NORWEGIAN UNIVERSITY OF LIFE SCIENCES
(NMBU)

REPORT: SIMULATION OF AN OIL SPILL

MODELING OIL SPILL DYNAMICS: A COMPUTATIONAL APPROACH

AUTHORS

MIN JEONG CHEON
LE UYEN NHU DINH

INF203 - Advanced programming project

NORWAY, JUNE 2024

This page intentionally left blank.

Contents

This page intentionally left blank.

1

Overall problem

1.1 Motivation and Goal

According to "INF203 Task Description", Bay City faces an environmental crisis due to an oil spill threatening fishing grounds and farmers' livelihoods (Kusch, n.d.). Our team committed to assist the people in Bay City by developing strategies to predict and monitor the spread of the spill towards the fishing grounds.

Our primary goal is to understand the dynamics of the oil spill and create a simulation model capable of accurately predicting the oil distribution over time.

1.2 Project Outlines and Objectives

There are two primary elements in this simulation problem: the coastal area's geometry (where the potential oil spill may occur) and the dynamics of oil spill, which aid in estimating calculations.

1.2.1 Geometry of The Coastal Area

Given that the oil predominantly appears the ocean's surface, it's efficient enough to work with a two-dimensional representation of Bay City's coastal area through a computational mesh file labeled "bay.msh," as depicted in Figure 1.1. This mesh uses Cartesian coordinates with the origin at the lowest left corner.

The current oil distribution centers around the spatial point
$\vec{x}^\star = \begin{pmatrix} x^\star \\ y^\star \end{pmatrix} = \begin{pmatrix} 0.35 \\ 0.45 \end{pmatrix}$, while the fishing grounds occupy the region $[0.0, 0.45]\times[0.0, 0.2]$, with x-coordinates ranging from 0.0 to 0.45 and y-coordinates from 0.0 to 0.2.

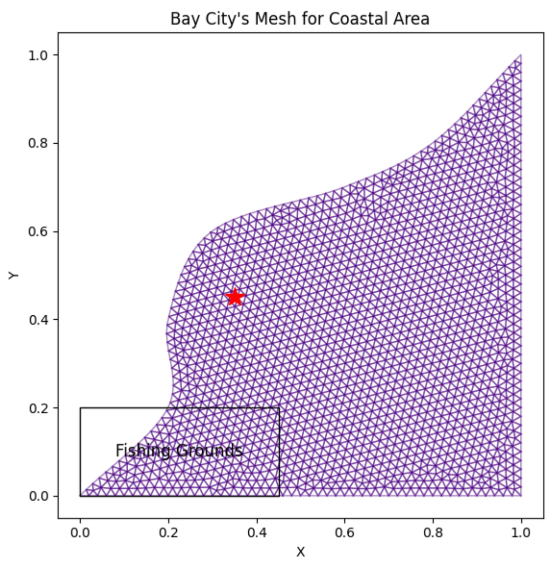


Figure 1.1: *The mesh grid of the coastal area in Bay City, where the oil distribution is stored.*

To compute the current oil distribution, we use the following expression:

$$u(t = 0, \vec{x}) = exp(-\frac{\|\vec{x} - \vec{x}^{\star}\|^2}{0.01}) \tag{1.1}$$

where $u(t = 0, \vec{x})$ represents the oil amount at a specific position $\vec{x}$ and time t.

However, we realized that it is computational-heavy and impractical to store all oil data for an infinite number of points within the bay mesh. Therefore, we divided the spatial domain into smaller subdomains known as cells. In this particular computational mesh, these cells has a typical form of triangle, created by points and line cells marking boundaries. Up to this moment, our objective is to predict the current (initial) oil distribution by (1.1). With the strategy of discretisation, the point $\vec{x}$ will be the midpoints of the respective cells.

1.2.2 Simulation by Dynamics of Oil Spill

The flow field

The movement of the oil is dictated by the underlying flow field which is represented in form of:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} y - 0.2x \\ -x \end{pmatrix} \tag{1.2}$$

Big Question: How can we simulate how this oil distribution evolves over time?

The oil flow changes over time. If we define the simulation starts at time 0 and ends at time t_{end}, then we can divide the time interval into N smaller intervals. The oil flow is updated during each time step $[t_n, t_{n+1}]$, where

$$t_n = n \cdot \Delta t \quad \text{and} \quad n = 0, \ldots, N - 1.$$

For a small time step $[t_n, t_{n+1}]$, the solution at cell i, denoted as $u[i]$, is updated based on the current cell and its neighboring cells. If the three neighbors of cell i are given as ngh_ℓ at edge ℓ for $\ell \in \{1,2,3\}$, the solution for cell i at time t_{n+1} can be calculated as follows:

$$u_i^{n+1} = u_i^n + F_i^{\mathrm{ngh}_1,n} + F_i^{\mathrm{ngh}_2,n} + F_i^{\mathrm{ngh}_3,n} \tag{1.3}$$

In equation (1.3), $F_i^{\mathrm{ngh}_\ell,n}$ denotes the amount of oil that is going to be updated in each cell$_i$ for neighbor ℓ. $F_i^{(\mathrm{ngh},n)}$ follows the equation below:

$$F_i^{(\mathrm{ngh},n)} = -\frac{\Delta t}{A_i} g\left(u_i^n, u_{\mathrm{ngh}}^n, \vec{\nu}_{i,\ell}, \frac{1}{2}(\vec{v}_i + \vec{v}_{\mathrm{ngh}})\right) \tag{1.4}$$

where g is given as

$$g(a, b, \vec{\nu}, \vec{v}) = \begin{cases} a \cdot \langle \vec{v}, \vec{\nu} \rangle & \text{if } \langle \vec{v}, \vec{\nu} \rangle > 0 \\ b \cdot \langle \vec{v}, \vec{\nu} \rangle & \text{else} \end{cases} \tag{1.5}$$

The vectors $\vec{n}$ have unit length, i.e., $\|\vec{n}\| = 1$. They are orthogonal to their corresponding face, that is, $\langle \vec{e}_\ell, \vec{n}_\ell \rangle = 0$, where $\vec{e}$ denotes each face of a triangle. A_i represents the area of cell i.

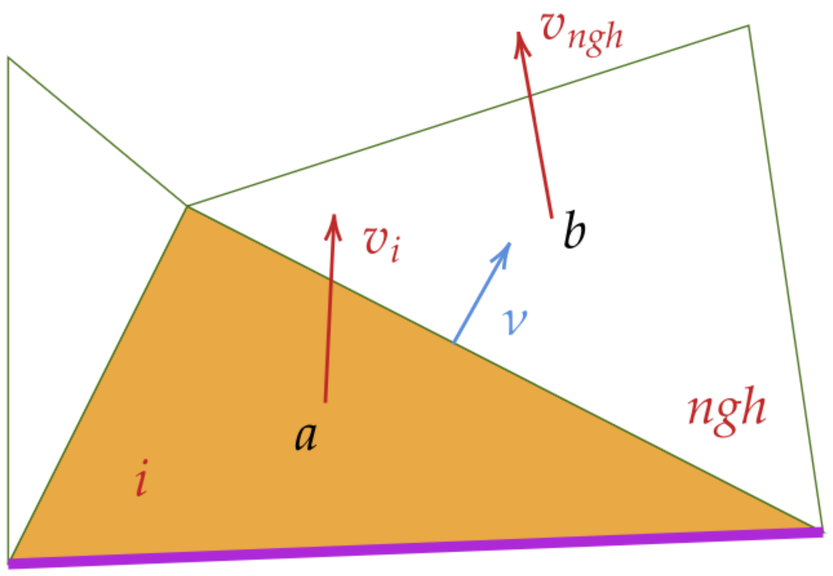


Figure 1.2: *Illustration of the flux on a given edge with neighbor ngh.*

1.2.3 Pseudo Code for updating the Oil Distribution

This Pseudocode below is the foundation to build the method "solution()" in class Simulation, which we will introduce later.

Algorithm 1 Oil Spill Simulation

```
Initialize:
for each cell in mesh_cells do
    u[index] ← compute initial oil using Formula (1.1)
    oil_distribution[index] ← u[index]
Updating with time:
for each step in num_steps do
    new_oil_distribution ← oil_distribution.copy()
    for each cell in mesh_cells do
        if cell is Triangle then
            total_flux ← 0.0
            for each neighbor in cell_neighbors do
                v ← computeAverageVelocity(cell, neighbor)
                n ← computeScaleNormal(cell, neighbor)
                dot_product ← dot(v, n)
                if dot_product > 0 then
                    g ← new_oil_distribution[cell_idx] * dot_product
                else
                    g ← new_oil_distribution[neighbor] * dot_product
                F ← -(dt / cell_area) * g
                total_flux ← total_flux + F
            new_oil_distribution[cell_idx] += total_flux
    oil_distribution ← new_oil_distribution
```

2

User Guide

The main.py script provides a solution for simulating and visualizing oil spills. This guide will walk you through the available functionalities of the code.

2.1 Software Installation

Set up Python Environment

1. Download and install Python from the Python website.
2. Install Visual Studio Code from the Visual Studio Code website.
3. Open Visual Studio Code.
4. Open the project folder provided by `Group37DinhCheon`.
5. Create a virtual environment by following these steps:
   (a) Press **Ctrl+Shift+P** to open the command palette.
   (b) Type **Python: Create Environment** and select it.
   (c) Choose a location for the virtual environment and confirm.

Install necessary packages Install necessary packages by opening the integrated terminal (Ctrl+‘) and running: `pip install requirements.txt`
Once you've completed these steps, your Python environment and Visual Studio Code will be set up to run `Group37DinhCheon`.

2.2 Running Simulations

To run the simulation and access its functionalities, execute the `main.py` script with appropriate command-line arguments. Below are the available options:
Command-line Arguments:

- `-c, -config`: Specifies the path to the configuration file. Default: `configs/input.toml`
- `-store-solution`: Option to store the solution as a text file.
- `-plot`: Option to plot the final oil distribution.
- `-video`: Option to create an oil distribution video.
- `-log-summary`: Option to log simulation summary.

- `-startTime`: Specifies the time to start the simulation from the restart file.

Example Usage: Running the command line below produces a plot showing the final oil distribution over time, and a video illustrating oil spread.

```
python main.py -c configs/input.toml --plot --store-solution --video
```

2.3 Output Interpretation

Upon executing the provided command line, the simulation automatically stores the resulting files in designated directories within the project structure. Users can conveniently access these files by simply navigating to the respective folders:
Configuration Files:

- Configuration files (`input.toml`, `config1.toml`, etc.) are stored in the `configs/` directory.

Simulation Results:

- Results for the default configuration (`input.toml`) are saved in the `results/input_results/` directory.
- Additional configuration-specific results are stored in their respective directories (`results/config1_results/`, `results/config2_results/`, etc.).

Folder for configuration files and results:

```
Group37DinhCheon/
    configs/
        input.toml
        config1.toml
        config2.toml
        config3.toml
    results/
        input_results/
            final_plot.png
            OilFishingSummary
            simulation_video.mp4
            solution.json
        config1_results/...
        config2_results/...
        config3_results/...
    main.py
    ....
```

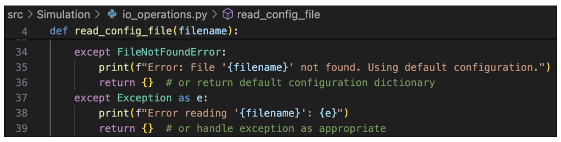
Accessing Results:
Users can simply navigate to the desired directory within the project structure to view and analyze the simulation results. Results include:

- Final oil distribution plots (`final_plot.png`) for visual analysis.
- Text files (`solution.json`) containing oil amounts per cell at each time step.
- Simulation videos (`simulation_video.mp4`) illustrating the oil spread over time.
- Logger (`OilFishingSummary`) that stores the amount of oil in the fishing grounds over time.

2.4 Error Handling

Common errors and troubleshooting steps:

- **Missing Config File:** Ensure that the specified configuration file exists at the provided path, for example `configs/...toml`.
- **Invalid Config File:** Check the configuration file for formatting errors or missing parameters. This issue is handled in the `read_config_file()` function in `io_operations.py`.



src > Simulation > io_operations.py > read_config_file

```python
def read_config_file(filename):
        except FileNotFoundError:
            print(f"Error: File '{filename}' not found. Using default configuration.")
            return {}  # or return default configuration dictionary
        except Exception as e:
            print(f"Error reading '{filename}': {e}")
            return {}  # or handle exception as appropriate
```

Figure 2.1: *Error handling while reading the configuration files*

- **Missing or Incorrect Mesh File:** Verify that the mesh file exists and is correctly specified in the configuration.
- **Initialization Failure:** Check for errors in the configuration parameters or simulation setup.
- **Plotting Failure:** Ensure that the required plotting libraries are installed and accessible.
- **Video Creation Failure:** Check the availability of video creation libraries and verify the video creation settings.
- **Logging Failure:** Verify the logging configuration and check for errors in the logging function.

3 Code Structure

3.1 Overview

The directory for the simulation core is:

```
Group37DinhCheon/
    src/ Simulation/
        bay.msh
        __init__.py
        mesh.py
        cells.py
        simulation.py
        visualization.py
        io_operations.py
        logger.py
    configs/
    results/
    tests/
    main.py
    README.md
    ....
```

The most important directory is `src/Simulation/`, which contains the essential code for computing oil distribution and various functionalities available to users.

- `mesh.py`, `cells.py`, and `simulation.py` are Python files that contain all necessary classes, as shown in the figure below.
- The three other Python files contain the required functionalities:
  - `visualization.py`: Generates a plot of the oil distribution at any given time and creates a video of the oil distribution over time.
  - `io_operations.py`: Reads input from configuration files, stores the solution as a text file, and restarts the solution at a given time using the stored solution file.

  - `logger.py`: Outputs all parameters used for the simulation (specified in the toml file) and tracks the amount of oil in the fishing grounds over time.

The relationship between the classes and their attributes and methods

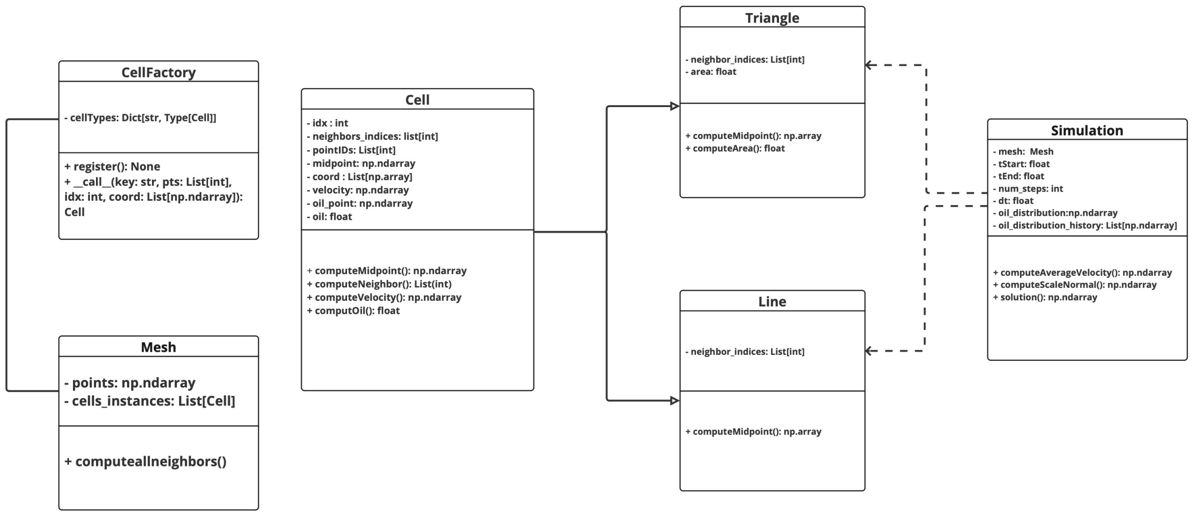

CellFactory
- cellTypes: Dict[str, Type[Cell]]
+ register(): None
+ __call__(key: str, pts: List[int], idx: int, coord: List[np.ndarray]): Cell
Mesh
- points: np.ndarray
- cells_instances: List[Cell]
+ computeallneighbors()
Cell
- idx : int
- neighbors_indices: list[int]
- pointIDs: List[int]
- midpoint: np.ndarray
- coord : List[np.array]
- velocity: np.ndarray
- oil_point: np.ndarray
- oil: float
+ computeMidpoint(): np.ndarray
+ computeNeighbor(): List(int)
+ computeVelocity(): np.ndarray
+ computOil(): float
Triangle
- neighbor_indices: List[int]
- area: float
+ computeMidpoint(): np.array
+ computeArea(): float
Line
- neighbor_indices: List[int]
+ computeMidpoint(): np.array
Simulation
- mesh: Mesh
- tStart: float
- tEnd: float
- num_steps: int
- dt: float
- oil_distribution:np.ndarray
- oil_distribution_history: List[np.ndarray]
+ computeAverageVelocity(): np.ndarray
+ computeScaleNormal(): np.ndarray
+ solution(): np.ndarray

Figure 3.1: *The diagram showing relationship between classes in Simulation Core*

3.2 Class and Functions Descriptions

The Cell Factory class, creates instances of different cell types based on the key telling whether the cell is triangle or line. It includes a dictionary _cellTypes that maps keys to cell type classes. The call method finds the corresponding cell type in the _cellTypes dictionary and creates a new cell object that contains function of vertexes' indices, cell index, and coordinates of vertices.

The Cell class, is an abstract class for different cell types. All cell instances has attributes including pts (index of the vertexes), coord(coordinates of those vertexes), and idx(index of the cell). It provides different methods:

- `computeNeighbor()`: Find the index of neighbor cells to each cell, Line cell has 2 neighbors (indices) while Triangle cell has 3.
- `computeMidpoint()` (an abstract method): Find the coordinates of midpoint in each cell.
- `computeVelocity()`: Find the flow field / velocity of each cell based equation (1.2)
- `computeOil()`: Find the initial amount of oil in each cell

This cell class is inherited by two child class, namely triangle and line cell.
The Mesh class, reads mesh data and create instances of cells and call the function to compute the neighbors for each cell.

Properties	**Classes: Triangle vs Line**	
Similarity	Both are a child class of Cell class with the aim to represent the cell type and inherit the methods from the mother class. Both have `__str__` method to describe the cell with its neighbor indices.	
Differences	The area is computed only in Triangle.	The line cells do not have a value for area, which leads to the idea that the flux on a line as a current cell cannot be updated.

Table 3.1: *Comparison of Triangle and Line Class Properties*

The Simulation class, simulates the movement and distribution of oil within a mesh over a time interval. It initializes the oil distribution dictionary to save the amount of oil in each cell. And its oil distribution history is updated in each small time step dt, by appending the calculated mesh's oil distribution at the current step. This process follows the section "Pseudo Code for updating the Oil Distribution".

3.3 Reasons Behind Our Structure

Based on our understanding of oil spill dynamics, flow field and computational formulas, we acknowledge that properties such as neighboring indices, midpoint coordinates, velocity (flow field), area, and initial oil amount are derived exclusively from cell attributes (vertex indices, cell index, and vertex coordinates). However, necessary properties for predicting oil distribution, such as scaled normal and average velocity, require computation based on attributes of the current cell and its neighbors. Therefore, we made the decision to place these functions/methods within the Simulation class, which consolidates information from the Mesh class, Cell class, and its subclasses (Triangle and Line).

4 Agile Development

4.1 Project Management Approach

After understanding the computation process of the oil spill and exploring with the mesh file of Bay City, we divided the requirements of the task to smaller and manageable tasks. Then we assigned each of us on some of the tasks. Using the story mapping, our general task are shown in figure 4.1 below.

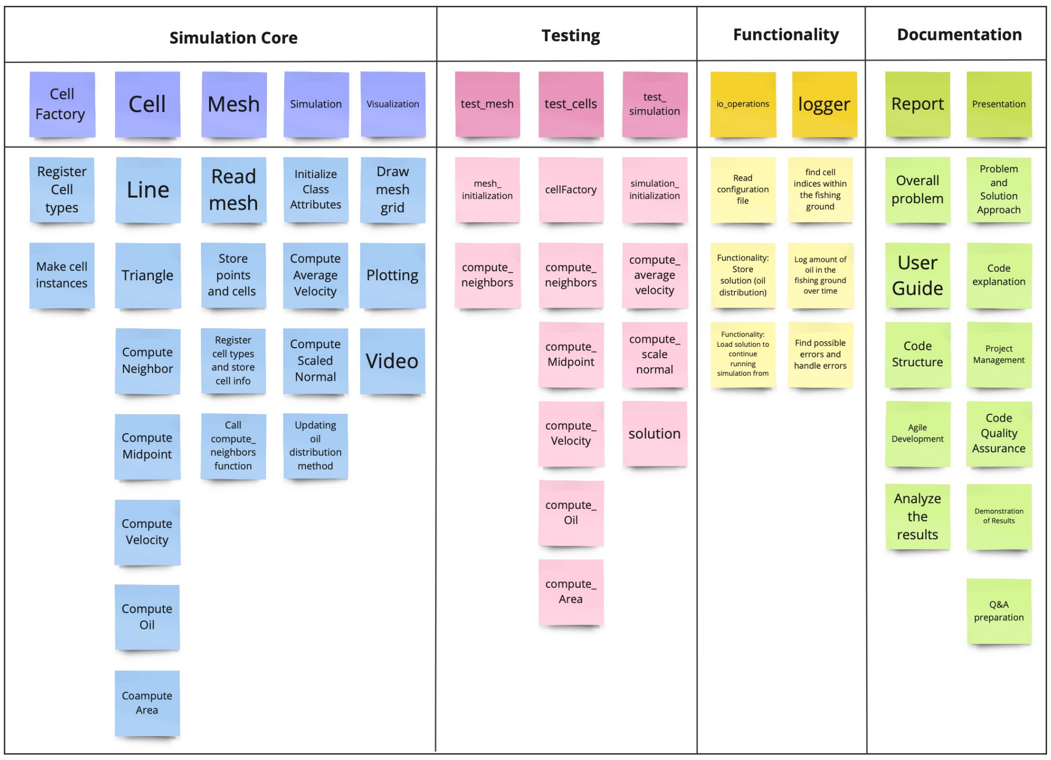
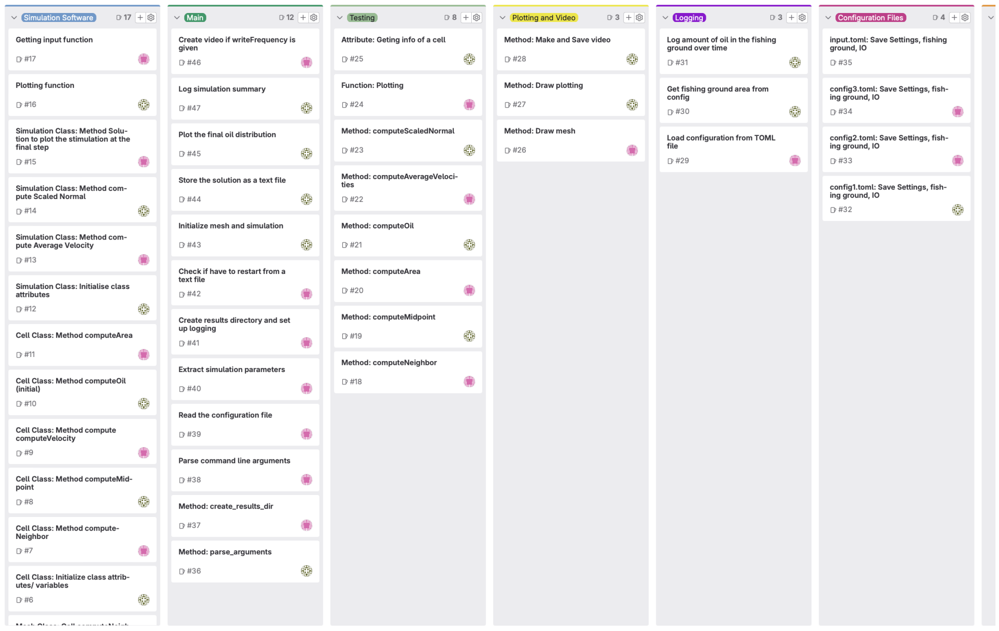


Figure 4.1: *Story mapping on our project of Simulation of Oil Spill on Computation Mesh.*

4.2 Team Collaboration

Initially, we allowed ourselves a period of exploration and understanding, during which we individually familiarized ourselves with the project's requirements. We had daily meetings for sharing insights gained during this phase. As the simulation framework began to take shape, we transitioned to more active, task-based development and utilized GitLab to ensure strong collaboration and efficient management of the project. Each small task was assigned as a issue with number, ensuring that once a part of the job was finished, clear commit messages were maintained. The figure 4.2 shows the issue board where managable tasks are assigned.



Simulation Software 17
- Getting input function #17
- Plotting function #16
- Simulation Class: Method Solution to plot the stimulation at the final step #15
- Simulation Class: Method compute Scaled Normal #14
- Simulation Class: Method compute Average Velocity #13
- Simulation Class: Initialise class attributes #12
- Cell Class: Method computeArea #11
- Cell Class: Method computeOil (initial) #10
- Cell Class: Method compute computeVelocity #9
- Cell Class: Method computeMidpoint #8
- Cell Class: Method compute-Neighbor #7
- Cell Class: Initialize class attributes/ variables #6

Main 12
- Create video if writeFrequency is given #46
- Log simulation summary #47
- Plot the final oil distribution #45
- Store the solution as a text file #44
- Initialize mesh and simulation #43
- Check if have to restart from a text file #42
- Create results directory and set up logging #41
- Extract simulation parameters #40
- Read the configuration file #39
- Parse command line arguments #38
- Method: create_results_dir #37
- Method: parse_arguments #36

Testing 8
- Attribute: Geting info of a cell #25
- Function: Plotting #24
- Method: computeScaledNormal #23
- Method: computeAverageVelocities #22
- Method: computeOil #21
- Method: computeArea #20
- Method: computeMidpoint #19
- Method: computeNeighbor #18

Plotting and Video 3
- Method: Make and Save video #28
- Method: Draw plotting #27
- Method: Draw mesh #26

Logging 3
- Log amount of oil in the fishing ground over time #31
- Get fishing ground area from config #30
- Load configuration from TOML file #29

Configuration Files 4
- input.toml: Save Settings, fishing ground, IO #35
- config3.toml: Save Settings, fishing ground, IO #34
- config2.toml: Save Settings, fishing ground, IO #33
- config1.toml: Save Settings, fishing ground, IO #32

Figure 4.2: *Our team project's issue board on GitLab*

Despite working on individual tasks, we consistently updating each other through the GitLab commiting code.

4.3 Testing and Quality Assurance

Testing and reassuring the quality of the code was one of the most important priorities for our team. During the exploration period, we investigated several possibilities for each function and tested them multiple times. Once the main framework was primarily built, we employed various testing strategies to improve and ensure the effectiveness of our code.

Testing Framework

We used `pytest` for its simplicity and robustness in handling various test cases, which needs to be installed from the terminal:

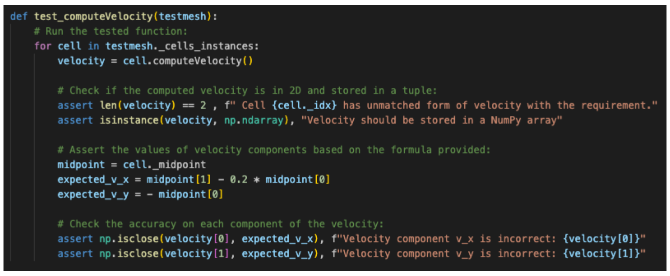
```
pip install pytest
```

Testing Folder
A folder served for testing is found in this directory:

```
Group37DinhCheon/
    tests/
        simple_mesh.msh
        test_cells.py
        test_mesh.py
        test_simulation.py
    ....
```

4.3.1 Types of Tests

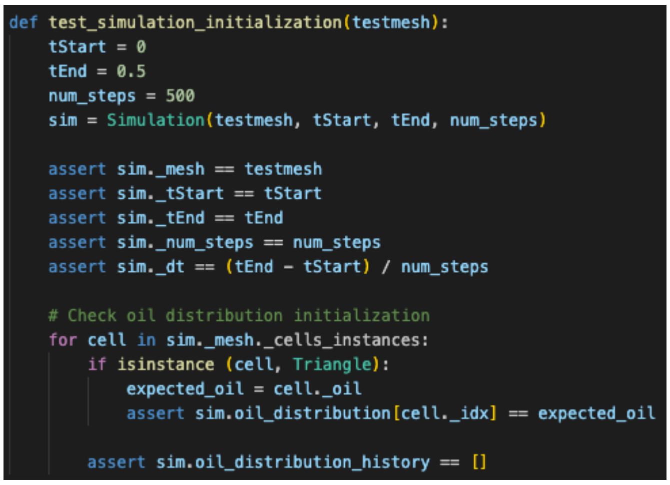
- **Unit Tests:** To verify individual functions and methods, such as `computeVelocity` and `computeOil`, to ensure they returned correct results.



```python
def test_computeVelocity(testmesh):
    # Run the tested function:
    for cell in testmesh._cells_instances:
        velocity = cell.computeVelocity()

        # Check if the computed velocity is in 2D and stored in a tuple:
        assert len(velocity) == 2 , f" Cell {cell._idx} has unmatched form of velocity with the requirement."
        assert isinstance(velocity, np.ndarray), "Velocity should be stored in a NumPy array"

        # Assert the values of velocity components based on the formula provided:
        midpoint = cell._midpoint
        expected_v_x = midpoint[1] - 0.2 * midpoint[0]
        expected_v_y = - midpoint[0]

        # Check the accuracy on each component of the velocity:
        assert np.isclose(velocity[0], expected_v_x), f"Velocity component v_x is incorrect: {velocity[0]}"
        assert np.isclose(velocity[1], expected_v_y), f"Velocity component v_y is incorrect: {velocity[1]}"
```

Figure 4.3: *Unit test on function computeVelocity()*

- **Integration Tests:** Ensured that different components, like `Mesh`, `CellFactory`, and `Simulation` worked together seamlessly.



```python
def test_simulation_initialization(testmesh):
    tStart = 0
    tEnd = 0.5
    num_steps = 500
    sim = Simulation(testmesh, tStart, tEnd, num_steps)

    assert sim._mesh == testmesh
    assert sim._tStart == tStart
    assert sim._tEnd == tEnd
    assert sim._num_steps == num_steps
    assert sim._dt == (tEnd - tStart) / num_steps

    # Check oil distribution initialization
    for cell in sim._mesh._cells_instances:
        if isinstance (cell, Triangle):
            expected_oil = cell._oil
            assert sim.oil_distribution[cell._idx] == expected_oil

        assert sim.oil_distribution_history == []
```

Figure 4.4: *Test on the initialization of class Simulation which integrates the "Mesh" class and its cell object*

- **Acceptance Tests:** Ran the complete simulation process in main.py to verify overall functionality and catch any integration issues. While most functionalities operated smoothly independently, a few fixable errors appeared during testing in main.py or the code showed weakness (time-consuming,...), highlighting inconsistencies and ineffectiveness.

5 Results

5.1 Simulation of Oil Distribution with Time

Using the plotting function, we can visualize the oil distribution at time increments of 0.1. These plots allow us to observe the direction of oil movement with the flow field and understand how the oil concentration evolves. A simulation was set up with the following parameters:

- `mesh_path`: `src/Simulation/bay.msh`
- `tStart`: 0.0
- `tEnd`: 0.6
- `num_steps`: 100

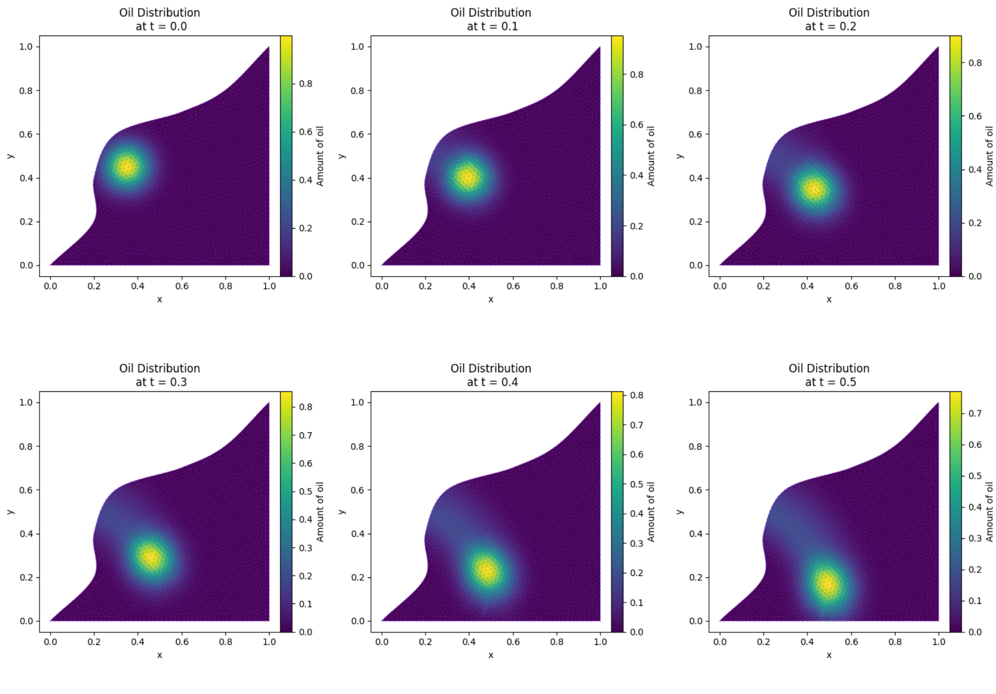


Figure 5.1: *Oil Distribution with time interval [0.0, 0.5]*

5.2 Different Simulation Scenarios

In the formula (1.4) for the flux of a cell, the flux is dependent on the time interval, Δt:

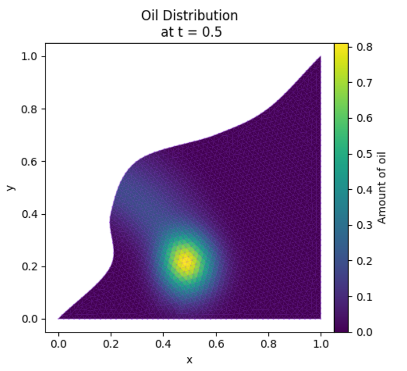
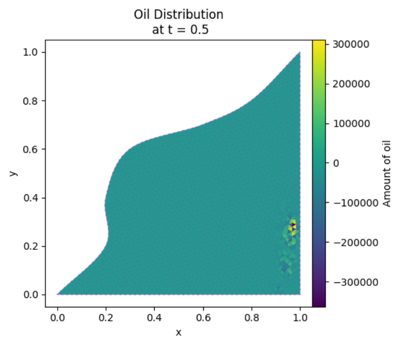
$$F_i^{(\text{ngh},n)} = -\frac{\Delta t}{A_i} g\left(u_i^n, u_{\text{ngh}}^n, \vec{\nu}_{i,\ell}, \frac{1}{2}(\vec{v}_i + \vec{v}_{\text{ngh}})\right) \tag{5.1}$$

The flux is directly proportional to Δt, meaning that if Δt increases significantly, the flux will also increase proportionally. Therefore, it is crucial to select the values of `tStart`, `tEnd`, and the number of time steps carefully since Δt is calculated by:

$$\Delta t = \frac{\text{tEnd} - \text{tStart}}{\text{num\_steps}}$$

Consequently, if the chosen time interval and number of steps result in a Δt that is significantly higher than 0.01, the oil distribution range can become highly erratic and unpredictable.

Below are the comparisons of two scenarios with their settings:

Scenario 1: Small Δt	**Scenario 2: High Δt**
• `tStart`: 0.0 • `tEnd`: 0.5 • `num_steps`: 200 	• `tStart`: 0.0 • `tEnd`: 0.5 • `num_steps`: 50 

5.3 Impact on Fishing Grounds

With the four provided configuration files, which only vary in the final time point $tEnd$, we can monitor the change in oil levels within the fishing grounds using the `log_summary()` function in `logger.py`. As we observe over time, we expect to see an increase in the amount of oil present in the fishing grounds. This is shown by reviewing the `OilFishingSummary` files in each result folder for these configurations, where each time step demonstrates a gradual rise in oil levels. To compare the logs across these configurations, we can examine both the lowest and highest recorded amounts for each setting.

Config files	**config1.toml**	**config2.toml**	**input.toml**	**config3.toml**
`tEnd`	0.7	0.6	0.5	0.4
Lowest amount of oil at $tStart$	0.10657	0.10657	0.10657	0.10657
Highest amount of oil at $tEnd$	100.38293	90.70551	66.32568	37.1776
Time step at amount of oil ≈ 1	29	35	42	54
Time point	0.1015	0.105	0.105	0.108

Table 5.1: *Comparison of logs across configuration files.*

The table confirms the expectation that the total amount of oil will increase as the time interval lengthens. Additionally, from the logs, we can predict the exact time step within the interval at which the amount of oil in the fishing grounds reaches a specific value. For instance, in the table above, we can determine the time step when the amount of oil reaches 1 unit. Hence, the exact time point in the time interval can be estimated using the formula:

$$\text{Time point} \approx \frac{Time\_step \cdot \text{tEnd}}{\text{num\_steps}}. \tag{5.2}$$

As seen in the table, across all varying settings of `tEnd`, the time point consistently shows approximately 0.105. This consistency in the time point, where the total amount of oil in the fishing grounds reaches a specific value, indicates the accuracy of the simulation system.
This is a powerful tool for farmers to take timely action accordingly. For example, if farmers believe that fish are unaffected as long as the oil in the fishing grounds stays below 10 units, this simulation system can inform them of the time available to implement a solution.

Bibliography

Kusch, Jonas (n.d.). *Task description on NMBU canvas*. Accessed: 2024-06-19. URL: https://nmbu.instructure.com/courses/10484/assignments/43787.