Norwegian University of Life Sciences
(NMBU)

REPORT: SIMULATION OF AN OIL SPILL

Authors

Jørgen Asmundvaag
Ivar Eftedal
Sebastian Sverkmo

Norway, January 2025

Contents

1 Introduction

The task involves simulating an oil spill in Bay City caused by a ship leak. Key information provided includes the spill's location, the flow patterns of the spread, and a robust solution approach to predict its behavior. This simulation operates in a "2D world," where the computational domain is discretized into triangular cells. These triangles are constructed based on mathematical reasoning and serve as the foundation for capturing the behavior of the oil spill. Additionally, a mesh file is provided as the primary source of input data for the simulation.

The triangular patterns play a crucial role in predicting flow dynamics, as the oil moves edge-to-edge across adjacent triangles. This localized propagation reflects the influence of the underlying flow field on the oil distribution over time. By leveraging this approach, the task enables an effective and computationally feasible method to assess and predict the environmental impact of the spill.

2 Running the Project Code

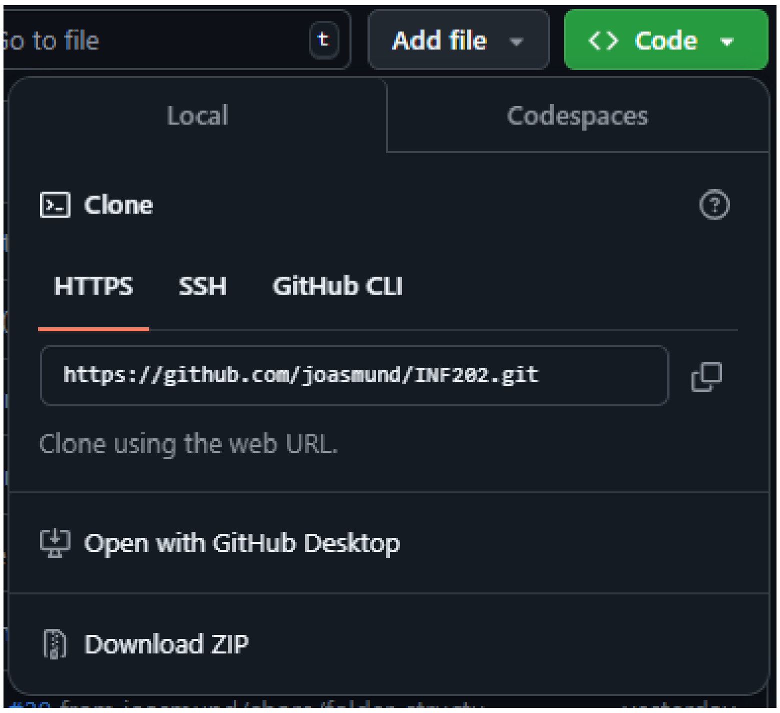
2.0.1 Prerequisites

Before getting started, ensure the following requirements are met:

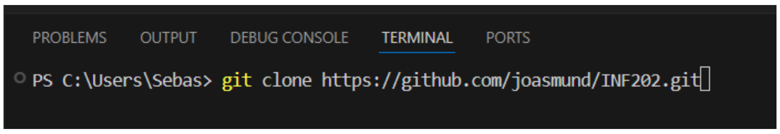
- You have downloaded the project repository to your local machine.
  - Clone the repository using the following command:

    ```
    git clone https://github.com/joasmund/INF202
    ```

  - Alternatively, you can download the project as a ZIP file from the repository page and extract it. - https://github.com/joasmund/INF202
  - Refer to the following image for guidance on how to clone the repository:



  - If you prefer to perform the cloning process manually using the terminal, see the example below:

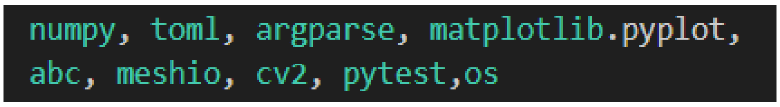


```
PS C:\Users\Sebas> git clone https://github.com/joasmund/INF202.git
```

- You have installed all necessary dependencies specified in the `requirements.txt` file that you will find in the program.
  - To install dependencies, use the command in the terminal:

    ```
    pip install -r requirements.txt
    ```

  - Additionally, refer to the following image for further context:



```
numpy, toml, argparse, matplotlib.pyplot,
abc, meshio, cv2, pytest,os
```

- We recommend using Python 3.12 or newer for optimal compatibility and performance.
- You have a code editor installed that supports Python development, such as Visual Studio Code (recommended).

2.0.2 Steps to Run the Code

1. Download the Project

- Clone the repository using Git or download it as a ZIP file and extract it to a directory on your computer.

2. Install Dependencies

- Open a terminal or command prompt and navigate to the root directory of the project.
- Install the required dependencies by running:

```
pip install -r requirements.txt
```

3. Navigate to the Main Script

- Locate the file named `main.py` in the project directory. This is the entry point for running the application.

4. Running the Code

You can run the code using one of the following methods:

a. Using the Terminal

- Open a terminal and navigate to the directory containing `main.py`.
- Run the script with the command:

```
python main.py
```

- To see all available command line arguments, use on of the following commands:

```
python -h
```

```
python --help
```

b. Using Visual Studio Code

- Open the project folder in Visual Studio Code.
- Locate `main.py` in the file explorer within the editor.
- Open `main.py` and click the "Play" button (usually displayed as a triangle icon at the top of the editor or in the debug section).

2.0.3 Notes

- Ensure your Python environment is correctly configured, and the version matches the project's requirements (e.g., Python 3.12 or newer is recommended).
- If you encounter any errors, verify that all dependencies are installed and that you are in the correct working directory.
- For further assistance, refer to the project's documentation or contact the team.

By following these steps, you should be able to run the project successfully and begin utilizing its functionality.

3 Formulas

3.0.1 Modeling

The simulation we are creating models the oil's evolution based on an initial distribution and a flow field representing ocean currents. The fishing grounds, located in $[0.0, 0.45] \times [0.0, 0.2]$, are the primary area of concern, with the aim of evaluating oil concentration over time.
The initial oil distribution is:

$$u(t = 0, \vec{x}) = \exp\left(-\frac{\|\vec{x} - \vec{x}_\star\|^2}{0.01}\right) \tag{3.1}$$

Here, $u(t = 0, \vec{x})$ represents the initial concentration of oil at point $\vec{x}$.
The flow field guiding the oil's movement is:

$$v(\vec{x}) = \begin{pmatrix} y - 0.2x \\ -x \end{pmatrix} \tag{3.2}$$

3.0.2 Cells

These conditions are applied to cells, which serve as the fundamental structure for organizing the field under investigation. In particular, these cells can be structured in the form of a triangle, referred to as **triangle cells**.

A triangle cell is defined by three points, denoted as $\vec{p}_1$, $\vec{p}_2$, and $\vec{p}_3$, which are connected by three edges: $\vec{e}_1$, $\vec{e}_2$, and $\vec{e}_3$.

$$\vec{e}_1 = \vec{p}_2 - \vec{p}_1, \quad \vec{e}_2 = \vec{p}_3 - \vec{p}_2, \quad \vec{e}_3 = \vec{p}_1 - \vec{p}_3 \tag{3.3}$$

Each triangle cell has a midpoint, calculated as:

$$\vec{x}_{\text{mid}} = \frac{1}{3}\left(\vec{p}_1 + \vec{p}_2 + \vec{p}_3\right). \tag{3.4}$$

The area of a triangle cell is given by:

$$A = 0.5 \cdot \left|(x_1 - x_3)(y_2 - y_1) - (x_1 - x_2)(y_3 - y_1)\right|. \tag{3.5}$$

Additionally, the outward-pointing normal vectors $\vec{n}_\ell$ must satisfy the following criteria:

The normal vectors $\vec{n}_\ell$ have unit length:

$$\|\vec{n}_\ell\| = 1. \tag{3.6}$$

The normal vectors are orthonormal to their corresponding facets:

$$\langle \vec{e}_\ell, \vec{n}_\ell \rangle = 0. \tag{3.7}$$

Another key quantity is the scaled normal:

$$\vec{\nu}_\ell = \vec{n}_\ell \cdot \|\vec{e}_\ell\|, \tag{3.8}$$

3.0.3 Flux and Edges

Given the modeling and structure of the cells, we can observe how the oil behaves. These conditions applied to edges and flux, will describe the exchange of oil between neighboring cells through shared edges.

To compute the flux $F_i^{(\mathrm{ngh},n)}$ for a triangular cell i at a given edge e_ℓ with a neighboring cell (denoted as ngh):

$$F_i^{(\mathrm{ngh},n)} = -\frac{\Delta t}{A_i} g\left(u_i^n, u_{\mathrm{ngh}}^n, \vec{\nu}_i, \ell, \frac{1}{2}(\vec{v}_i + \vec{v}_{\mathrm{ngh}})\right), \tag{3.9}$$

where $g(a, b, \vec{v}, \vec{\nu})$ is defined as:

$$g(a, b, \vec{v}, \vec{\nu}) = \begin{cases} a \cdot \langle \vec{v}, \vec{\nu} \rangle & \text{if } \langle \vec{v}, \vec{\nu} \rangle > 0, \\ b \cdot \langle \vec{v}, \vec{\nu} \rangle & \text{otherwise.} \end{cases} \tag{3.10}$$

For a triangular cell i with neighbors $\mathrm{ngh}_1, \mathrm{ngh}_2, \mathrm{ngh}_3$, the total oil amount at time t_{n+1} is given by:

$$u_i^{n+1} = u_i^n + F_i^{(\mathrm{ngh}_1,n)} + F_i^{(\mathrm{ngh}_2,n)} + F_i^{(\mathrm{ngh}_3,n)}. \tag{3.11}$$

Important notes for nodes

The nodes define the vertices of a triangular cell and are denoted by $\vec{p}_1, \vec{p}_2, \vec{p}_3$. These vertices are connected by the edges e_1, e_2, e_3 and possess the following key properties:

- **Position:** Each node is represented as $\vec{p}_i = (x_i, y_i)$, where $i \in \{1, 2, 3\}$, indicating its spatial coordinates.
- **Centroid and Area Contribution:** Nodes play a fundamental role in determining the centroid and area of the triangular cell.
- **Shared Connectivity:** Nodes can be shared among adjacent cells, serving as common vertices that connect neighboring elements in the mesh.

3.1 Pseudo-Code, formulas and actions

Algorithm 1 Triangle Operation, how we create a triangle

```
Require: A set of 3 points $pts$, an index $idx$, and a list of all points in the mesh $points$
Ensure: Functions for computing the area, center, midpoints, and edge normals of a triangle
  1. Area Calculation:
     Extract the 2D coordinates of the triangle vertices:
       coords_2d $\leftarrow pts$
     Let $(x_1, y_1), (x_2, y_2), (x_3, y_3) \leftarrow$ coords_2d
     Compute the area of the triangle using the determinant formula:
       $\text{Area} \leftarrow 0.5 \times \left| x_1 \cdot (y_2 - y_3) + x_2 \cdot (y_3 - y_1) + x_3 \cdot (y_1 - y_2) \right|$
     Output: Area is the absolute area of the triangle.
  2. Center Calculation:
     Compute the geometric center (midpoint) of the triangle:
       $\text{Center} \leftarrow \frac{1}{3} \cdot \left( (x_1, y_1) + (x_2, y_2) + (x_3, y_3) \right)$
     Output: Center is the centroid of the triangle.
  3. Midpoints of Edges:
     For each pair of adjacent vertices $(p_i, p_{i+1})$:
       Compute midpoint:
         $\text{Midpoint}(i) \leftarrow \frac{p_i + p_{i+1}}{2}, \quad i = 1, 2, 3$
     Ensure edge wrapping: $(p_3 \rightarrow p_1)$
     Output: List of all midpoints of the edges.
  4. Edge Normals:
     For each edge $(p_i, p_{i+1})$, calculate the normal vector:
       Let edge vector:
         $\text{Edge Vector}(i) \leftarrow (x_{i+1} - x_i, y_{i+1} - y_i)$
       Compute normal vector:
         $\text{Normal Vector} \leftarrow \frac{\begin{bmatrix} -(y_{i+1} - y_i) \\ x_{i+1} - x_i \end{bmatrix}}{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}$
     Repeat for all edges $(p_1 \rightarrow p_2)$, $(p_2 \rightarrow p_3)$, and $(p_3 \rightarrow p_1)$.
     Output: List of normalized vectors for all edges.
```

4 Code Structure

4.1 Module Overview

Understanding the architecture of the project is crucial for effective development and maintenance. This section provides an overview of the main modules that constitute the backbone of the simulation, outlining their roles and how they interact to achieve the desired functionality.
The project is organized around three main modules: `cell.py`, `factory.py`, and `mesh.py`. Each of these modules has a specific responsibility that contributes to the overall simulation process.

- **cell.py**: This module contains the definitions of various cell classes, such as `Vertex`, `Line`, and `Triangle`. These classes represent the geometric elements in the simulation and include necessary attributes and methods to handle oil volumes, areas, normal vectors, faces, velocity fields, and neighboring cells. Using mathematical calculations, we model the dynamics of the cells, update oil volumes, and compute flux to simulate oil transfer between cells.
- **factory.py**: The `CellFactory` class in this module implements the Factory Design Pattern, providing a flexible way to register and instantiate different cell type objects. This pattern makes it easy to extend the system with new cell types without altering existing code, promoting scalability and maintainability.
- **mesh.py**: This module manages the creation and administration of the geometric mesh on which the cells are based. It performs essential operations such as calculating cell areas, normal vectors for faces, and identifying neighboring cells. Furthermore, it initializes oil values and velocity fields based on the midpoints of the cells and coordinates the interactions between cells in the simulation. The `Mesh` class also gathers all necessary information for the simulation and registers the cells using the `CellFactory`.

This modular structure ensures a clear separation of concerns and promotes a maintainable and extensible codebase. By organizing the code in this manner, we can efficiently handle complex mathematical calculations and dynamic updates essential for an accurate and robust simulation of oil propagation.

4.2 Development Strategy

By utilizing this structure, you establish all the fundamental components before attempting to assemble the complete system. This approach is arguably the most effective strategy, as it allows for incremental testing to identify any issues within the foundational framework. Consequently, this method enhances the overall quality and reliability of the code. Additionally, such a strategy facilitates parallel development, where different team members can work on separate modules simultaneously without causing integration conflicts.

5 Project Planning and Structuring

When working on a group project, it is important to keep track of progress, both individually and as a group. To achieve this, we employed several methods for organization and tracking.

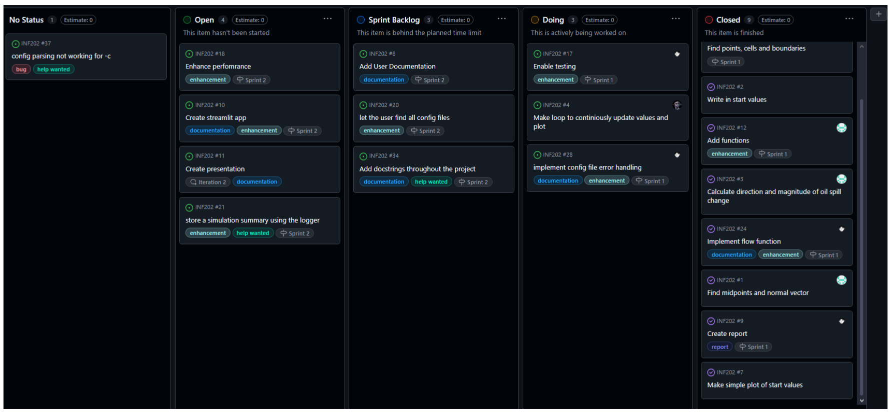
5.1 Initial Planning

After the task was assigned, we gathered as a group to:

- Review the task in detail.
- Share perspectives on effective ways to approach it.
- Sketch out ideas on a whiteboard, breaking the task into logical components step by step.

Once we were satisfied with the structure, we transitioned to a digital format:

1. A **Gitboard section** was created to list all tasks that needed to be done.
2. Team members selected the tasks they wanted to work on.



No Status 1 Estimate: 0

INF202 #37
config parsing not working for -c
bug | help wanted

Open 4 Estimate: 0
This item hasn't been started

INF202 #18
Enhance perfomrance
enhancement | Sprint 2

INF202 #10
Create streamlit app
documentation | enhancement | Sprint 2

INF202 #11
Create presentation
Iteration 2 | documentation

INF202 #21
store a simulation summary using the logger
enhancement | help wanted | Sprint 2

Sprint Backlog 3 Estimate: 0
This item is behind the planned time limit

INF202 #8
Add User Documentation
documentation | Sprint 2

INF202 #20
let the user find all config files
enhancement | Sprint 2

INF202 #34
Add docstrings throughout the project
documentation | help wanted | Sprint 2

Doing 3 Estimate: 0
This is actively being worked on

INF202 #17
Enable testing
enhancement | Sprint 1

INF202 #4
Make loop to continiously update values and plot

INF202 #28
implement config file error handling
documentation | enhancement | Sprint 1

Closed 9 Estimate: 0
This item is finished

Find points, cells and boundaries
Sprint 1

INF202 #2
Write in start values

INF202 #12
Add functions
enhancement | Sprint 1

INF202 #3
Calculate direction and magnitude of oil spill change

INF202 #24
Implement flow function
documentation | enhancement | Sprint 1

INF202 #1
Find midpoints and normal vector

INF202 #9
Create report
report | Sprint 1

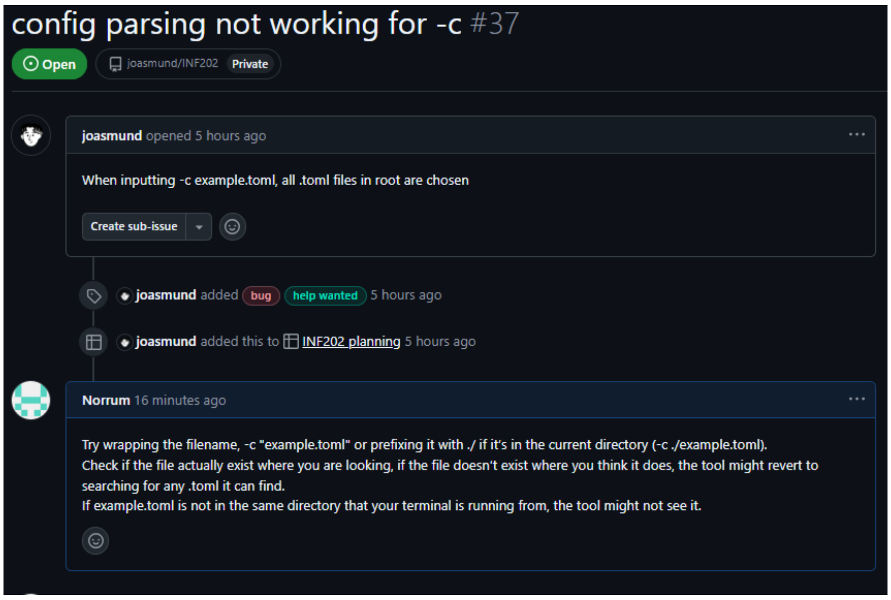
INF202 #7
Make simple plot of start values

Figure 5.1: *Screenshot of the Gitboard*

5.2 Collaborative Workflow

We implemented the following workflow to ensure efficient collaboration:

1. Each team member worked on their assigned tasks in *individual branches*.
2. Changes were pushed to their respective branches for review.
3. The group evaluated the solutions collaboratively, discussing:
   - Whether the solutions were effective.
   - Whether alternative approaches could be better.


config parsing not working for -c #37

joasmund opened 5 hours ago

When inputting -c example.toml, all .toml files in root are chosen

joasmund added bug help wanted 5 hours ago

joasmund added this to INF202 planning 5 hours ago

Norrum 16 minutes ago

Try wrapping the filename, -c "example.toml" or prefixing it with ./ if it's in the current directory (-c ./example.toml).
Check if the file actually exist where you are looking, if the file doesn't exist where you think it does, the tool might revert to searching for any .toml it can find.
If example.toml is not in the same directory that your terminal is running from, the tool might not see it.

Figure 5.2: *Screenshot showing collaborative evaluation of solutions in the Gitboard section.*

4. Approved solutions were merged into the `main` branch and used as the foundation for further development.

5.3 Benefits of the Approach

This structured approach provided several advantages:

- Clear organization of tasks and responsibilities.
- Seamless collaboration through GitHub branching.
- Continuous improvement of solutions through group discussions.
- A reliable and well-documented project structure.

5.4 Key Tools and Techniques

Whiteboarding: Used for brainstorming and breaking tasks into logical components.
GitHub: Leveraged for task management, version control, and collaborative coding.
Branching Strategy: Allowed team members to work independently while maintaining project cohesion.

5.5 Challenges with the Workflow

While the branch-push workflow provided many benefits, it also introduced a challenge related to the distribution of responsibilities. Specifically, the process of merging branches into the `main` branch occasionally led to an imbalance in contributions. Some team members naturally took on the role of reviewers and merge decision-makers more frequently than others. This created a slight inequality, as those who merged changes to `main` often spent additional time reviewing and ensuring the quality of the code, compared to others who primarily pushed their branches for review.

By implementing one or more of these strategies, we aimed to achieve a fairer distribution of responsibilities while maintaining the efficiency of our workflow.

5.6 Storymapping

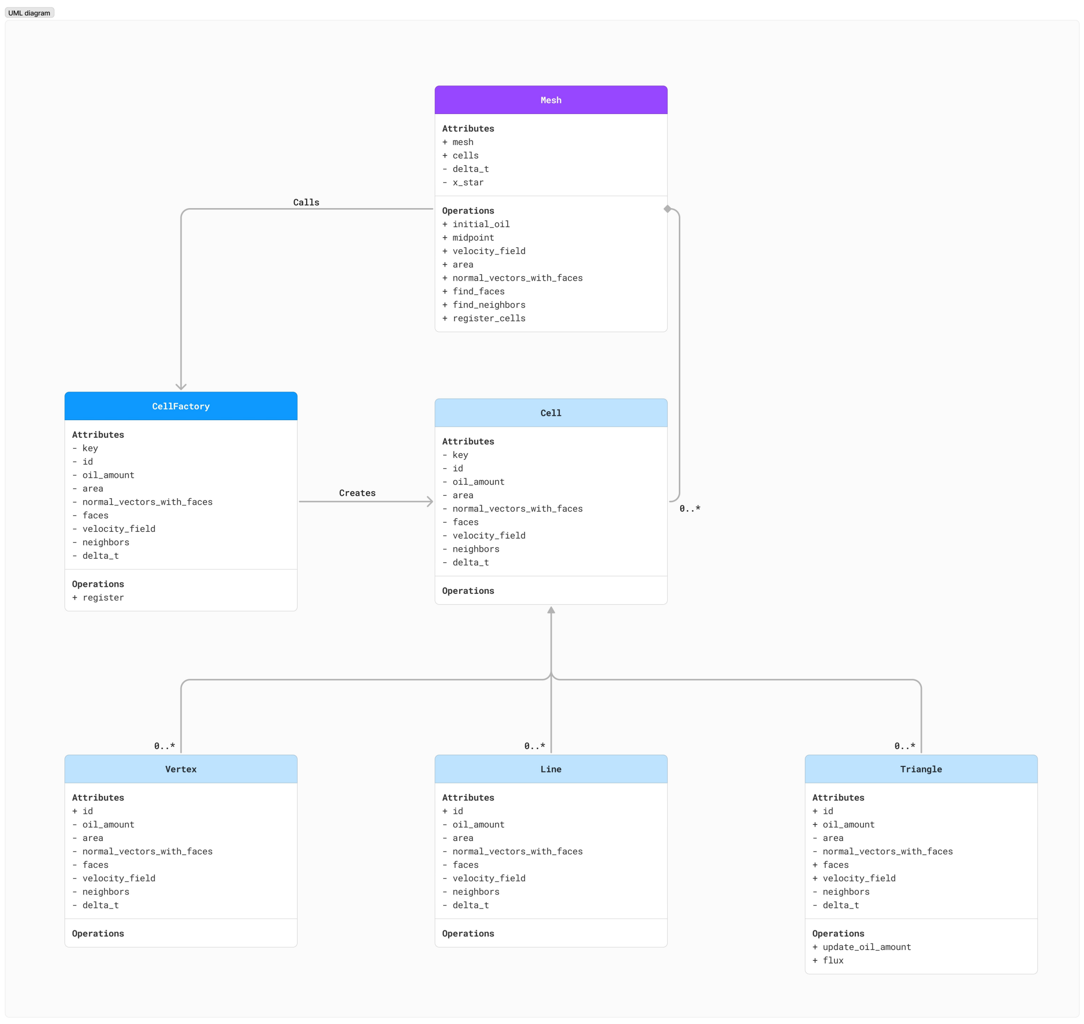


Figure 5.3: *The way we mappet out our code*

6

Simulation Results

Due to challenges in getting the simulation to function as intended, we were unable to produce meaningful results. As mentioned in the section below, we believe we have identified the potential reasons why the code is not performing optimally.

Therefore, we are not presenting our settings.

6.1 Visual Outputs

Here is the initial state of the simulation (see Figure 6.1), and the final state (see Figure 6.2):

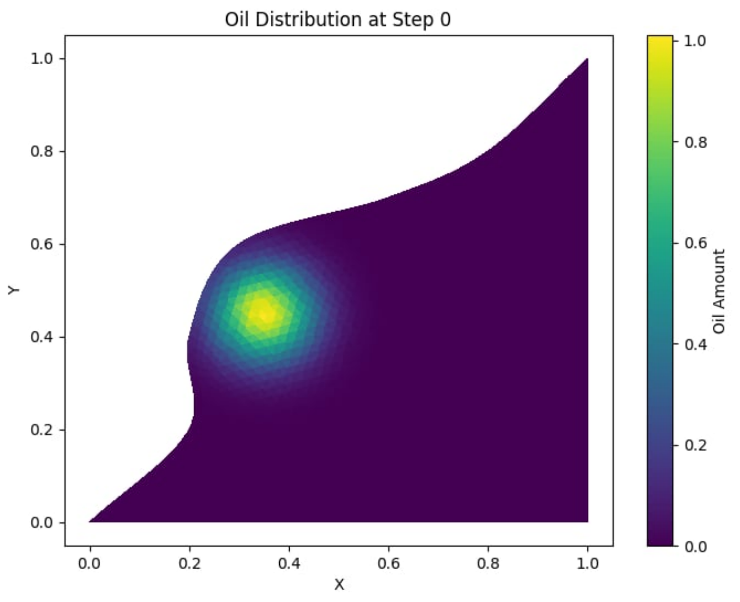


Figure 6.1: *Initial state of the simulation.*

We have included these images, which are located in the `Figures` folder, for reference and visualization of the problem.

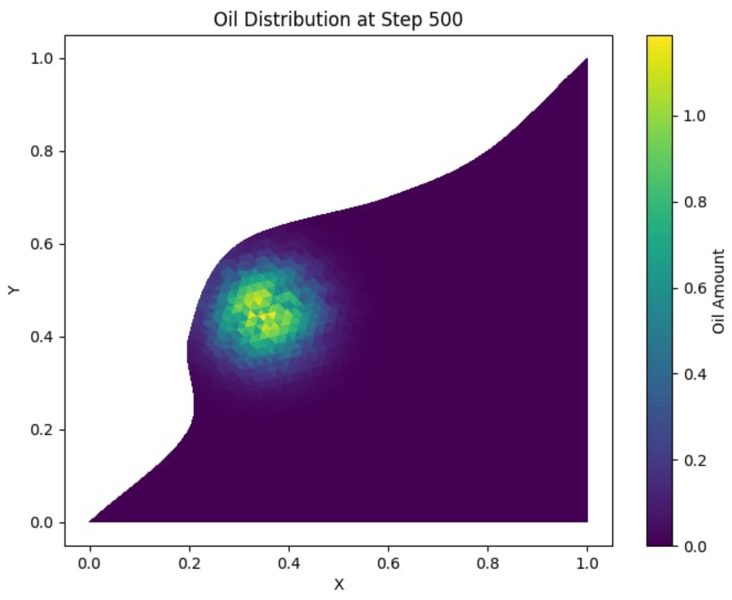


Figure 6.2: *Final state of the simulation.*

7 Improvements

During our project, we encountered significant challenges in getting the code to simulate as expected. Despite several days of debugging and troubleshooting, we were unable to resolve the issue completely. Our troubleshooting process included consulting teaching assistants (TAs), exploring online resources such as well-known coding platforms like Stack Overflow, and seeking advice from the professor.
After extensive investigation, we hypothesized that the issue lies in how previous values for the flow cells are managed. Specifically, it seems that the simulation fails to properly propagate the state of neighboring cells. Instead, the same cells repeatedly get updated, leading to the effect of filling the surrounding cells as if they were buckets, without correctly tracking the flow to adjacent areas over time. This behavior results in the simulation effectively "standing still" in one cell and incrementally filling its immediate neighbors, which does not align with the intended functionality.
We discovered this potential root cause just one day before the project deadline. Unfortunately, addressing this issue would likely require several additional days of development and testing, time that was not available to us before the submission deadline.