Simulation of Oil Spills Using Python

Mehdi Jamali, Ole Andreas Grøtting, Jørgen Svendsli

January 22, 2025

1 Overall Problem

The objective of this task is to develop a Python-based simulation framework that models the dynamics of oil spills over a two-dimensional domain.

The task involves creating a computational mesh composed of triangular and linear elements to model the flow of oil between these elements and its interaction with predefined velocity fields. At the start of the simulation, a Gaussian distribution will be used to represent the localized origin of the oil spill. As the simulation progresses, the oil will disperse based on flux computations derived from the geometry and velocity of each mesh element. The simulation will account for physical properties such as velocity fields and geometric constraints, allowing for accurate modeling of oil dispersion over time.

In addition to the simulation model, the task requires the development of visualization tools to represent the evolution of the spill and assess its impact on critical areas, such as fishing grounds and coastal zones. These visualizations will help analyze the spatial distribution of the oil and the extent of environmental damage.

The final deliverable will be a fully functional simulation framework, accompanied by a report that includes a detailed description of the methodology used to develop the model, the results of the simulation.

This task aims to demonstrate the application of computational techniques in environmental modeling and offers an opportunity to gain practical experience in programming and simulation.

2 User Guide

To run the simulation, follow these steps:

1. Install the required Python libraries, from `requirements.txt`

2. Configure the simulation parameters in the `config` file(s). The config file includes:

   - `meshName`: Name of the mesh file defining the simulation domain.
   - `tStart` (to set tStart, a restartFile must be provided), `tEnd`, and `nSteps`: Simulation time range and number of steps.
   - `borders`: Coordinates defining the fishing grounds.
   - `logName`: Filename/path for the mesh file.
   - `writeFrequency` (optional): How often it will save simulation step and image, if 0 no video will be provided.
   - `restartFile` (optional): Filename/path for the mesh file.

3. Execute the simulation by running `main.py`. The script accepts the following command-line arguments:

   - `-c` or `--config_file`: Path to the config file (default "input.toml")
   - `--find_all`: Find and run all config files in the main folder
   - `-f` or `--folder`: Specify the folder to search for config files (requires `--find_all`)

   The simulation can be initialized in two ways:

   - If a restart file is specified, the simulation will initialize from the saved state. (tStart must be set to match with the time in the given restartFile)
   - Otherwise, it will begin with the default oil distribution

4. For each configuration file, the simulation creates an output directory named `output_{logName}` containing:

   - Simulation state files
   - Generated images
   - Log file
   - Video output

   If no writeFrequency is given only the final simulation state will be given.

3 Code Structure

Our code is structured in multiple files in a "Simulation" package including the files, cells.py, mesh.py, Simulator.py, and Visualizer.py.

The mesh file as named, takes the mesh and processes it. It also includes the cell factory that sorts and makes the cells into Triangle and Line classes.

The cells file includes the Triangle class and here the geometric properties required for the triangle cells, such as edges, area, normals, and scaled normals are calculated. These properties were essential for the simulator's calculations; formulas are shown below.

$$\text{self.\_edges} = \begin{bmatrix} \text{self.points[1]} - \text{self.points[0]}, \\ \text{self.points[2]} - \text{self.points[1]}, \\ \text{self.points[0]} - \text{self.points[2]} \end{bmatrix} \quad \text{normal} = \frac{\begin{bmatrix} \text{edge[1]} \\ -\text{edge[0]} \end{bmatrix}}{\|\text{edge}\|} \tag{1}$$

$$\text{self.\_scaled\_normals} = [n \cdot \|\text{e}\| \mid n, \text{e} \in \text{zip}(\text{self.\_normals}, \text{self.\_edges})]$$

$$\text{self.\_area} = 0.5 \cdot edge1 \times edge2 = 0.5 \cdot |\text{edge1}[0] \cdot \text{edge2}[1] - \text{edge1}[1] \cdot \text{edge2}[0]|$$

Next in our Simulator file we have a main function called step() that takes the simulation to the next step. In short, it takes the old oil values. Then, for every cell and for every neighbor of the cell, it finds what edge id that cell shares with the current neighbor, with the find edge function. Then, it calculates the flux over that edge. After it has summed up the total flux over all edges of the triangle, it updates the oil level of the current cell and goes on to the next.

For finding the shared edge between the cell and current neighbor, the following formula was used 2:

$$\text{edge}_i = i \quad \text{where} \quad \begin{cases} S_p = P(cell) \cap P(ngh) \\ E = \begin{bmatrix} (P_0(cell), P_1(cell)) \\ (P_1(cell), P_2(cell)) \\ (P_2(cell), P_0(cell)) \end{bmatrix} \\ p_1, p_2 \in S_p \end{cases} \tag{2}$$

Then for that edge, the flux is calculated with 3:

$$f = \begin{cases} -\frac{\Delta t \cdot (cell_o) \cdot \phi}{A_c}, & \text{if } \phi > 0 \\ -\frac{\Delta t \cdot (ngh_o) \cdot \phi}{A_c}, & \text{otherwise} \end{cases} \tag{3}$$

cell_o is current cells oil level and ngh_o is neighbors oil level. The included calulations are: V(x,y) is the velocity function: $(y - 0.2 * x, -x)$

$v_i = V(c_x, c_y)$

$v_n = V(n_x, n_y)$

$\bar{v} = \frac{1}{2}(v_i + v_n)$

$\vec{n} = ScaledNormal$

$\phi = \bar{v} \cdot \vec{n}$

4 Agile Development

The development process for this project followed an agile methodology, emphasizing iterative progress and collaborative teamwork. The process consisted of the following stages:

4.1 Planning

The initial phase involved defining the scope of the project and prioritizing tasks. A Git board was used to organize tasks into categories such as "To Do," "Sprint backlog," "In Progress," and "Completed." This helped ensure that the team maintained focus on the most critical features while allowing flexibility for adjustments.

4.2 Incremental Development

The development process was divided into smaller, manageable increments. Each increment focused on implementing a specific feature or addressing a particular problem, starting with basic functionality, such as loading and visualizing the mesh, and advancing to more complex tasks like oil flux computation and visualization.

4.3 Collaborative Workflows

To avoid merge conflicts and streamline collaboration, the code base was organized into modules early in the project. This allowed team members to work on different components (e.g., simulation logic, visualization tools, or configuration management) simultaneously without interfering with each other's progress.

4.4 Story Mapping

Initially, we began with a single Python file containing all the simulation logic. Our first steps involved importing and visualizing the mesh, guided by exercises on neighbor calculation and plotting. These exercises, which were similar to our task goals, helped us generate a static image of the mesh and compute the neighbors for each cell.

From there, we focused on creating a basic working simulation, prioritizing functionality to run simulations. Subsequently, we implemented visualization of results and data analysis. As the project evolved, we added features to improve usability, such as input parsing and the ability to run multiple configurations with different settings.

However, as the single file grew larger, it became increasingly difficult to work efficiently, particularly in collaborative environments. Frequent merge conflicts arose when using tools like Git. Drawing on lessons from one of the lectures, we reorganized the code into a structured package, splitting it across multiple files. This reorganization significantly improved both collaboration and maintainability.

We also moved the plotting functionality to Visualizer.py, so we could see our plots by calling a function in Visualizer, so that we could see our work in the simulator file easier. This modular approach allowed us to proceed with the simulation confidently, knowing each component could be tested independently. The visualizer only plots and has to be given the current oil distribution state. The visualizer file is also where the video is created.

First step was to initialize the oil distribution to get the oil in the mesh. Then to actually simulate, made a step function that grabs all the old oil values, then for every cell it loops through every neighbor of that cell. It finds what edge of the current cell that is shared with the current neighbor and then calculates the flux on that edge between the cell and its neighbor. Finally, the step function takes the total flux of every edge of that cell and updates the cell's oil amount. We split up the "compute_flux" and "find_shared_edge" into their own functions to easier test where things are going wrong and make the step function more readable.

With the simulator finished we had the bulk of the work done. Next was testing and implementing the other functionalities required like finding and reading configs, parsing inputs and catching errors where they could come up.

Figure 1 shows the Git board used for tracking progress.

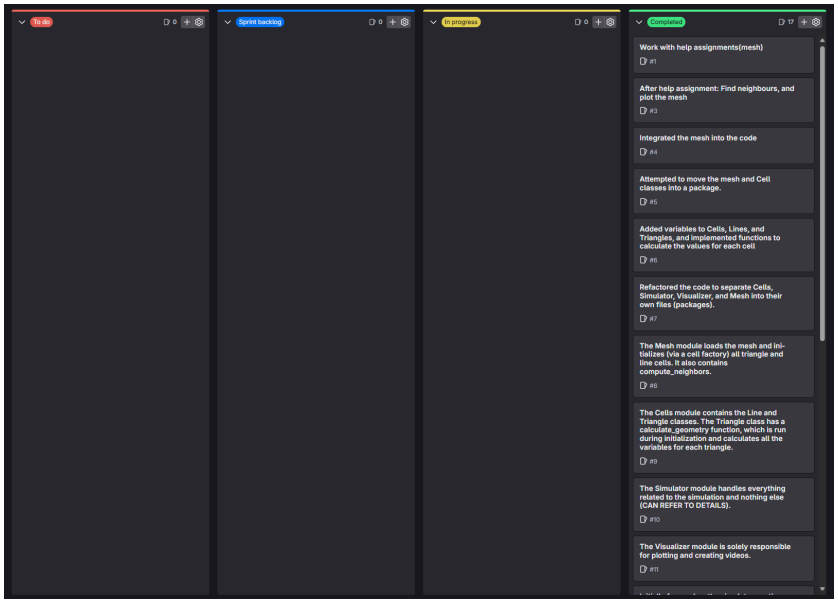

Figure 1: Git board illustrating the development process.

5 Results

The simulation was run under various configurations to evaluate its behavior. Figure 3 shows the initial oil distribution, while Figure 4 shows the simulation at time t = 0.5

Tests with varying time step sizes Δt revealed that larger steps reduce computational costs but may compromise accuracy, as seen in Table 1. Time steps in the lower range around 30 or lower made the simulation break down, and the oil did not go where expected. However we couldn't see too much of a difference in 500, 300 and 100 steps.

nSteps	Accuracy	Computation Time
500	High	Long
100	Moderate	Short
30	Low	Very Short

Table 1: Comparison of time step sizes and their effects.

6 Conclusion

This project demonstrated the use of Python for simulating oil spill dynamics using a mesh-based approach. The framework effectively models oil distribu-

tion, provides visual insights, and highlights the impact of parameter choices such as time step size. Future work could incorporate more complex velocity fields and real-world data to enhance realism.

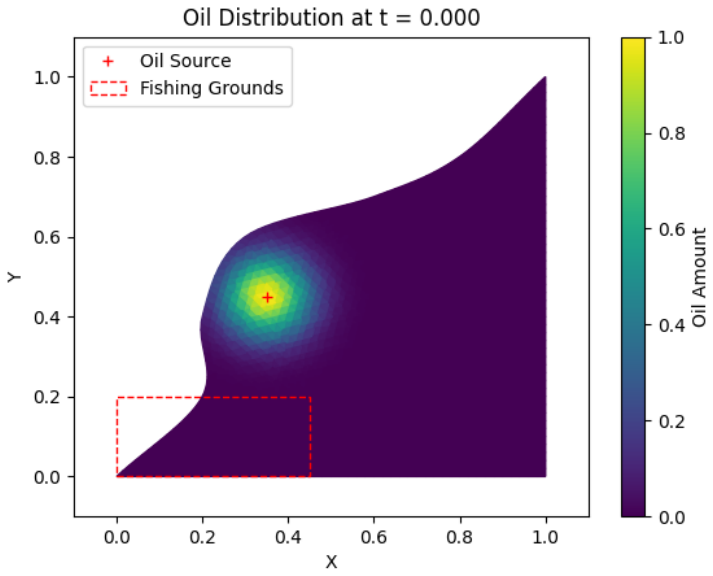

Figure 2: Initial oil distribution at $t = 0$.

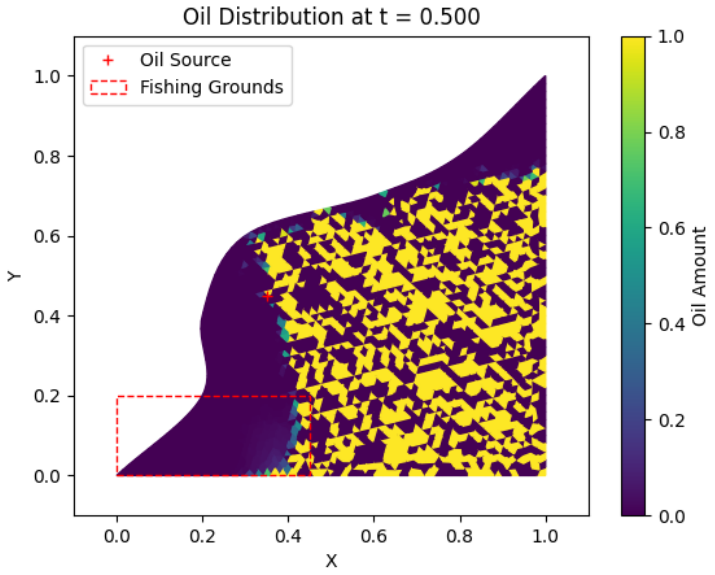

Figure 3: Initial oil distribution at $t = 0.5$. With 25 steps

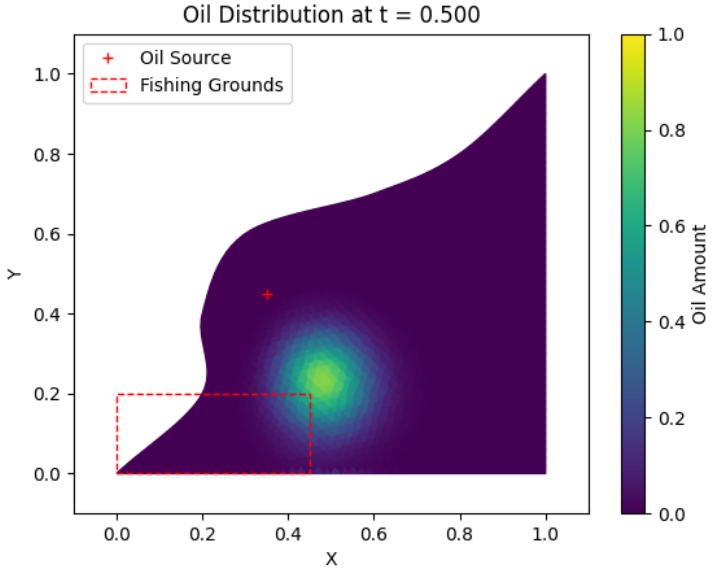

Figure 4: Oil distribution at time t = 0.5. With 300 steps.