NORWEGIAN UNIVERSITY OF LIFE SCIENCES
(NMBU)

BAY CITY OIL SPILL SIMULATION

COURSE: **INF202**
PROJECT ASSIGNMENT IN ADVANCED PROGRAMMING

AUTHORS

ANDREAS CARELIUS BRUSTAD
HÅKON BEKKEN
JOHANNES HUSEVÅG STANDAL

NORWAY, JANUARY 2026

This page intentionally left blank.

Contents

This page intentionally left blank.

1 Introduction

Computer simulations are widely used in science and engineering to model complex systems and phenomena. They allow researchers to analyze and predict the behavior of systems under various conditions, providing insights that may be difficult or impossible to obtain through traditional experimental methods. The problem given of an oil spill has a real world implementation and is of significant importance for environmental concern. There are multiple examples like the Deepwater Horizon oil spill in 2010 Boufadel et al., 2014, where computational simulations were crucial in order to predict where surface oil would go, aiding skimming, booming, and shoreline protection.

Figure 1.1: *Deepwater Horizon oil spill (Masson, 2010)*

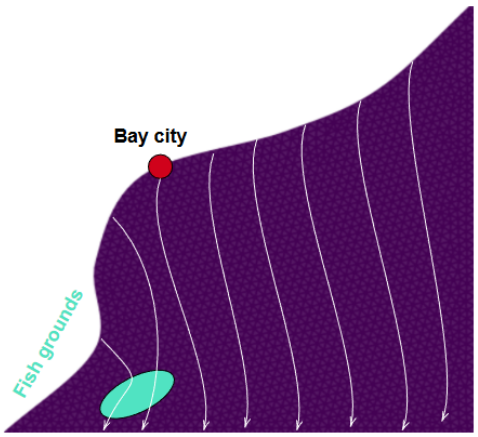

Figure 1.2: *Bay city (Kusch, 2026)*

Our simulation aims to model oil trajectory and spread forecasting in Bay city. Outside Bay city is a fishing ground that is voulnerable to oil spills. This report will discuss the mathematical models used to represent the oil spill dynamics, the numerical methods to solve these models, and the implementation of the simulation.

2

Overall problem and solution

2.1 Problem Description

The fictional fishing town of *Bay City* has reported an oil spill from one of their ships. The objective of the simulation is to assess the impact of the oil spill on the surrounding fishing grounds and to determine appropriate measures to protect the local fish population.

An external group of researchers has provided a simplified two-dimensional flow field that is used to approximate ocean currents in the region. In addition there is provided a two-dimensional map of Bay City from the file `bay.msh`. The data includes the coastline and ocean for the region.

2.2 Initialisation

When a new simulation object is created, a mesh file is loaded from the configuration file. The mesh consists of a collection of cells (triangles for the ocean and lines for the coastline). Each cell stores a scalar value representing the oil density within the area of the cell.

The initial oil distribution is centered at the spatial point

$$\vec{x}^\star = (x^\star, y^\star)^\top = (0.35, 0.45)^\top,$$

and is defined by the function

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

where $u(t, \vec{x})$ denotes the oil density at position $\vec{x}$ at time $t = 0$. The oil density is evaluated at the centerpoint of each cell and stored as the initial oil density value.

2.3 Velocity field

The vector field defines the direction and magnitude of oil flow at each cell. The simplified ocean currents are given by this formula:

$$\vec{v}(\vec{x}, t) = \begin{pmatrix} y - 0.2x \\ -x \end{pmatrix}.$$

The simulation supports a parameter for time t to account for any development in the ocean currents. This is not used in the current formula, but enables the programs extendability. If the vector field is time-independent this drastically increases the effectiveness of precomputated values. This assumption makes a much faster "faucet-optimized" simulation possible.

2.4 Oil Transfer

Let u_i^n be the oil density in cell i (area A_i) at time t^n. Each edge ℓ of cell i has a scaled outward normal $\vec{v}_{i,\ell}$ pointing toward its neighbor ngh_ℓ.
The velocity at each edge is the average of the velocities at its endpoints:

$$\vec{v}_{i,\ell}^n = \frac{1}{2}\left(\vec{v}(\vec{x}_i, t^n) + \vec{v}(\vec{x}_{\text{ngh}_\ell}, t^n)\right)$$

The flux through an edge is defined as:

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

The flux contribution from edge ℓ is:

$$F_i^{(\text{ngh}_\ell, n)} = -\frac{\Delta t}{A_i}\, g(u_i^n, u_{\text{ngh}_\ell}^n, \vec{v}_{i,\ell}, \vec{v}_{i,\ell}^n)$$

These edge contributions are first computed and stored. Then the oil density is updated for cell i as:

$$u_i^{n+1} = u_i^n + \sum_{\ell=1}^{3} F_i^{(\text{ngh}_\ell, n)}$$

Each cell exchanges oil with its neighbors through edges, using the flux function. Updates are calculated first for all cells, then applied simultaneously.

2.5 Simulation types

2.5.1 Main simulation

When the simulation starts, the solver initializes the values according to the provided functions. The mesh connects cells that share an edge as neighbors, storing a reference

to each neighbor in a dictionary within the cell. Each entry in this dictionary associates a neighbor cell with the corresponding outward-pointing scaled normal vector, which points from the main cell toward that neighbor. Given a total simulation time t_{end} and N time steps, the time step size is:

$$\Delta t = \frac{t_{\text{end}}}{N}.$$

After initialization, the solver checks whether it can run the optimized faucet simulation and produce the same result. If the criteria are not met, it falls back to the standard simulation.

2.5.2 Standard Simulation

For a time-dependent velocity field $\vec{v}(\vec{x}, t)$, the above fluxes are recomputed at every time step. At each timestep $t^n = n\Delta t$, fluxes are evaluated for all triangle cells, and the change in oil densities are stored in a buffer variable. After all the cells have calculated their update value, they add it to their stored oil density

2.5.3 Faucet Simulation

When the velocity field is time-independent, its time derivative vanishes, and the velocity at each cell remains constant for all times:

$$\frac{\partial \vec{v}}{\partial t} = [0, 0],$$

This means that all properties associated with a cell can be precomputed before the simulation starts using the given formulas.
The flux direction across an edge is determined by:

$$\langle \vec{v}_{I,\text{ngh}}, \vec{\nu}_{I,\text{ngh}} \rangle$$

We only consider flow out of a cell and into its neighbour. The flux direction will align with the velocity if $direction > 0$.
For each such outgoing edge, a *faucet* from cell I to cell
ngh is defined. Each faucet is characterized by two constant coefficients,

$$\text{flow}_{I\to\text{ngh}} = \frac{\Delta t}{A_I}\langle \vec{v}_{I,\text{ngh}}, \vec{\nu}_{I,\text{ngh}} \rangle, \qquad \text{flow}_{\text{ngh}\to I} = \frac{\Delta t}{A_{\text{ngh}}}\langle \vec{v}_{I,\text{ngh}}, \vec{\nu}_{I,\text{ngh}} \rangle,$$

where A the cell areas.
During a single time step, the oil density update induced by one faucet ($I \to$ ngh) is given by

$$u_I^{n+1} = u_I^n - u_I^n \,\text{flow}_{I\to\text{ngh}},$$
$$u_{\text{ngh}}^{n+1} = u_{\text{ngh}}^n + u_I^n \,\text{flow}_{\text{ngh}\to I}.$$

The total update of a cell is obtained by applying all faucets connected to it. After all faucet contributions have been accumulated, the oil densities of all cells are updated simultaneously.

2.6 Conservation

This formulation guarantees conservation of the total oil amount. For each faucet, the oil removed from the source cell is exactly redistributed to the neighboring cell, with changes scaled by the corresponding cell areas. Since conservation holds locally for every faucet, it also holds globally over the entire computational domain.

3

User guide

Follow the installation guide on GitHub (https://github.com/Suilerac/INF202—Group-18), and use the virtual environment as the interpreter.
Below is the `input.toml` file. The simulation takes in the following arguments in order to run.

3.1 Configuration file

```
[settings]
nSteps = 100 #number of time steps
tEnd = 0.5   #Specific end time (MUST BE FLOAT value)


[geometry]
meshName = "meshes/bay.msh"         #name of computational mesh
borders = [[0.0, 0.45], [0.0, 0.2]] # boarders of fishing grounds


[IO]
logName = "input.log" #name of logfile
writeFrequency = 20   #frequency of output video.
```

3.2 Command line arguments

Below is a table of command line arguments, with a description for each:

Arg1	Arg2	Desc
`-f`	`--folder`	Config file location (root if not given)
`-c`	`--config_file`	Config file name (input.toml if not given)
`--find_all`		Uses all configs in target folder

Running **`main.py`** without arguments will use **`input.toml`** in root directory as configuration file.

3.3 Simulation output

The simulation will be stored in a **newly created folder** with the specific name you gave the `.toml` file. It will output the following files:

```
-> "name_of_config_file".log # Display oil density in fishing grounds
-> "name_of_config_file".mp4 # A video showing the simulation
-> "name_of_config_file".png # The last picture generated by the simulaton
```

4

Code structre

Building a clear and consistent code structure ensures readability, maintainability, and scalability. It supports efficient collaboration and provides a solid foundation for future development.

4.1 Folder structre

```
Inf202-Group-18 /:  Root folder
├── Examples /:  Example notebooks
├── Config /:  location for toml files
├── GifsAndPictures /:  Pictures used in readme
├── meshes /:  loaction for computational meshes
├── Report /:  The report
├── src /:  Contains the source code
│   ├── Geometry
│   │   ├── geometry.py
│   │   ├── mesh.py
│   │   ├── cell.py
│   │   ├── line.py
│   │   └── triangle.py
│   ├── Simulation
│   │   ├── plotter.py
│   │   ├── simulation.py
│   │   └── solver.py
│   └── InputOutput
│       ├── commandlineParser.py
│       ├── log.py
│       └── tomlParser.py
├── temp /:  temporary location for images created while running sim
└── Tests /:  Folder for unit tests
```

The source code is split into three packages, each with their own functionalities and purposes.

The Geometry package, as the name suggests, handles data relating to the geometry provided. It has classes for the various different cell types, and a class for handling the mesh data, and by extension, centralizing the data from the cell classes.

The InputOutput package handles data relating to user input and output. In this case, the user input is config file(s) in toml format and command-line arguments. The user output is a log file. As such, this package has one class for each of those functionalities. A class to handle logging, a class to read and parse command-line arguments, and a class to parse and provide the data of toml files.

The Simulation is the most central package, and contains classes that handle the data relating to the actual simulation. It has a plotter class that handles the various aspects of visualization of data, including image and video creation. The solver class implements all the formulas needed in the simulation, putting the advanced math in one place. The simulation class brings those two together, handling the overall logic needed to make use of the tools provided in the two other classes. The simulation class also brings the other two packages together, and thus is the most complex class in our project. Because this particular class relies on everything else, writing unit tests that only relied on this class's logic was difficult. However, with the other classes having their own unit tests, the location of a potential problem can be safely assumed based on the overall context.

The separation of these packages made the program more structured and easier to work with. Because of how independent these packages are, it also increased testability, as we could quickly write isolated tests for almost everything added. The packages aren't entirely independent of course, as some utilization of each other is necessary to actually get a simulation. We managed to avoid circular dependencies, and worked hard to make each class as independent of other classes as possible, and to keep necessary dependancy as linear as possible.

4.2 UML-diagram

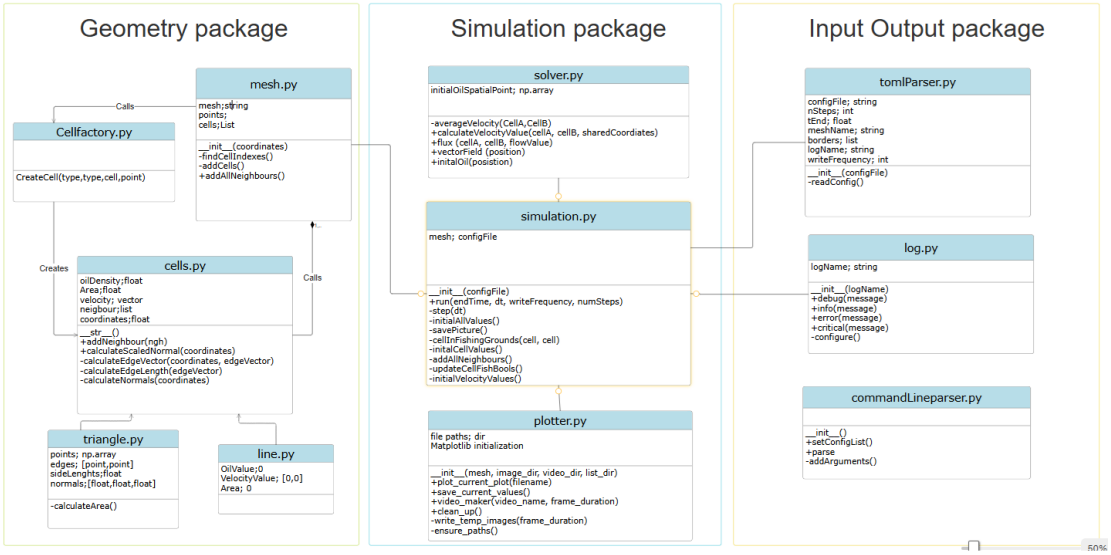

Figure 4.1: *Final UML diagram*

The UML diagram above showcases what was described in the folder structure section. Everything is centralized on simulation, with no class except for simulation being fully dependant on more than one other class. Thus, the entire project could be built iteratively with an intuitive flow through scrum concepts described below.

4.3 Quality

The entire project resulted in a highly scalable, well structured and optimized package. The process of adding a new cell type is as simple as creating a class for it that inherits Cell, and write the function for calculating area. Then add this new class to the types dict in CellFactory. All other math and related data is generalized for all convex polygons.

The structure of the code has made it intuitive to use, where a full simulation with custom parameters can be run with just a few lines of code. We have also worked hard to ensure that data handling happens where it makes sense. Geometric data handling regarding individual cells happens inside the cell class. Geometric data handling related to the mesh as a whole happens in Mesh. Simulation data handling happens in Simulation, visualization happens in Plotter, and simulation math is handled in Solver. This makes it easy to read, easy to work with, and easy to modify.

The code is well optimized, with a single simulation running in less than 5 seconds on one of our machines. This further improved our ability to rapidly iterate, as there was less time spent waiting, and more time spent fixing the things that went wrong.

4.4 Further development

No code is perfect, and there is always improvement to be done. In the Geometry package, more cell creation logic could be moved from Mesh to CellFactory, to the point of just passing the entire meshio data structure, and letting CellFactory take care of the rest. The area function for Cell can also be generalized for all polygons by splitting them into triangles from the centerpoint and summing up the triangle areas.

In the Simulation package we could improve the visualization, as currently the color of cells in the fishing grounds stays cyan regardless of how much oil enters. This can make it difficult to spot whether oil actually enters under certain conditions. In addition to these notes, there is always more to improve, and bugs or flaws we didn't catch. Overall the project follows the parameters for success defined by the task description.

5

Agile development

Agile development is an iterative approach to software creation that emphasizes flexibility and collaboration. The main idea is breaking a complex problem into smaller parts. Through small, frequent releases of working software, the strategy will produce results that get gradually closer to the end goal.

5.1 Story map and 3rd party software

Our approach to implement agile development was firstly to clarify expectations to eachother working in a group. Secondly, we spent the first days reading and understanding the problem thoroughly. By breaking the problem down to smaller pieces, we got a clear idea about what solutions the problem would require. By identifying this, we structured the work by creating a story map.

5.1.1 StoriesOnBoard

We used a 3rd party software from https://StoriesOnBoard.com with a 14 days free trial in order to facilitate agile development. We chose StoriesOnBoard because it offered more features than native github projects, had a nice layout, a user friendly GUI, and the ability to integrate with GitHub. Focusing on documentation and organizing the main structure, we achieved a foundation to start tackling the problem. Applying timelines and sprints, we also gained an idea as to when certain tasks were supposed to be done.

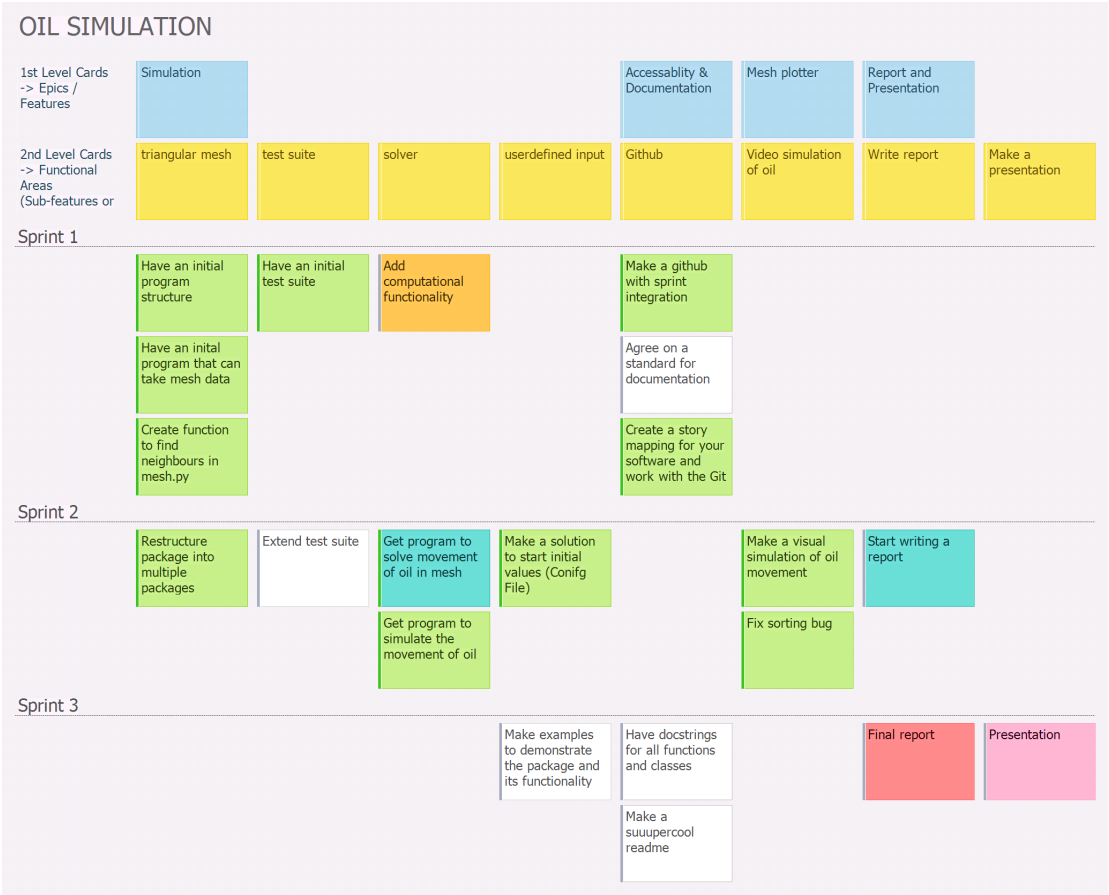

Figure 5.1: *Story map*

Our story map follows a traditional setup with Epics at the top, user stories on the second level and tasks below their respective user story. The tasks were assigned under a Sprint that was set with a duration of 5 days. They emphasize a detailed decription and checkmarks within the task, rather than a big quantity of tasks in order to have a clear overview in the Story map. If a task was missing certain elements, rather than creating new issues or sub issues, it gave a better overview to edit the description and add checkmarks.

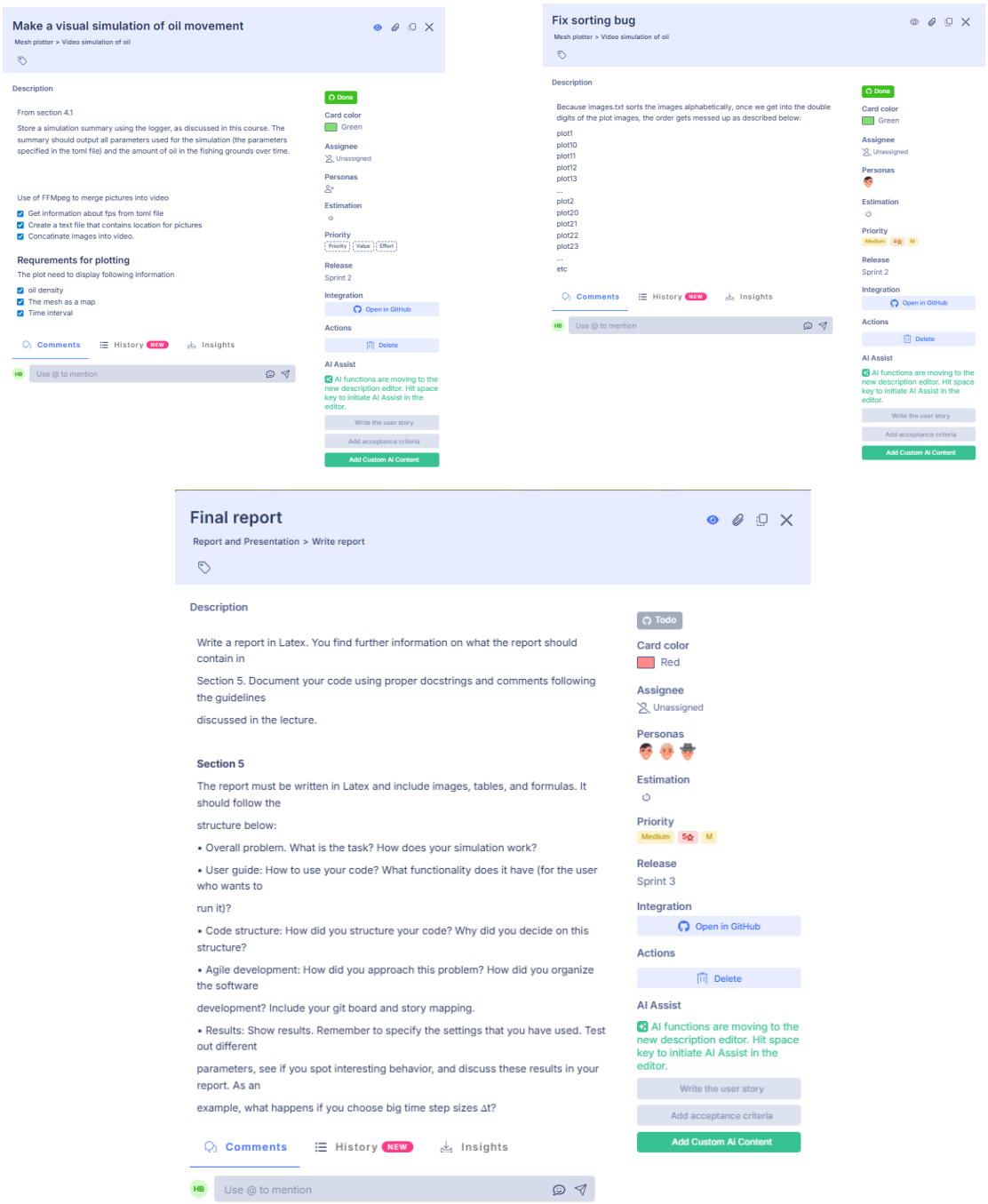

Figure 5.2: *Example of tasks*

Looking at some examples of tasks created on StoriesOnBoard, to demonstrate several features with agile development. It has instantaneous replication of GitHub issues. The possibility to stage multiple issues and bulk push them was useful. This enables us to keep a thorough overview, and it made the creation of good descriptions easier. A task could have multiple attributes for weights through priority, difficulty and effect. By having this 3rd party software, we were able to systematically and visually structure the project through agile development principles.

5.2 github

Implementing continuous integration (CI) by using tox with GitHub Actions streamlined the project testing across multiple environments. On every pull request the GitHub workflow ran our test units using pytest. During the project, we worked to keep coverage above 90% in order to catch mistakes early. The final project got well over 90% coverage as seen in Figure 5.4. The benefits with continuous integration is all about catching regressions early through automated linting, unit tests, and cross-version compatibility checks. This setup ensures reliability before merging pull requests.

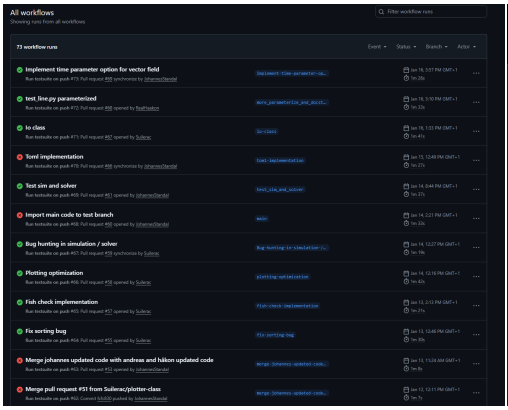

Figure 5.3: *Implementation of contionuous integration in github actions*

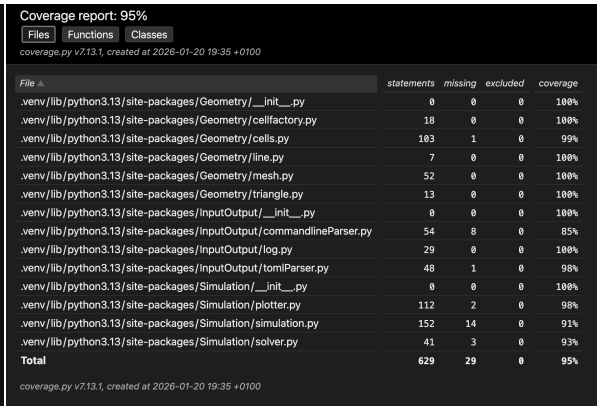

Figure 5.4: *Pytest HTML coverage report of our final project*

As demonstrated, having implemented continuous integration early on, gave significant help throughout the project, ensuring working code before merging.

6 Results

In conclusion, we got the following results with a few select simulations:

	Default	Longer duration	Fewer steps
nSteps	500	2500	5
tEnd	0.5	2.5	0.5
meshName	bay.msh	bay.msh	bay.msh
borders	[[0.0, 0.45], [0.0, 0.2]]	[[0.0, 0.45], [0.0, 0.2]]	[[0.0, 0.45], [0.0, 0.2]]
logName	log	log	log
writeFrequency	20	10	1
Oildensity in fishing grounds at end	19.33	35.84	32.13
Runtime on Macbook Air M4	2.7s	20.6s	0.6s

There were some interesting observations about the two last items. The default values ran as expected, concluding with oil entering the fishing grounds as shown:

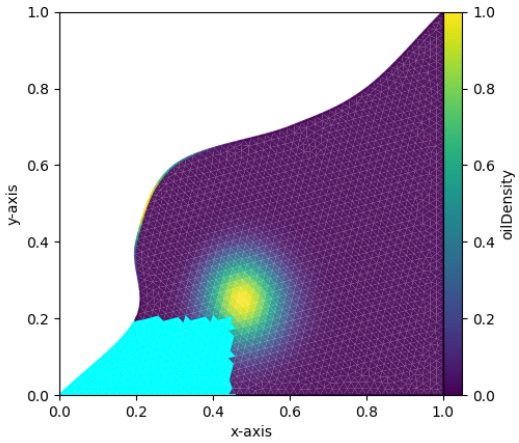

Figure 6.1: *Result of default sim*

If we let it go to $tEnd = 2.5$ however, as seen in the "Longer Duration" tab, we see that the

oil density in the fishing grounds actually stabilizes. For quite some time towards the end, the oil density stayed static at 35.84, which was also visible in the video. This is because as the oil hits the mesh borders, the flow stops, as line cells cannot give or receive oil.

The results of the "Fewer steps" simulation were also interesting. The oil density rapidly grew accross the entire plot, leaving almost the entire plot completely covered in oil by the end. This is because the dt variable grows so large that the oilDensity increases beyond what is reasonable.

This page intentionally left blank.

Bibliography

Boufadel, Michel C. et al. (2014). "Simulation of the Landfall of the Deepwater Horizon Oil on the Shorelines of the Gulf of Mexico". In: *Environmental Science & Technology* 48. DOI: 10.1021/es5012862.

Kusch, Jonas (2026). *Bay City*. Accessed: 2026-01-15. URL: https://nmbu.instructure.com/courses/13212/files/2662590?module_item_id=375271 (visited on 01/15/2026).

Masson, Petty Officer First Class John (2010). *Deepwater Horizon/BP oil spill - ATLANTIC*. Accessed: 2026-01-15. URL: https://picryl.com/media/deepwater-horizonbp-oil-spill-494e6d (visited on 01/15/2026).