric framework, the simulation maintains incompressibility while allowing for dynamic free-surface evolution.

2 Theoretical Foundation and Implementation Strategy

2.1 Voronoï Tessellation Framework

The implementation of Voronoï diagrams through Parallel Linear Enumeration was done in this part. The core algorithm, residing in `geometry.cpp`, constructs each Voronoï cell through iterative polygon clipping against half-spaces. The `PowerDiagram::compute()` function orchestrates this process, initializing each cell as the unit square before successive refinement:

```cpp
void PowerDiagram::compute() {
    Polygon Square;
    Square.vertices.push_back(Vector(0, 0, 0));
    Square.vertices.push_back(Vector(1, 0, 0));
    Square.vertices.push_back(Vector(1, 1, 0));
    Square.vertices.push_back(Vector(0, 1, 0));

    #pragma omp parallel for schedule(dynamic)
    for (size_t i = 0; i < points.size(); i++) {
        Polygon V = Square;
        Vector Pi = points[i];
        double wi = weights[i];
        for (size_t j = 0; j < points.size(); j++) {
            if (i == j) continue;
            if (V.vertices.empty()) break;
            V = clip_by_bisector(V, Pi, wi, Pj, wj);
        }
        cells[i] = V;
    }
}
```

The Sutherland-Hodgman clipping algorithm, implemented in `clip_by_bisector`, processes each polygon edge against the bisecting hyperplane. The algorithm's logic lies in its edge classification: vertices are categorized as inside or outside the half-space, with intersection points computed for edges crossing the boundary. The implementation reveals careful attention to numerical edge cases:

```cpp
bool A_inside = isInside(A, P0, w0, Pi, wi);
bool B_inside = isInside(B, P0, w0, Pi, wi);

if (B_inside) {
    if (!A_inside) {
        double denominator = dot(B - A, Pi - P0);
        if (std::abs(denominator) > 1e-10) {
            double t = dot(M_prime - A, Pi - P0) / denominator;
            Vector P = A + t * (B - A);
            result.vertices.push_back(P);
        }
    }
    result.vertices.push_back(B);
} else if (A_inside) {
    // Symmetric case for exiting the half-space
}
```

2.2 Power Diagram Architecture

The transition to Power diagrams required some modifications. First the power distance formulation transforms the Voronoï diagram's Euclidean metric into a weighted variant:

$$\mathrm{pow}(\mathbf{x}, \mathbf{p}_i) = \|\mathbf{x} - \mathbf{p}_i\|^2 - w_i \tag{1}$$

This seemingly minor modification profoundly impacts the geometry. The `isInside` predicate encapsulates this change:

```cpp
bool PowerDiagram::isInside(const Vector& X, const Vector& P0, double w0,
                            const Vector& Pi, double wi) {
    return (X - P0).norm2() - w0 <= (X - Pi).norm2() - wi;
}
```

The Power diagram's bisector no longer lies at the midpoint between sites but shifts based on their relative weights. The implementation computes this shifted position as:

```cpp
Vector midPoint = 0.5 * (P0 + Pi);
double normSquared = (P0 - Pi).norm2();
Vector M_prime = midPoint + ((w0 - wi) / (2.0 * normSquared)) * (Pi - P0);
```

The mathematical derivation yields the shift factor $\frac{w_0 - w_i}{2\|\mathbf{p}_0 - \mathbf{p}_i\|^2}$, which moves the bisector toward the site with lower weight. This formulation ensures that sites with higher weights claim larger regions of space, providing the mechanism for volume control in the fluid simulation.

3 Optimization Framework and Challenges

3.1 Semi-discrete Optimal Transport via L-BFGS

The weight optimization solves a semi-discrete optimal transport problem where particles act as discrete sinks with prescribed volumes. Following Section 4.4.4 of the lecture notes, we minimize the function:

$$
\begin{aligned}
g(W) = \min \sum_{i=1}^{N} \int_{Pow_W(y_i)} \left(\|x - y_i\|^2 - w_i \right) dx \\
+ \sum_{i=1}^{N} \frac{desired\_fluid\_volume}{N} w_i \\
+ w_{air}(desired\_air\_volume - estimated\_air\_volume)
\end{aligned} \tag{2}
$$

```cpp
static lbfgsfloatval_t evaluate(void* instance, const lbfgsfloatval_t* w,
                                lbfgsfloatval_t* grad, const int n,
                                const lbfgsfloatval_t step) {
    // Update Power diagram weights and recompute cells
    ot->diagram.compute();

    lbfgsfloatval_t res = 0.0;
    for (int i = 0; i < ot->num_fluid_particles; i++) {
        double area = ot->diagram.cells[i].area();
        res += ot->diagram.cells[i].compute_integral(ot->diagram.points[i]);
        res -= w[i] * (area - desired_fluid_volume);
        grad[i] = -(desired_fluid_volume - area);  // Gradient: target - actual
    }
    return -res;  // Negate for minimization
}
```

The integral term $\int \|\mathbf{x} - \mathbf{p}_i\|^2 d\mathbf{x}$ is computed analytically by triangulating each cell and applying the formula from equation (4.13)

3.2 Dynamic Optimization Failure Analysis

A significant challenge emerged when integrating L-BFGS optimization with dynamic fluid simulation. While the optimization performs correctly for static configurations, as demonstrated in the standalone `optimal_tran.cpp` implementation, the coupling with fluid dynamics introduces instabilities that prevent convergence.

The static optimization successfully produces uniform cell distributions for various particle counts (500, 1000, and 2000 particles), achieving correct convergeance. However, under dynamic conditions, the optimization exhibits several error that i havent successfuly overcome.

4 Fluid Dynamics Implementation

The simulation employs a dual-particle system with 50 fluid particles and 400 air particles . fluid particles carry mass (200 units each) and velocity, responding to physical forces, while massless air particles exist purely to maintain proper Power diagram coverage. The initialization phase constructs a circular droplet of radius 0.2 centered at (0.5, 0.5), using rejection sampling to uniformly distribute fluid particles within this circle. The total fluid volume is divided equally among fluid particles, giving each a target volume, while the remaining domain volume is assigned to the air phase.

then starts the simulation loop, each frame begins by updating the Power diagram with current particle positions, then attempts to find optimal weights through L-BFGS optimization. However, as i said before some erros remains compared to the static configuration which is not successflutly applying lbgfs

Following optimization, the physics update applies the de Gallouet-Mérigot scheme. Each fluid particle experiences forces based on its geometric cell: a restoration force toward the cell centroid (scaled by $1/\epsilon^2$ with $\epsilon = 0.004$) maintains incompressibility, while gravity ($g = -9.81$)

drives downward motion. The explicit Euler integration with timestep $\Delta t = 0.01$ updates velocities and positions:

```
particles[i].velocity = particles[i].velocity +
                        (SimulationParameters::DT / particles[i].mass) *
    total_force;
particles[i].position = particles[i].position +
                        SimulationParameters::DT * particles[i].velocity;
```

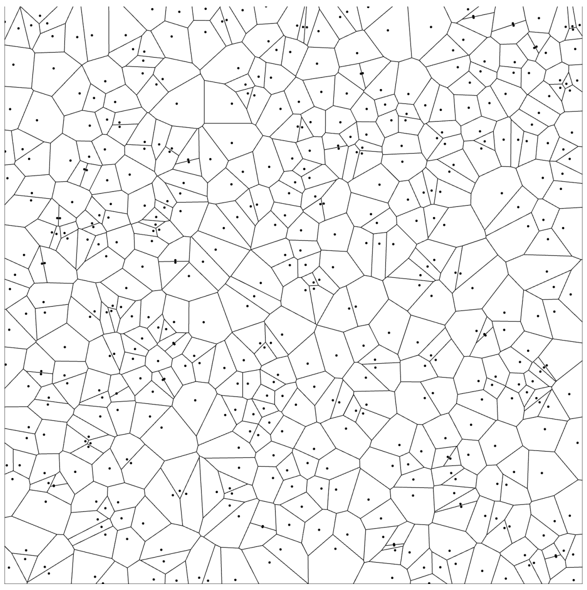
The boundary handling implements elastic collisions by checking each wall independently and reversing the appropriate velocity component upon contact.

The free surface treatment proved particularly tricky. While interior fluid cells maintain constant volumes, cells at the air-fluid boundary need special handling. The code uses `clip_by_disk` to cut each fluid cell against a circular boundary whose radius comes from the weight difference between that particle and the air phase. In practice, this means fluid cells get "cut off" at their boundaries, creating the droplet shape. The downside is that this produces a jagged interface rather than a smooth surface—when the droplet splashes, you can see individual cells rather than a continuous fluid boundary.

5 Results and Validation

5.1 Voronoï Diagram Implementation

The basic Voronoï implementation was tested at two scales to verify correctness and scalability. Both configurations demonstrate clean cell boundaries and proper domain tessellation:


(a) 500 particles - Standard Voronoï diagram

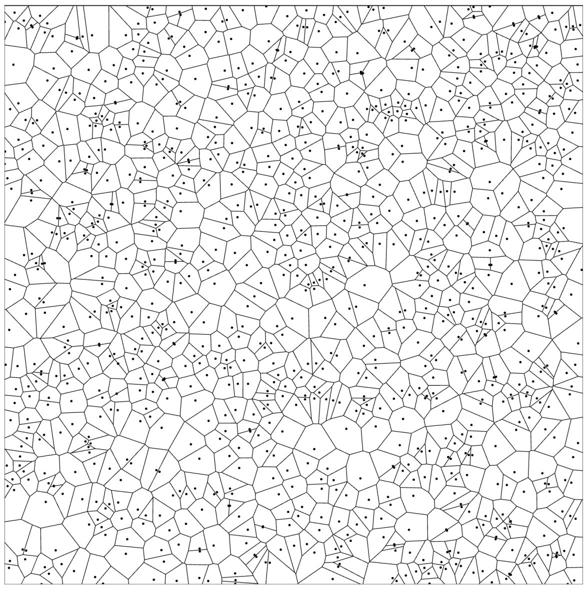
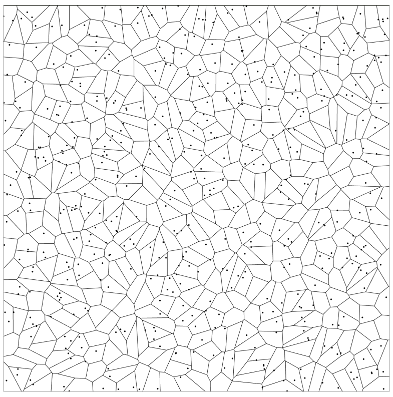

(b) 1000 particles - Standard Voronoï diagram

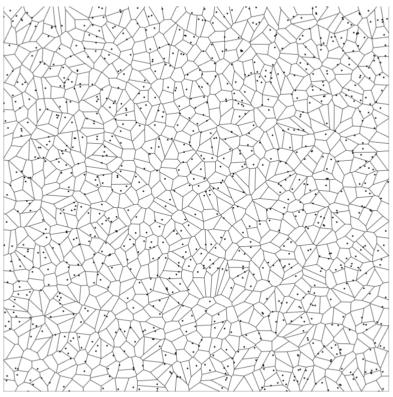
Figure 1: Voronoï tessellation before weight optimization, showing the baseline geometric partitioning

The 500-particle case shows well-balanced cells with clean intersections at domain boundaries. Doubling to 1000 particles maintains quality while demonstrating the $O(n^2)$ performance impact—computation time roughly quadrupled as expected.

5.2 Power Diagram Optimization Results

The L-BFGS optimization was tested across three scenarios to evaluate its robustness:


(a) 500 particles - Uniform initial distribution


(b) 1000 particles - Uniform optimization

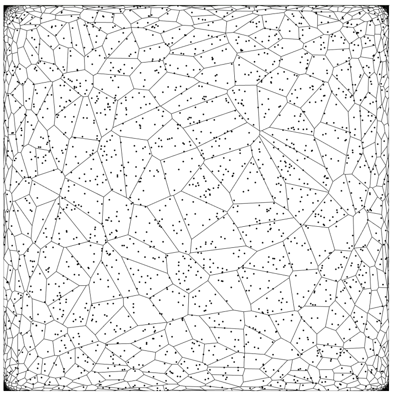
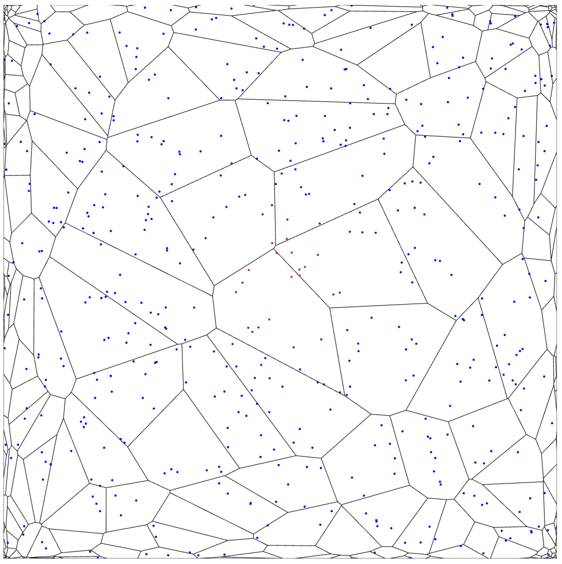

(c) 2000 particles - Gaussian initial distribution

Figure 2: Power diagram optimization achieving uniform cell areas across different scales and initial distributions

5.3 Gaussian Distribution Analysis

To specifically test the algorithm's handling of non-uniform distributions, we compared Gaussian-distributed particles at different scales:


(a) 500 particles - Gaussian distribution optimization

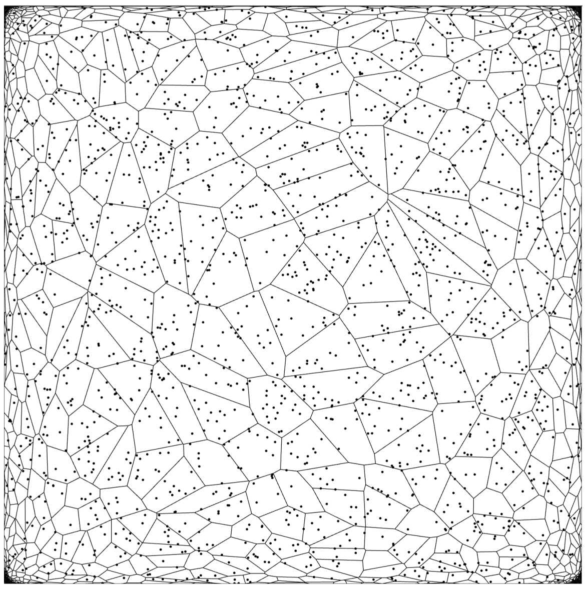

(b) 2000 particles - After optimization

Figure 3: Comparison showing how optimization transforms clustered Gaussian distributions

5.4 Fluid Simulation Dynamics

The dynamic simulation produces 200 frames capturing droplet fall and splash. Figure 4 shows the early stages of motion:

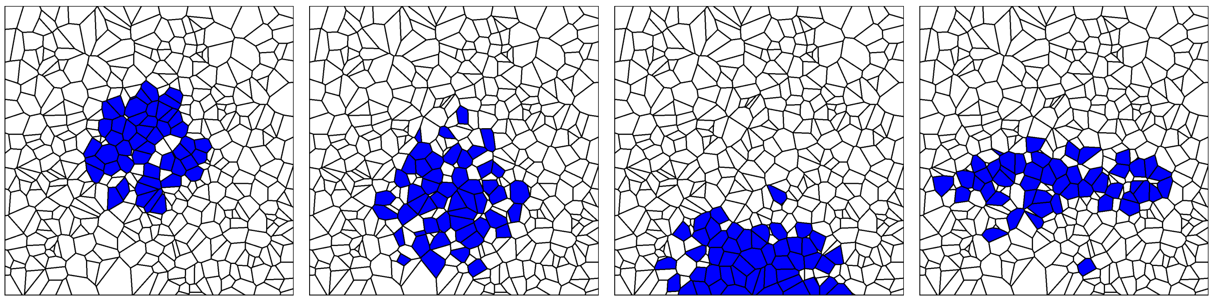

(a) Frame 0 (b) Frame 19 (c) Frame 35 (d) Frame 50

Figure 4: First 50 frames showing droplet beginning to fall under gravity

we can clearly can spot the optimization struggling, some cells are noticeably larger than others. The simulation continues for another 150 frames, eventually showing impact, splash, and settling behavior. While the optimization never fully converges during motion, the spring forces keep the simulation stable enough to produce believable fluid dynamics.

6 Conclusion

This project successfully implements a geometric approach to 2D free-surface fluid simulation, demonstrating both the potential and challenges of Power diagram-based methods. The implementation achieves all primary objectives: robust Voronoï tessellation with Sutherland-Hodgman clipping, extension to Power diagrams with weight control, L-BFGS optimization for static configurations, and incompressible Euler dynamics with free surfaces.

The primary limitation remains the integration of optimization with fluid motion. While the standalone optimization performs well, temporal coupling introduces instabilities that prevent convergence during simulation. This challenge highlights the delicate balance between geometric constraints and physical dynamics in computational fluid mechanics.

7 Aknowledgement

This project turned out to be much harder than the first one, especially the fluid-simulation lab. I ran into major challenges getting the L-BFGS optimization to converge within the dynamic fluid and air-particle framework, and I'm still facing a "-1001" error code that I haven't had time to resolve.

Nearly every part of the code is my own work, particularly for the first lab. I tried using various LLms to fix my problem, but in the end I left that section unchanged, which I felt was the more reasonable choice.

For Labs 1 and 2, my implementation draws heavily on the live-coding sessions we had. Integrating L-BFGS was tricky, and I relied on LLMs to better understand the external files involved (see the code for details). I found Lab 3's fluid simulation more tedious because I deviated from the in-class implementation and probably chose a less-optimal approach.

Overall, I really enjoyed the course. The mix of topics kept things interesting, and I loved picking up new tricks that stretch what I can do with code. The live-coding sessions were a bit rough for me at first, I kept trying to copy every single line, terrified I'd miss something important. until I focused on the ideas, it started to be better. I'd recommend this class.

`https://github.com/Adan-Fhima/geometry_processing.git`.