NORWEGIAN UNIVERSITY OF LIFE SCIENCES
(NMBU)

OIL SPILL SIMULATION

NUMERICAL MODELING OF OIL DISPERSION IN COASTAL WATERS

AUTHORS

IFTIKHAR AMIRI
MAGNUS VASLAG LØVØY
TOM-CHRISTIAN ARMES

NORWAY, JANUARY 2025

This page intentionally left blank.

Contents

Contents i

1 Introduction 1

2 Code Structure 2

3 Agile Development 5

4 Results 8

5 Discussion & Conclusion 11

This page intentionally left blank.

1

Introduction

Computer simulations are important for tackling real-world issues in fields like engineering and environmental sciences. This report focuses on creating a numerical simulation to model an oil spill in a made-up coastal town called "Bay City." The main aim is to find out how much oil seeps into the fishing areas over time and to explore the various factors that affect its spread.

To predict how the oil moves, the simulation breaks the area down into a computational mesh. It then recalculates the flows and updates the distributions in a step-by-step manner. The simulation has features like the ability to restart from saved states, create visual outputs, and use adjustable parameters through TOML files. The findings show that changing timestep settings and the resolution of the mesh significantly impacts the accuracy of oil concentration estimates in the fishing areas. By applying basic algorithms and object-oriented design principles, this simulation provides a straightforward and cost-effective way to model oil dispersion.

2

Code Structure

The user guide for the code is available in the `README.txt` file located in the project directory for detailed instructions on usage.

The project is organized into multiple files and the key components are described below:

1. **main.py**
    - Serves as the program's entry point, handling command-line arguments and simulation setup.
    - Processes single or multiple TOML configuration files via the `TomlProcessor` class.
    - Manages logging and simulation execution, including batch processing of configuration files.
2. **src/Simulation**
    - **solver.py**:
        - Implements the `Solver` class to run simulations, calculate fluxes, and update oil concentrations.
        - Handles fishing area oil calculations, plots results, and generates simulation videos.
    - **mesh.py**:
        - Implements the `Mesh` class to manage the computational domain and identify fishing area cells.
        - Reads mesh files and initializes cells, assigning neighbors for flux calculations.
    - **cells.py**:
        - Defines the `CellFactory` for creating different cell types (`Triangle` and `Line`).
        - `Triangle` cells calculate geometric properties and simulate oil flow; `Line` cells handle boundary conditions.
3. **data/**: Stores input mesh files (e.g., `bay.msh`).

4. **tests/**
    - **test.py**: Provides unit tests to verify core functionality, including flux calculations and neighbor assignment.
5. **user_data/**
    - Contains `input.toml` and other configuration files for defining simulation settings and parameters.

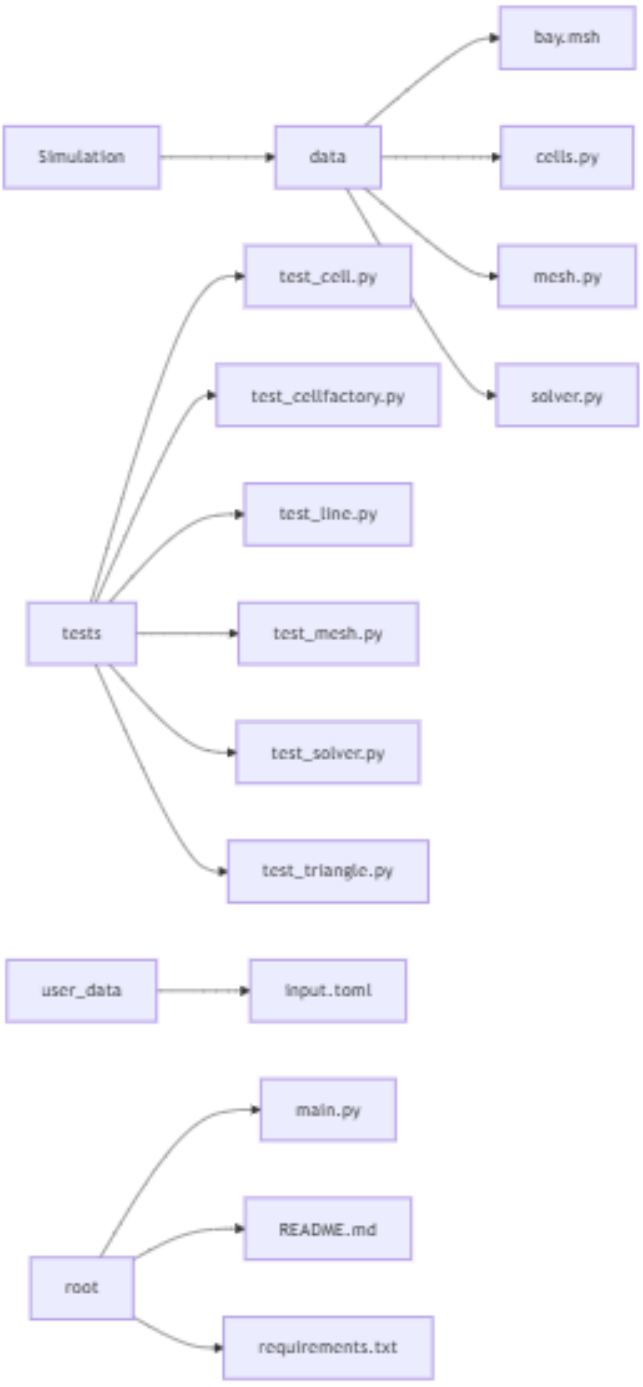

Figure 2.1: *Code Structure Diagram*

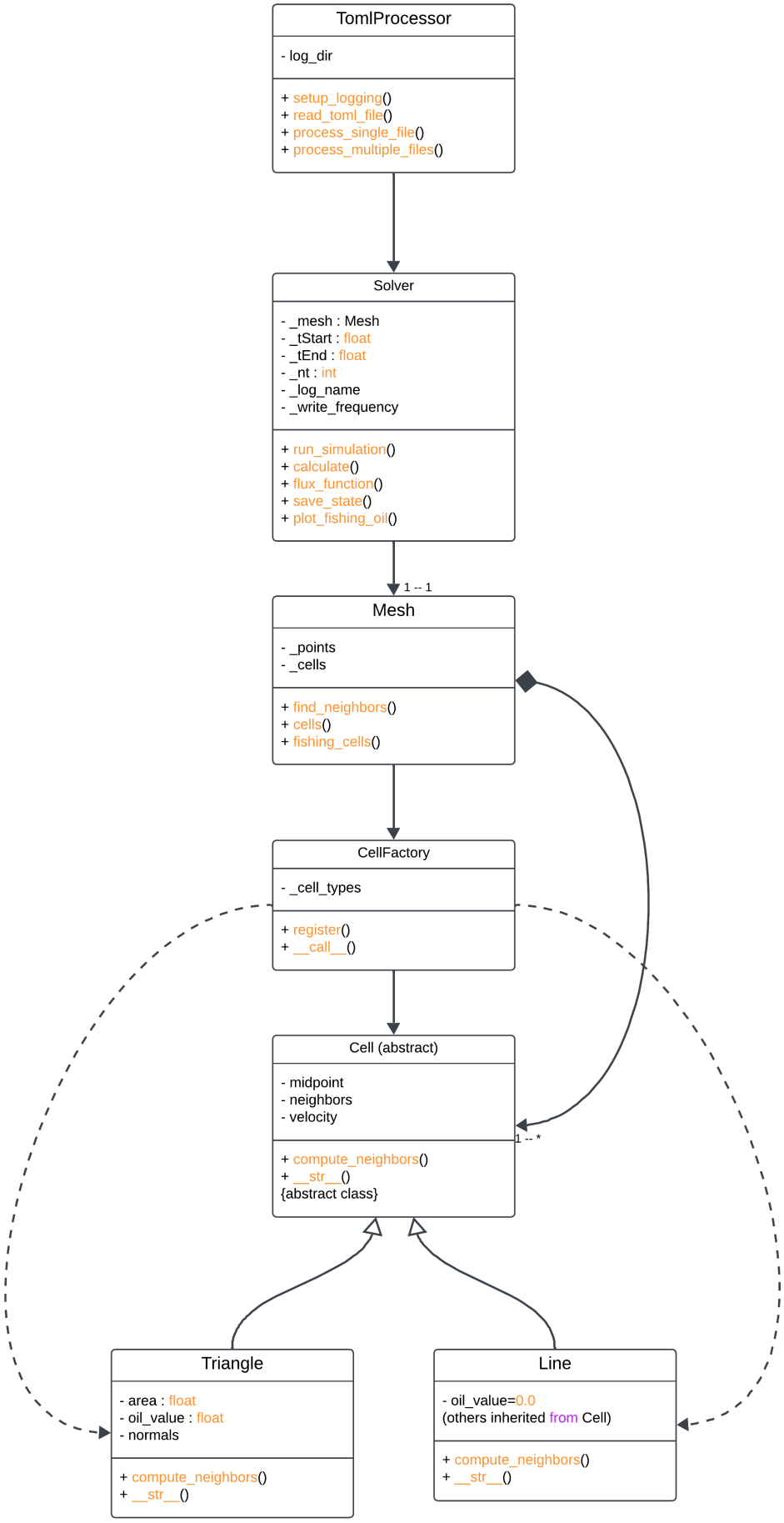

Figure 2.2: *UML Diagram of the System*

3

Agile Development

We kicked off our Agile process on day one by diving into the task, breaking down the goal of simulating an oil spill into a flowchart of smaller tasks. We then fleshed out these tasks into classes and functions, specifying the input and output variables.

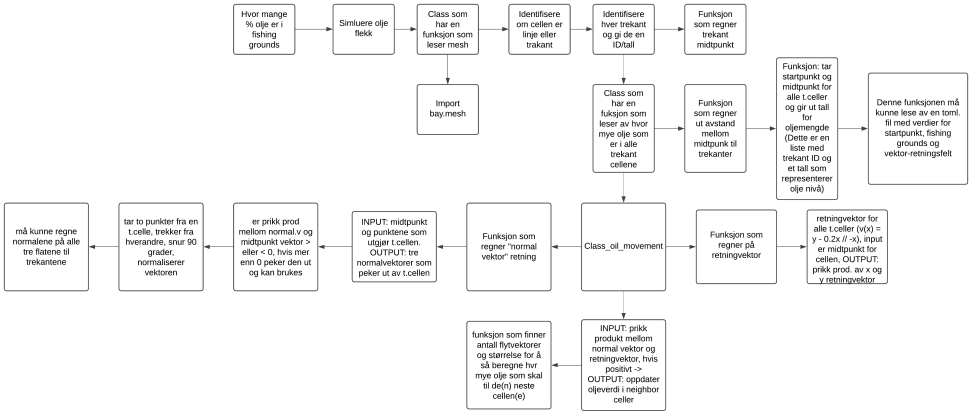

Figure 3.1: *Flowchart illustrating the breakdown of tasks and dependencies for simulating an oil spill.*

By day two, we learned Agile methodologies and transferred our tasks to a GitLab board to keep track of everything. We started with the first task and used paired programming—one person wrote the code while the others reviewed and discussed the next steps, while updating the board. This method helped improve the quality of the code and increased our team's understanding, although it did slow us down a bit.
Our first misstep was putting too much focus on user input features, like where the oil starts and the fishing areas, before we had plotted the mesh. Our professor pointed out that other teams were already running their simulations, a sign we weren't sticking to Agile principles. We shifted our focus to getting a semi-functional simulation up and running so we could debug iteratively.

By the middle of the second week, we had something that worked, but we encountered problems with the oil movement and a few negative values. After some troubleshooting, the simulation crashed altogether, and we had to roll back to a previous version. With

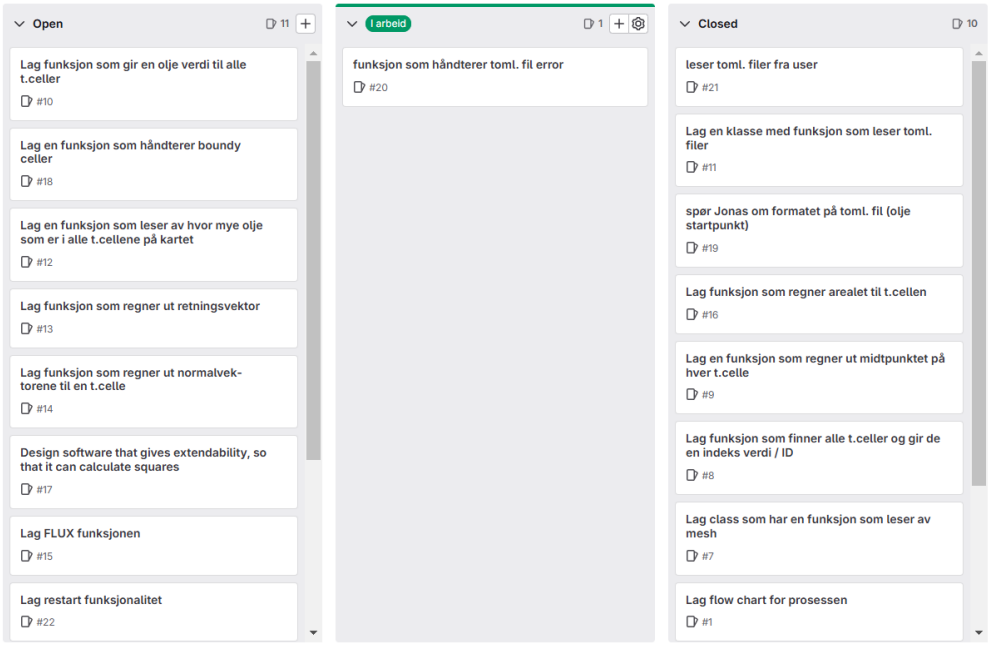

Figure 3.2: *Gitboard from Day 5 showing the progress and task tracking.*

help from the TAs, we spotted weaknesses in our code structure and reworked it according to solid object-oriented principles. Fixing these issues was not easy and took the rest of the week and weekend, but by Sunday, we finally had the simulation running smoothly.

As we got closer to our deadlines, we changed our strategy from paired programming to a more distributed approach, splitting up tasks like writing the report, implementing TOML functionality, and developing tests. This shift really boosted our efficiency, and we met up after school during the last week to wrap everything up.

This whole experience highlighted how crucial it is to stick to Agile methods and really get to grips with the code structure before jumping into development. It definitely improved both our coding skills and our overall understanding.

This page intentionally left blank.

4 Results

Our results show that running the `input.toml` file, which sets a default of 500 timesteps, the overall computation time comes out to 62 seconds. The oil starts from the initial point, travels into and through the fishing grounds, and reaches a peak of 30.82 units before it starts to decline and eventually exits the area. The graph below shows the timesteps along the x-axis, highlighting how long the oil stays in the fishing grounds.

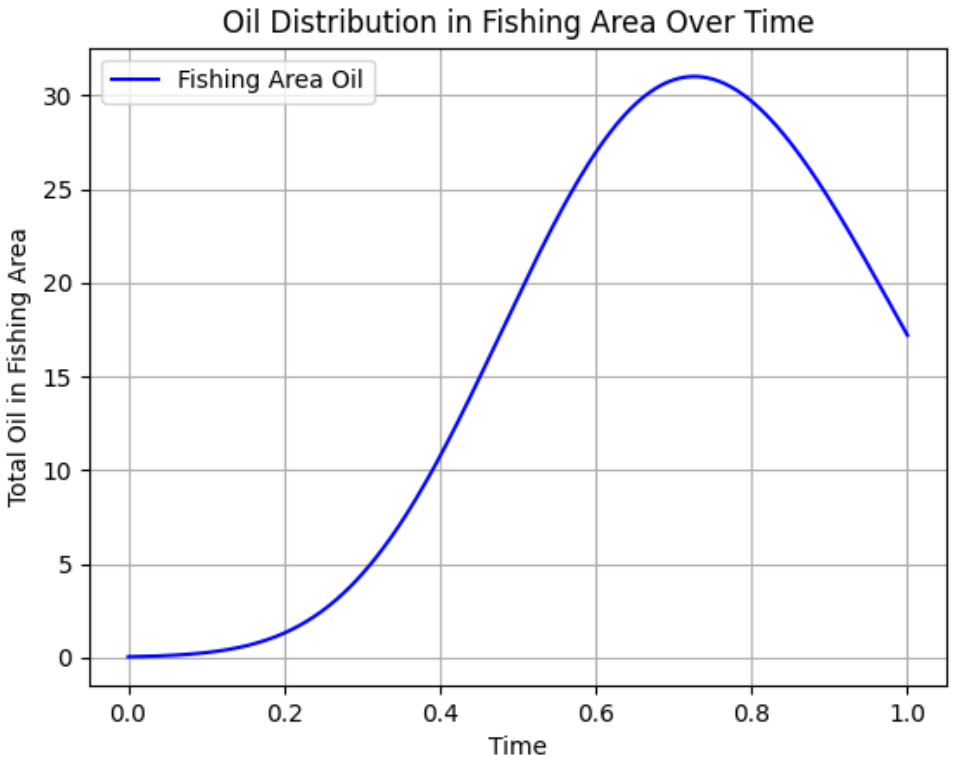

Figure 4.1: *Oil distribution in the fishing area over time*

We wanted to see how the code performed with different timestep settings. To do this, we tested the code with timesteps of 10, 50, 100, 200, 500, and 1000. Our analysis concentrated on how these changes impacted memory usage, the time it took to initialize the solver/simulation class, total computation time, and the maximum oil value in the fishing grounds. We used a logger to record all metrics, except for memory consumption, which we tracked with an external package. See summary of the performance results in Table 4.1.

Table 4.1: *Performance Metrics for Various Timesteps*

Timesteps	**10**	**50**	**100**	**200**	**500**	**1000**
Peak Memory Usage (MB)	169.7	170	168.8	169.3	169.3	169.6
Total Memory Usage (MB)	5102	5200	5370	5847	6768	8343
Solver Initialization Time (s)	13.2	13.4	12.8	13.8	13.5	13.8
Total Computation Time (s)	42	44	47	51	62	82
Maximum Oil Value in Fishing Grounds	n/a	n/a	32.4	31.4	30.82	30.65

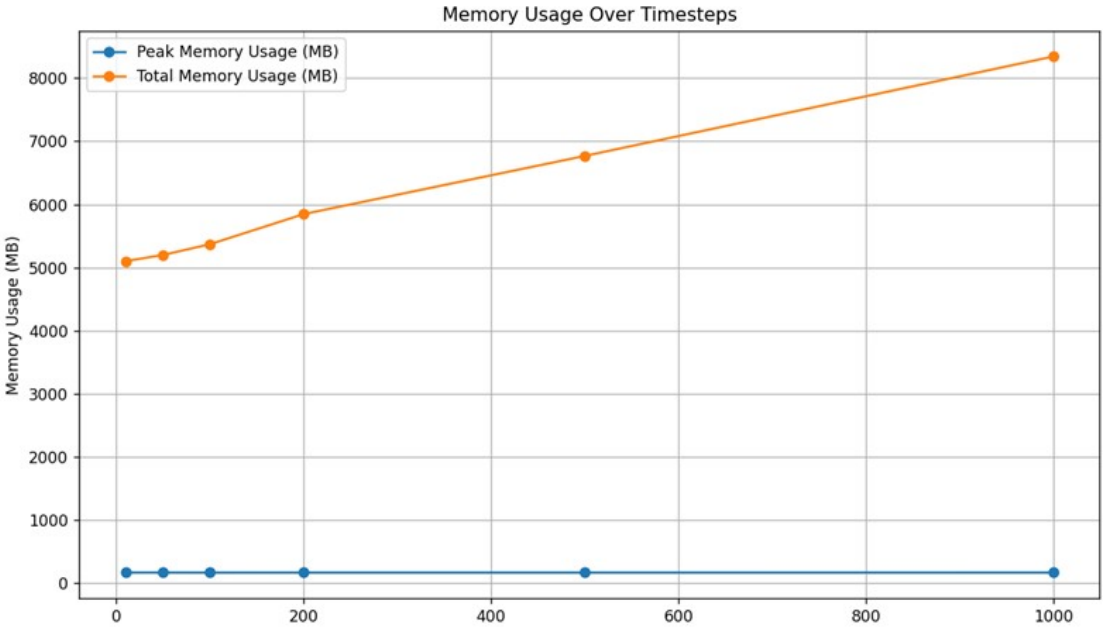

Figure 4.2: *Computation Time and Oil Values vs. Timesteps*

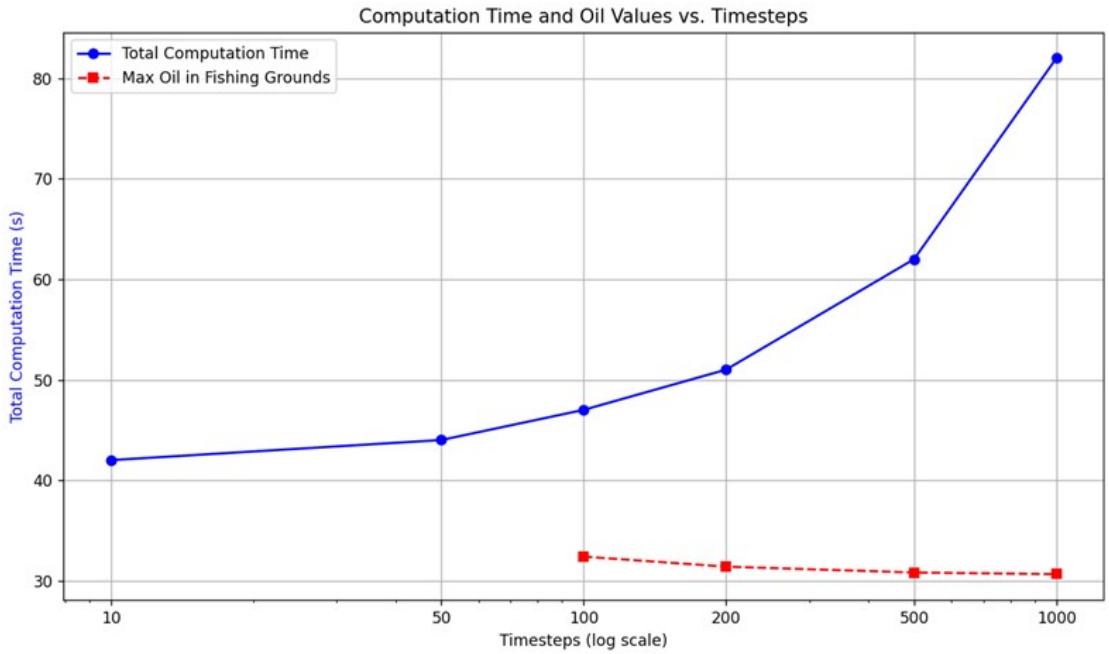

Figure 4.3: *Computation Time and Oil Values vs. Timesteps*

5

Discussion & Conclusion

The table highlights how increasing timesteps impacts simulation performance metrics. When we use higher timesteps, it allows for more frequent calculations, which enhances the modeling of oil dispersion, but it also increases the computational costs. Each timestep recalibrates the flux through the triangular mesh and updates the positions of the oil, improving detail and precision, though at the cost of more resources.

As we increase the number of timesteps, both the computational time and total memory usage go up because we need extra calculations and storage for the flux values and oil concentrations. On the other hand, the time it takes to initialize the solver stays pretty consistent at about 13.5 seconds, no matter how many timesteps we have. This is probably because the initialization happens before the iterative process kicks in. Alongside this, peak memory usage remains steady at around 170 MB since values are stored per timestep and don't need extra memory. Any small changes in these metrics are likely just random fluctuations in the system.

Interestingly, higher timestep counts lead to lower maximum oil values in fishing areas, suggesting improved accuracy. For instance, when we ran with 10,000 timesteps, the simulation levels off at an oil value of 30.47, indicating it has converged on a solution. In contrast, lower timestep counts, such as 10 or 50, result in unstable outputs because of skipped or inaccurately calculated flux interactions, which can lead to numerical errors. Using a finer mesh could enhance accuracy by better capturing the velocity fields within the cells, but this would also demand more computational resources.

Finding the right balance between mesh resolution and the number of timesteps is crucial for achieving both stability and efficiency. For example, while 10,000 timesteps give us the most accurate oil value, 100 timesteps provide a practical trade-off, yielding a value of 32.4—close to the converged solution. This highlights the ongoing balance between computational cost and accuracy, underscoring the importance of optimization for a well-rounded simulation.