Simulating an oil spill

January 3, 2025

1 Motivation

Many problems in physics, engineering, and chemistry cannot be fully explored through classical experiments. For instance, an engineer might need to determine the best design among ten different plane models, evaluating which has the least drag, the highest lift, and minimizes kerosene consumption. Building and testing all ten designs is often prohibitively expensive. Even with unlimited financial resources, it remains challenging to fully analyze an airfoil's properties through physical measurements, as sensors cannot be placed at every point on the airfoil. Consequently, the information gathered from building and testing all ten designs is inherently limited.

This is where computer simulations prove invaluable. Computer simulations recreate the physical setting within a computer, allowing experiments to be conducted numerically. As a result, no physical planes need to be built. Additionally, simulations provide precise data on the flow around the airfoil at every point in space, offering critical insights into how the best design can be further improved.

In this course, you will build such a simulation. The setting, a coastal town that wants to know if an oil spill will affect their fishing grounds, is fictional. However, the algorithm you will construct is used extensively in different simulations in academia and industry. Moreover, the techniques used to design your software and manage the project reflect the approach taken by many companies nowadays.

2 The problem

The fishing town of "Bay City" has reported an oil spill from one of their ships. They have enlisted your help to assess the impact of the spill on their fishing grounds and to determine the necessary measures to protect the fish population. Meanwhile, another group of researchers is modeling oceanic flows. To test your code, they have provided a simplified flow field that approximates the main currents in this region. Additionally, you are supplied with a two-dimensional map of "Bay City," including the coastline and surrounding ocean, along with a corresponding computational mesh, "bay.msh". To determine positions on this map, we use the coordinate axes x and y, where the point $(0, 0)^\top$ is located at the lower bottom of the map. Currently, the oil distribution is centered around the spatial point $\vec{x}_\star = (x_\star, y_\star)^\top = (0.35, 0.45)^\top$ and is given by

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

where $u(t, \vec{x})$ is the amount of oil at a given position $\vec{x}$ at a time t. The fishing grounds are located in the area $[0.0, 0.45] \times [0.0, 0.2]$, that is the x-coordinate lies in the interval $[0.0, 0.45]$ and the y-coordinate lies in the interval $[0.0, 0.2]$. The movement of oil is dictated by the underlying flow field which takes the form

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

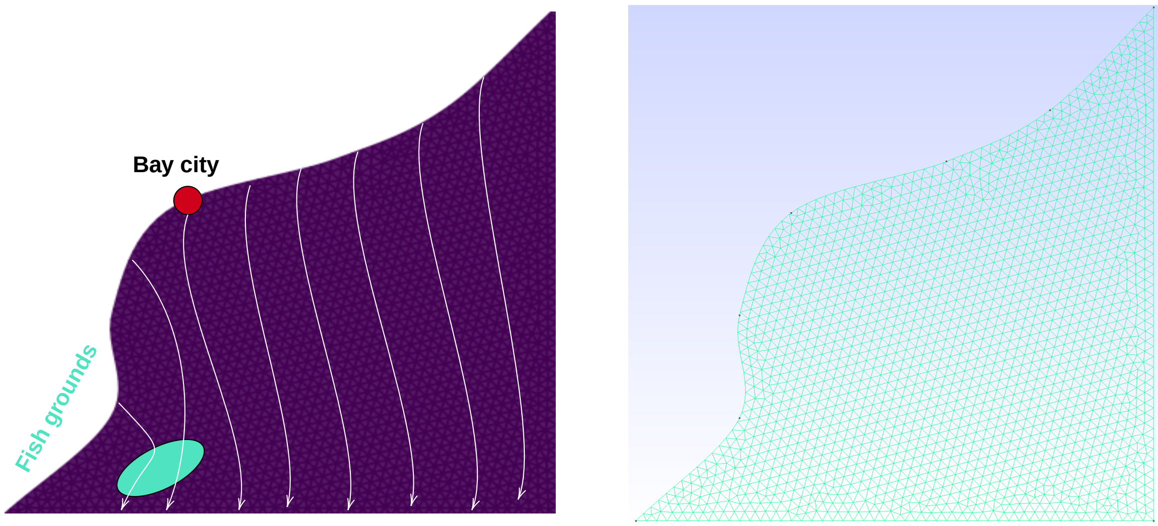

Figure 1: Left: Map of Bay City and surroundings. Right: Computational mesh of the ocean.

3 What is a numerical simulation?

Ultimately, we are interested in simulating the movement of oil along a given flow over time. Hence, we simulate the process of the known flow of the ocean $v(\vec{x})$ moving the oil in a specific direction. This occurs in a two-dimensional domain since the oil floats on the surface. Therefore, our first task is to find a way to represent the oil on the surface in a computer program.

Note that we cannot store a function $u(\vec{x})$ at every point because there are infinitely many points in $\mathbb{R}^2$ (the two-dimensional plane). So, how can we select a finite number of points to describe such a function accurately? A common approach is to divide the spatial domain into many small subdomains, known as cells. These cells are often triangles (and may include points or lines at the boundary). To represent $u(t = 0, \vec{x})$ (the oil distribution at time $t = 0$), we evaluate this function at a fixed number of points, specifically the midpoints of each cell. This provides a reasonably accurate representation of the oil distribution at time 0; see the left side of Figure 2.

Now, the question arises: How can we simulate the evolution of this oil distribution over time? The oil follows the flow field $v(\vec{x})$. Thus, over a small time interval, the oil will not jump from one cell to a random distant cell. Instead, it will move only into neighboring cells—that is, cells that share at least two points with the current cell. To ensure that the oil only moves into neighboring cells, we divide our time interval $[0, t_{\text{end}}]$ into N smaller subintervals $[t_n, t_{n+1}]$, where $t_n = n \cdot \Delta t$ and $n = 0, \cdots, N-1$. Here, Δt is chosen such that $t_N = t_{\text{end}}$. The solution is then updated over these small time intervals, during which the oil moves only into neighboring cells. To define how this evolution into neighboring cells is computed, we first need to better understand the properties associated with a cell.

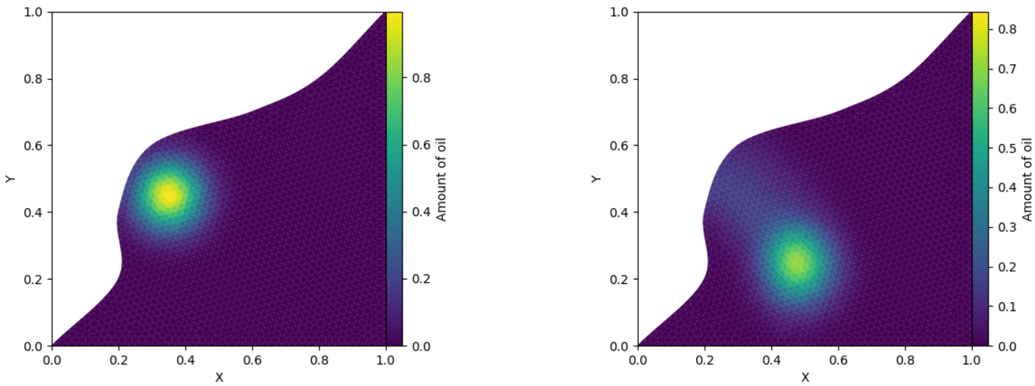

Figure 2: Left: Oil distribution at $t = 0$. Right: Possible oil distribution at $t = 0.5$.

3.1 Cells

Cells can be either triangles or lines. A line cell is used solely to specify how the simulation should interact with the boundary of the computational domain. Only the cells that constitute this boundary are classified as line cells; lines within the computational domain that do not lie on the boundary are not considered cells. A line cell consists of two points, and the amount of oil in a line cell is always zero, remaining constant over time.

A triangle cell (see Figure 3) consists of three points $\vec{p}$, which are connected by three facets (also called edges or cell boundaries) $\vec{e}$. Another important variable is the midpoint $\vec{x}\text{mid} = \frac{1}{3}(\vec{p}1 + \vec{p}2 + \vec{p}3)$. Additionally, the outward-pointing normal vectors $\vec{n}$ have unit length, i.e., $\|\vec{n}\| = 1$. These vectors are orthonormal to their corresponding facets (that is, $\langle \vec{e}\ell, \vec{n}\ell \rangle = 0$), and they point away from the cell center. To ensure that the normal vectors point outward, the angle between $\vec{p}\ell - \vec{x}\text{mid}$ and $\vec{n}_\ell$ must be less than 90 degrees.

Another key quantity is the scaled normal $\vec{\nu}\ell = \vec{n}\ell \cdot \|e_\ell\|$, where $\|e_\ell\|$ is the length of edge e_ℓ. Denoting the point $\vec{p}\ell = (x\ell, y_\ell)^\top$, the area of a triangle cell is given by $A = 0.5 \cdot |(x_1 - x_3)(y_2 - y_1) - (x_1 - x_2)(y_3 - y_1)|$.

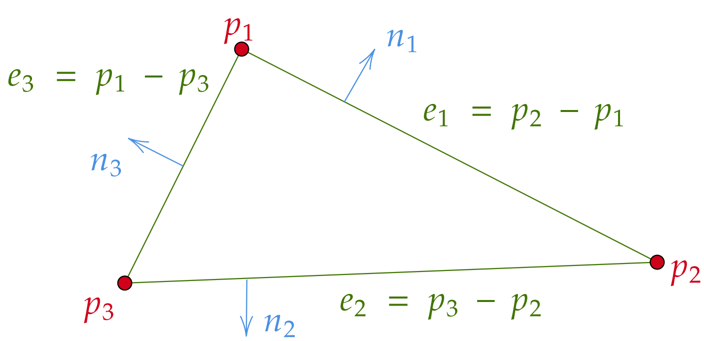

Figure 3: Triangle cell with points $\vec{p}$, facets $\vec{e}$, and outward pointing normal vectors $\vec{n}$.

3.2 Fluxes over edges

To determine how oil in this triangle is exchanged with its neighbors, let us consider a triangle cell with index i and a neighboring cell at facet e_ℓ with index `ngh`. In the following, we denote A_i as the area of cell i, $\nu_{i,\ell}$ as the scaled normal of cell i at edge e_ℓ, the velocity field at the midpoint of cell i as $\vec{v}i$, and the velocity at the midpoint of cell `ngh` as $\vec{v}\texttt{ngh}$. If the amount of oil in cell i at time t_n is denoted by u_i^n, and the amount of oil in cell `ngh` at time t_n is denoted by $u_{\texttt{ngh}}^n$, the amount of oil in cell i changes over the facet e_ℓ by an amount of

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

where g is given as

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

Then, if the three neighbors of cell i are given as $\mathtt{ngh}_\ell$ at edge ℓ for $\ell \in \{1, 2, 3\}$, the amount of oil at time t_{n+1} is given as

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

For example, assume that cell $i = 2$ has neighbors $3, 17, 6$. Then, to compute the oil amount at time t_5, we use the formula

$$u_2^5 = u_2^4 + F_2^{(3,4)} + F_2^{(17,4)} + F_2^{(6,4)}.$$

Note that the amount of oil will only change over time in triangle cells. Other cells remain constant. Also, note that neighboring cells can be line cells.

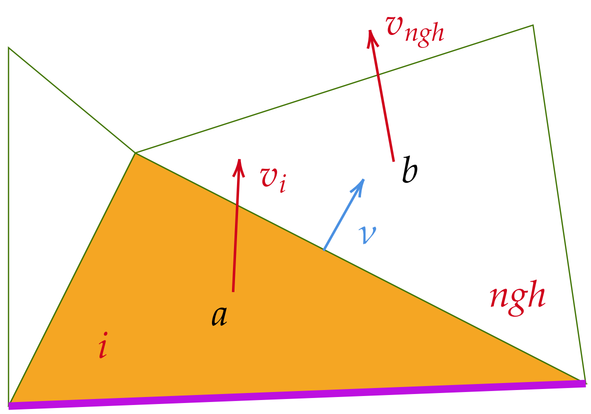

Figure 4: Illustration of the flux on a given edge with neighbor `ngh`. The amount of oil in cell i is denoted as a, and the amount of oil in the neighboring cell is denoted by b. Note that the length of ν is not depicted correctly for ease of presentation, and certain indices are left out. Also, cell i has a line cell (the purple line on the bottom) as a neighbor.

4 The task

You and your team are tasked with writing software to simulate the oil spill at Bay City using the approach outlined in this document. The software should be user-friendly and easily extendable. More detailed information is provided in the following section. Since this information includes concepts you will learn throughout the course, do not be discouraged if some points are unclear on the first day.

The requirements are categorized into *delivery* (the hard requirements your code must fulfill), *quality* (how these requirements are implemented, structured, tested, etc.), and *documentation* (how these requirements are recorded and explained).

4.1 Delivery

- Develop Python software that simulates the movement of the oil spill for the provided geometry and velocity field. Use the mechanisms described in this course to achieve this.
- Provide an option to store the solution as a text file and to restart the simulation at a specified time using the stored solution file. That is, the user can run the simulation to a time $t_\star$ and then continue this simulation to a later time t_{end}.
- Provide functionality to generate a plot of the oil distribution at the final time and create a video of the oil distribution over time.
- Provide the ability to read `toml` files that specify simulation parameters. This config file must have the structure and functionality described in Figure 5. Return an error when the specified `toml` file does not exist or has inconsistent/missing entries. The program should record a video when provided with the optional parameter `writeFrequency`. The program should restart the simulation with the solution values provided in a restart file when the parameter `restartFile` is provided. If the restart file is provided, a start time must be provided and vice-versa. If no start time is provided, the program automatically chooses time $t = 0$ as the start time. If no `logName` is provided, the program picks the name "logfile". The user has to provide the remaining options.
- Enable running multiple config files. The program should be able to find all config files in a given folder and run a simulation for each config file. Each simulation's result must be stored in a separate folder with the config file's name as the folder name. Ensure that this will not overwrite existing folders, which are not result folders. To tell the program to search for and then run all config files it found, the user can provide a command line argument `--find_all`. This will find all config files in the main program folder unless a specific folder is specified; see next point.
- To look for config files in a given folder `folder_name`, the user can use the command line argument `-f folder_name` or `--folder folder_name`.
- Ensure that command line options allow users to specify which config file to read or whether the program should search for config files in a specified folder. If no command line option is provided, the program should default to reading the config file "input.toml". To read a config file `example.toml`, the user can use the command line options `-c example.toml` or `--config_file example.toml`
- Store a simulation summary using the logger, as discussed in this course. The summary should output all parameters used for the simulation (the parameters specified in the toml file) and the amount of oil in the fishing grounds over time.

4.2 Quality

- Choose a meaningful code structure that makes your code extendable. Make sure your code is well-written and readable.
- Be efficient and extendable.
  1. Use object-oriented programming to structure your code: Use the object-oriented programming paradigms discussed in this course to facilitate extendability. Enforce data encapsulation when possible. To check the extendability of your software, for example, ask yourself how easy it would be to add a new cell type, like a quadrangle, to your code.
  2. Be efficient by avoiding unneeded operations and memory. Avoid computations that are not required for this task. Make sure that the same information is not computed or stored multiple times. Avoid multiple `if` statements when possible.

  Extendability and data encapsulation are the main priority. When you have to make a software design choice that impacts data encapsulation, extendability, and performance, choose the option that improves extendability over data encapsulation and data encapsulation over performance.
- Use GitLab to work collaboratively. You will have to submit a git log with your code in which you document your use of git. See also the section `Documentation`.

```
# Example configuration file
[settings]
nSteps = 500 # number of time steps
tStart = 0.1 # start time
tEnd = 0.2 # end time

[geometry]
meshName = "bay.msh"
borders = [ [0.0, 0.45], [0.0, 0.2]] # define where fish are located

[IO]
logName = "log" # name of the log file created
writeFrequency = 10 # Frequency of output video. If not provided, no video is recorded.
restartFile = "input/solution.txt" # Restart file must be provided if start time is provided.
```

Figure 5: Config file structure.

- Provide suitable tests using the concepts and tools discussed in this course. Make sure tests are provided for over 85 percent of your functions. Implement tests for the different test types discussed in the lecture, and ensure you follow the design concepts for tests as discussed.
- Create a package of your software and use the folder layout discussed in this course.
- Catch errors if these occur and provide meaningful error messages.

4.3 Documentation

- Write a report in Latex. You find further information on what the report should contain in Section 5. Document your code using proper docstrings and comments following the guidelines discussed in the lecture.
- Implement and document software development strategies: Create a story mapping for your software and work with the GitLab board. Document the usage of these in your report.

5 The report

The report must be written in Latex and include images, tables, and formulas. It should follow the structure below:

- Overall problem. What is the task? How does your simulation work?
- User guide: How to use your code? What functionality does it have (for the user who wants to run it)?
- Code structure: How did you structure your code? Why did you decide on this structure?
- Agile development: How did you approach this problem? How did you organize the software development? Include your git board and story mapping.
- Results: Show results. Remember to specify the settings that you have used. Test out different parameters, see if you spot interesting behavior, and discuss these results in your report. As an example, what happens if you choose big time step sizes Δt?

The length will depend on the number of tables and figures. It should have three pages of plain text (excluding images/tables/formulas).

6 The presentation and discussion

The presentation and discussion will occur on January 27 or January 28. If you have hard restrictions (for example, another exam on the same day) and can therefore only make it on one of the two days, please write a mail. All three students in a group are examined together. The exam starts with a 5-minute presentation, followed by a 25-minute discussion. The presentation needs to be submitted on

Canvas by January 25, 11:59 p.m. During the exam, the examiners will have either the presentation slides or your code open, and you can tell them if they should switch between the two. The exam is conducted in English, and we cannot provide help in other languages (including Norwegian).

6.1 Presentation

The presentation should have the following structure and flow:

- You are a software developer presenting your product to your client
- Explain the problem and the solution approach
  - Computer simulations are relevant.
  - The approach you chose is promising.
  - The results are good (if they are).
- Explain to the client/examiner how you have solved the task
  - How did you manage the project?
  - Overall structure of the code
  - Examples of how you solved specific aspects/problems
- Persuade the client/examiner that your code is trustworthy
  - How did you ensure quality?
  - Is the code in maintainable shape (tests/documentation, etc)
- Persuade the client that your code is productive
  - Ease of use. Is it easy to use and extend the code you have written?
- Show some interesting results. Make sure the examiners understand the settings you have used.

Recall that the presentation is limited to five minutes. Therefore, double-check what is important and what is not.

6.2 Discussion

In the discussion parts, the examiners ask students questions about their code and course content. Commonly, the discussion starts with students being asked to present and explain parts of their code. If the examiners have detected a weakness in the students' code (mistake, bad structure, inefficiency, etc.), they will usually see if the students are able to spot this weakness and correct it. This is often followed by general questions about the course content. If students cannot answer directly, the examiners can try to provide hints. This is to your benefit and means the examiner will not directly mark the question as failed but will see if the student has some knowledge about this question. It is important to underline that the examiners pick the questions. That is, the students cannot pick the questions they want to be asked or exclude questions they do not want to be asked.

To prepare for the discussion, talk to your partner and ask each other questions about the code and the course content. For example, ask: "Where does your code perform this functionality? Can you explain this part of the code? When the code calls this function, where does it jump to?".

7 Evaluation and grading

The evaluation of the code factors into the total grade by 70 percent, whereas the exam (presentation and discussion) factors in by 30 percent.

- 70 percent code (the group gets the same grade if everyone contributed equally)

  - Delivery (30 percent): Does your code deliver on all requirements described in the delivery description?
  - Quality (30 percent): Does your code deliver on all requirements described in the quality description? Do you choose a good code structure? Is it well-written, extendable, readable, and efficient? Do you follow object-oriented paradigms? Are there any bugs? Have you tested your code sufficiently? Are tests meaningful and well-written?
  - Documentation (10 percent): Does your code deliver on all requirements described in the documentation description? Do you have meaningful docstrings for all classes, methods, and modules? How is the report written? Do you use meaningful formulas and images? Do you include tables? How has the agile development been documented and implemented?
- 30 percent exam (individual grade)
  - Presentation (10 percent)
  - Discussion (20 percent)

The percentages factor into the final grade in the following way:

- Lower end of E is 40 percent.
- Thus, e.g., for Delivery, a "just passing" code will give 40 of 30 percent, i.e., 12 percent to the total, while a perfect delivery would give 30 percent to the total.
- Grades explained here: `https://www.uhr.no/_f/p1/i4bfb251a-5e7c-4e34-916b-85478c61a800/karaktersystemet_generelle_kvalitative_beskrivelser.pdf`

8 Delivering your code and presentation slides

- Your code must be delivered on Canvas by January 22, 2 p.m. Your presentation must be delivered on Canvas by January 25, 11:59 p.m., as a PDF file.
- Only one submission is allowed per group. Only one of you submits the code. Send your partner a screenshot of the submission notification.
- Hand in a zip folder with your code and documentation. Ensure this folder only includes subdirectories, python files, toml files, a requirements.txt, your git log, your report as a pdf, and the bay.msh file. Remove virtual environments, git folders, etc.
- If you are group X and your last names are LastName1 and LastName2, name your folder GroupXLastName1LastName2. The same holds for your presentation. That is, use the name GroupXLastName1LastName2.pdf.
- Ensure your code runs and all required packages are documented in the requirements.txt file. Do not include packages that you do not use.