NORWEGIAN UNIVERSITY OF LIFE SCIENCES
(NMBU)

REPORT: SIMULATION OF AN OIL SPILL

A FINITE VOLUME SIMULATION OF
OIL TRANSPORT IN COASTAL WATER

AUTHORS

OSCAR WIERSDALEN THUNOLD
JONAS OKKENHAUG BRATLAND
TOBIAS GALTELAND WÅLE

INF202

NORWAY, JANUARY 2026

Contents

1 Introduction

1.1 Task introduction

Computer simulations play a central role in modern engineering, as many of the problems we face require immense amounts of time and resources if they were to be tested physically. By recreating a physical setting within a computer, we can use numerical simulations to obtain working solutions to real life problems.

In this project, we are assigned the task of simulating the transport of an oil spill in a fictional coastal environment named Bay City. The goal of the simulation is to predict how the oil is distributed over time and to assess the impact on the fishing grounds located nearby.

The domain is located on a two-dimensional mesh, consisting of triangular cells. The mesh also include line cells which determines the boundary of the mesh. The amount of oil is stored cell-wise in various quantities, where as the midpoint of each cell evaluate oil amounts. The movement of oil is dictated by an underlying flow field and is simulated using an explicit time-stepping scheme. During each time step, oil is distributed to neighboring cells through fluxes that follow both the flow field and the structure of the mesh.

In this report, we document and explain our program and code structure, the formulas and physics running the simulation. Our devolopment and planning are documented to give insight in our workflow and development cycle. We show our results from tests both during the development process, and our final simulation results. Finally, the report also includes instructions on how to download, configure and run the simulation, so that you can test it out yourself.

1.2 Formulas

1.2.1 Initial condition

Defines the initial distribution at t = 0. The concentration of oil is located at a prescribed point; which represents the location of the spill. When t = 0, this distribution serves as the starting point for the simulation and is used to initialize the oil values sorted in the computational cells.

$$u(0, \mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}^*\|^2}{\sigma^2}\right), \tag{1.1}$$

1.2.2 Velocity field

The underlying flow field vector which determines the movement of oil is given:

$$v(\vec{x}) = \begin{pmatrix} y - 0.2x \\ -x \end{pmatrix}. \tag{1.2}$$

1.2.3 The flux function

Determines the amount of oil transported between neighboring cells. The flux depends on the dot product between the velocity vector and the outward normal of the edge. If the flow is directed out of the current cell, the oil value of that cell is used; otherwise the oil value of the neighboring cell is used. This ensures that oil is transported in the right direction.

$$\text{FLUX}(a, b, \boldsymbol{\nu}, \mathbf{v}) = \begin{cases} a\,(\mathbf{v} \cdot \boldsymbol{\nu}), & \text{if } \mathbf{v} \cdot \boldsymbol{\nu} > 0, \\ b\,(\mathbf{v} \cdot \boldsymbol{\nu}), & \text{otherwise.} \end{cases} \tag{1.3}$$

1.2.4 Time stepping

During a single time step, the contribution from the neighboring cells are calculated by summing the fluxes across shared edges. This contribution scales with the size of the time step. Only triangle cells are updated, while boundaries remain fixed.

$$u_i^{n+1} = u_i^n + F_i^{(\text{ngh}_1, n)} + F_i^{(\text{ngh}_2, n)} + F_i^{(\text{ngh}_3, n)}. \tag{1.4}$$

2

User Guide

This chapter describes how to install, configure, and run the oil spill simulation software. The guide is intended for users who want to execute simulations using configuration files without modifying the source code.

2.1 Requirements

The software is implemented in Python and requires a Python 3 environment. All external dependencies are listed in the file `requirements.txt` and can be installed using `pip`:

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

2.2 Program Execution

The simulation is executed via the main module `src.main`. All runtime behavior is controlled through configuration files written in the TOML format. The program supports two execution modes: single configuration mode and batch mode.

2.2.1 Single Configuration Mode

In single configuration mode, the program runs one simulation using a specified configuration file. This is done by providing the `-c` or `-config` command line option:

```
python -m src.main -c input.toml
```

If no configuration file is specified, the program defaults to using `input.toml` located in the working directory.

After the simulation completes, the following output is generated:

- A visualization of the final oil distribution (`final.png`)
- A video showing the time evolution of the oil distribution (`video.avi`), provided that `writeFrequency` is greater than zero
- A log file containing simulation parameters and time-dependent diagnostic output

All output files are stored in a dedicated directory under `results/`, named after the configuration file.

2.2.2 Batch Mode

Batch mode allows multiple simulations to be executed sequentially using a collection of configuration files. This mode is enabled using the `-find all` command line option.

To run all configuration files in a given folder, the following command can be used:

```
python -m src.main --find all -f example/
```

This command searches the specified folder for all files with the `.toml` extension and runs a simulation for each file. Each simulation produces its own result directory to prevent overwriting existing data.

If no folder is specified, the program searches for configuration files in the current directory:

```
python -m src.main --find all
```

2.3 Configuration File Format

Simulation parameters are defined in TOML configuration files. The configuration file specifies numerical parameters, geometry information, and input/output options. Missing or inconsistent entries are detected at runtime and result in descriptive error messages.

The following parameters are supported:

- **Required parameters:**
  - `meshName`: Name of the mesh file (located in the project root directory)
  - `nSteps`: Number of time steps
  - `tEnd`: End time of the simulation
- **Optional parameters:**
  - `borders`: Spatial extent of the fishing grounds
  - `writeFrequency`: Frequency at which images are written to disk. A value of zero disables video generation.
  - `logName`: Name of the log file (default: `log.log`)

2.3.1 Example Configuration File

An example configuration file is shown below:

```
# Simulation parameters
nSteps = 250
tEnd = 0.5

# Geometry
meshName = "bay.msh"
borders = [[0.0, 0.45], [0.0, 0.2]]

# Input/Output
logName = "log"
writeFrequency = 1
```

Setting `writeFrequency` to zero disables image writing and video generation, which can significantly reduce runtime for large simulations.

2.4 Runtime Considerations

The runtime of a simulation depends on the mesh size and the number of time steps. Large meshes or small time steps may result in runtimes of several minutes. When running batch simulations, users are advised to verify configuration parameters carefully to avoid unnecessary computational cost.

3

Code Structure

This chapter describes the overall structure of the codebase and motivates the main design decisions. The primary goals guiding the structure were modularity, extendability, and clear separation of responsibilities, in line with the object-oriented and software engineering principles discussed in the course.

3.1 Overall Structure

The project is organized as a Python package with a clear separation between simulation logic, input/output handling, utilities, and tests. A high-level overview of the file structure is shown below:

```
src/
  main.py
  config.py
  plotting.py
  video.py
  simulation/
    simulation.py
    mesh/
      mesh.py
      cells.py
    physics/
      flux.py
  utils/
    logger.py
tests/
```

All core functionality is located inside the `src/` directory, while test code is isolated in the `tests/` directory. This separation ensures that the simulation code can be executed

independently of the test suite and follows standard Python packaging conventions.

3.2 Entry Point and Configuration Handling

The program entry point is `src/main.py`. This file is intentionally kept lightweight and primarily orchestrates the simulation workflow:

- parsing command line arguments,
- loading and validating configuration files,
- initializing the simulation,
- managing output and logging.

Configuration handling is encapsulated in the `Config` class in `config.py`. All user-defined parameters are read from a TOML file and validated before the simulation starts. Centralizing configuration logic in a dedicated module avoids scattering parameter checks throughout the code and makes it straightforward to extend the configuration format in the future.

3.3 Simulation Core

The numerical simulation logic is contained in the `simulation/` package. The central component is the `Simulation` class (`simulation.py`), which is responsible for advancing the solution in time using a finite volume method.

The `Simulation` class does not handle mesh construction, file input/output, or visualization directly. Instead, it operates on abstract data structures provided by other modules. This design choice enforces a clear separation between numerical algorithms and auxiliary functionality, improving readability and testability.

3.3.1 Mesh Representation

Mesh-related functionality is grouped under `simulation/mesh/`. The `Mesh` class (`mesh.py`) is responsible for reading the mesh file using `meshio`, storing point coordinates, and constructing cell objects.

Cells are implemented using a class hierarchy defined in `cells.py`. The `Cell` class serves as a concrete base class that provides shared functionality such as cell indices, point references, and neighbor storage. It also implements the neighbor detection logic used by all cell types.

Concrete cell types such as `Line` and `Triangle` inherit from `Cell` and extend it with type-specific behavior. For triangle cells, geometric properties such as area, midpoints,

edge normals, and edge-to-neighbor mappings are computed and stored within the `Triangle` class.

Although `Cell` is not an abstract class, it is intended as a base class and is not used directly to represent physical mesh elements in the simulation. This design allows shared functionality to be implemented once while retaining flexibility in how cell objects are constructed and used.

3.3.2 Geometric and Physical Quantities

Geometric properties such as triangle area, midpoints, edge normals, and edge connectivity are encapsulated within the `Triangle` class. These quantities are computed once and exposed through property methods, preventing unnecessary recomputation during the simulation loop.

The physical model is separated into the `physics/` subpackage. In particular, numerical flux calculations are implemented in `flux.py`. By isolating the flux computation from the time-stepping logic, the numerical scheme becomes easier to test independently and simpler to modify or replace.

3.4 Input, Output, and Visualization

Input/output functionality is handled outside the simulation core. Plotting routines are collected in `plotting.py`, which is responsible for visualizing the solution on the mesh and producing image files. Video generation is implemented in `video.py`, which assembles image frames into a video file when enabled.

This separation ensures that visualization and post-processing do not interfere with the numerical solver and can be disabled entirely to improve runtime performance.

Logging is implemented in `utils/logger.py`. Using a dedicated logging utility allows consistent formatting and flexible control over output destinations without polluting the simulation code with file handling logic.

3.5 Testing Structure

All tests are placed in the `tests/` directory and mirror the structure of the source code. Unit tests cover individual components such as cells, flux calculations, mesh construction, configuration parsing, and the simulation step. Integration tests ensure that these components work together correctly.

Test meshes and auxiliary data are stored in `tests/data/`, keeping test-specific resources separate from production input files. This organization improves maintainability and supports automated testing workflows.

3.6 Design Rationale

The chosen structure prioritizes extendability over minimal code size. Responsibilities are distributed across small, focused modules, reducing coupling between components. Object-oriented design is used where it improves clarity and future extensibility, particularly for mesh cells and simulation components.

Overall, the structure reflects a balance between numerical efficiency, software engineering principles, and the course requirement of producing maintainable and testable scientific software.

4

Development

In our approach to this problem, we started with thoroughly reading through the task to understand the problem and the requirements. Initially, we had to identify where to start and what to prioritize to get started with the code.

For project management, we decided to work with GitHub for this project, as the majority of the group had more experience with GitHub. We could then start working on the structure for the task defining milestones and labels in GitHub.

This structure helped to distribute tasks within the group and provided better supervision during the development period, making it easier to manage daily tasks and stay on schedule.

4.1 Development Process

In the approach to understanding the task, we divided the information into parts. We began working on the computational mesh, including how we could read the mesh file and define the different cells. Then we could proceed and implement the physics such as the vector field and flux equations before we could start plotting the results.

To develop this project in a clean and structured way, we had to work on one component at a time. Each component was individually implemented and tested before being integrated with the rest of the system. Then we could test whether the components worked together as intended or adjust the code if necessary to ensure that it worked as intended.

4.2 Strategy

Our strategy when starting to work on the task was largely based on a partial scrum approach as taught in the lectures. The work was divided into smaller parts similar to sprints, and the work sessions were started with a discussion of current status, including

what had been completed and what remained.

GitHub issues were actively used to maintain control of the development so that we could avoid overlapping or colliding commits between group members. By doing so, we could maintain a clear understanding of how to distribute the work and what was left to do.

4.3 Story mapping and Git board

For structure during development, we used GitHub issues, milestones, and labels. For each day, we started working on the issues we have prepared the day before. That way we could effectively start working in the morning. All issues got a label and were assigned a milestone for good structure. The issues were initially created based on a simple story-mapping approach, where larger goals were broken down into smaller, task-oriented issues.

These figures illustrate how GitHub issues were used in the early stages of the project. Figure 4.1 shows how the issues were organized before development started, while Figure 4.2 shows the state of the issues after the workday was completed.

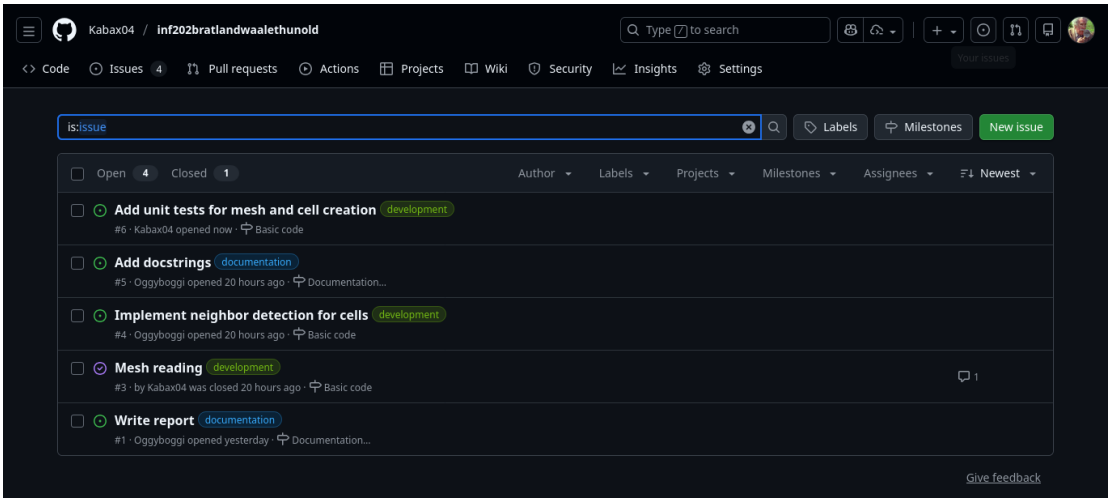

Figure 4.1: *GitHub issues overview before development on January 7*

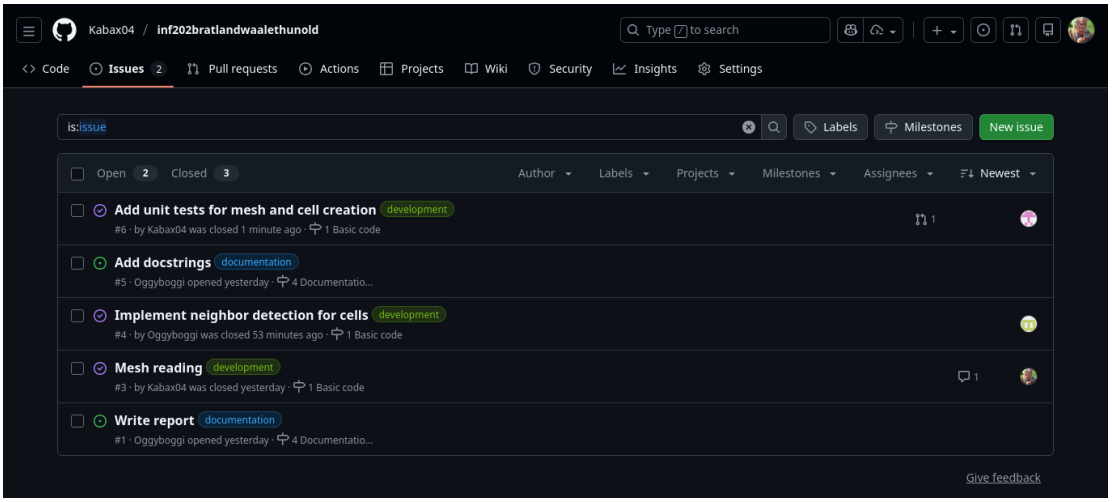

Figure 4.2: *GitHub issues overview after development on January 7*

As shown in the figures above, labels such as *development* (green) and *documentation* (blue) were used to categorize issues. In addition, the milestones to which the issues were assigned are visible, including *Basic code* and *Documentation and finalization*. Figure 4.3 shows the final state of the milestones after the coding phase, with only the report and presentation remaining.

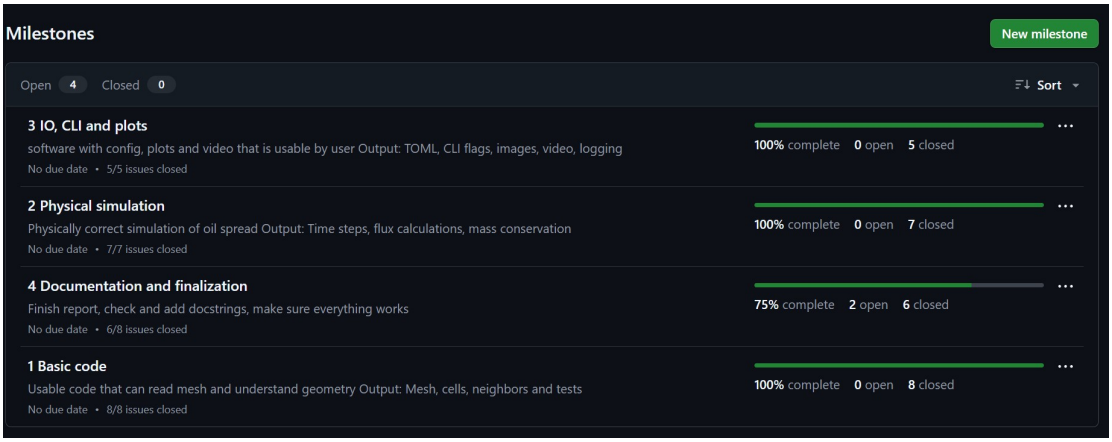

Figure 4.3: *Milestones after code is finished, only report and presentation remaining.*

This workflow helped reduce merge conflicts and ensured that group members could work on different tasks in parallel. It also helped us maintain control over the overall progress and complete tasks according to the planned schedule.

4.4 Sprint and work distribution

The development was organized into short, sprint-like periods. The sprints had different durations because of the amount of code for each sprint. At the beginning of each sprint we planned issues and distributed the tasks every day. Progress was discussed regularly each day, allowing the group to track ongoing work and adjust tasks when necessary.

This approach helped ensure efficient collaboration and prevented overlapping work during development.

Tasks were distributed based on experience and level, with the aim of balancing both task complexity and overall workload among group members.

4.5 Version control and git usage

In addition to using GitHub as our project management tool during development, it was also used for version control. A branch-based strategy was applied, where each issue was implemented in a separate branch. This ensured that the main branch would always contain stable and working code in case of problems during development.

The use of branches allowed us to develop code simultaneously. By working together and maintaining clear communication and coordinate tasks, we were able to steer clear of any merge conflicts, and new branches could be integrated smoothly.

When closing issues and branches, we used commit messages to document what was done, mainly to allow us to review previous changes in case we needed to understand what had been edited or developed during earlier stages. This helped us keep the documentation clear throughout the project.

4.6 Tests and quality

For testing the developed code, pytest was used continuously throughout the entire project. Tests were written alongside the implementation of each issue. This allowed us to fix and restructure the code at an early stage if necessary. The result of this extensive testing meant that when all components were completed, we only had minor problems before the simulation could be executed correctly.

To ensure the inputs in the TOML file (`input.toml`) are correct, we implemented the function `_validate()` in the configuration file (`config.py`). This function checks if all variables have been provided and that the values expected to be positive are positive. If the validation fails,an error message will inform the user of what is wrong.

4.7 Challenges and learning points

The main challenge in this task was gaining an overview of its extent. Establishing a sprint plan and distributing the workload over the period was essential to be able to finish in time. Once a clear structure was established, the development progressed smoothly. The group developed a clear understanding of what needed to be delivered each day to meet the deadline.

The main learning outcomes for the group include an improved ability to understand a task like this, manage the workload effectively, and collaborate on a software development project. The hands-on experiences from this collaboration will further strengthen us in future studies and in professional work environments.

4.8 Summary

In this project, a simulation of oil flow was developed using a computational mesh. We worked in sprint-like periods inspired by the Scrum framework. This allowed us to effectively develop the project in a structured way. Continuous testing throughout the entire development sprint enabled us to easily integrate all the sections of code together for the simulation.

GitHub was used extensively for management and version control. This helped us collaborate and maintain structured code throughout the entire project.

The final solution is a simulation tool that models the flow of oil on water. The solution produces consistent simulations meets the requirements specified in the assignment.

During the entirety of this project the group gained valuable experiences in collaborative software development. These experiences will strengthen us in future academic work as well as in professional software development environments.

5

Results

This chapter presents numerical results from the oil spill simulation and investigates the effect of different time step sizes. All simulations were performed on the same mesh (`bay.msh`) and with identical initial and geometric settings. Video generation was disabled to reduce runtime overhead.

5.1 Simulation Setup

All simulations were run with a fixed end time of $t_{\text{end}} = 0.5$. The time step size Δt was controlled indirectly by varying the number of time steps n_{steps}, such that

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

The amount of oil inside the fishing ground region was logged throughout the simulation, and the final value was used as a quantitative comparison metric.

5.2 Effect of Time Step Size

Table 5.1 summarizes the results for different choices of n_{steps} and corresponding time step sizes.

n_{steps}	Δt	Final oil in fishing ground
250	0.002	3.320×10^{-3}
125	0.004	3.323×10^{-3}
50	0.010	3.328×10^{-3}
25	0.020	3.277×10^{-3}
5	0.100	3.986×10^{-3}

Table 5.1: *Final amount of oil in the fishing ground for different time step sizes.*

For small time step sizes ($\Delta t = 0.002$ and $\Delta t = 0.004$), the final oil amounts are nearly identical, indicating that the solution is stable and only weakly dependent on the time

discretization in this regime. Increasing the time step to $\Delta t = 0.010$ still produces comparable results, although minor deviations can be observed.

When the time step is increased to $\Delta t = 0.020$, a noticeable deviation from the baseline solution occurs. While the simulation remains numerically stable, the reduced temporal resolution leads to a loss of accuracy.

For the largest time step ($\Delta t = 0.100$), the result deviates significantly from all other simulations. The final oil concentration in the fishing ground increases sharply, indicating non-physical transport behavior.

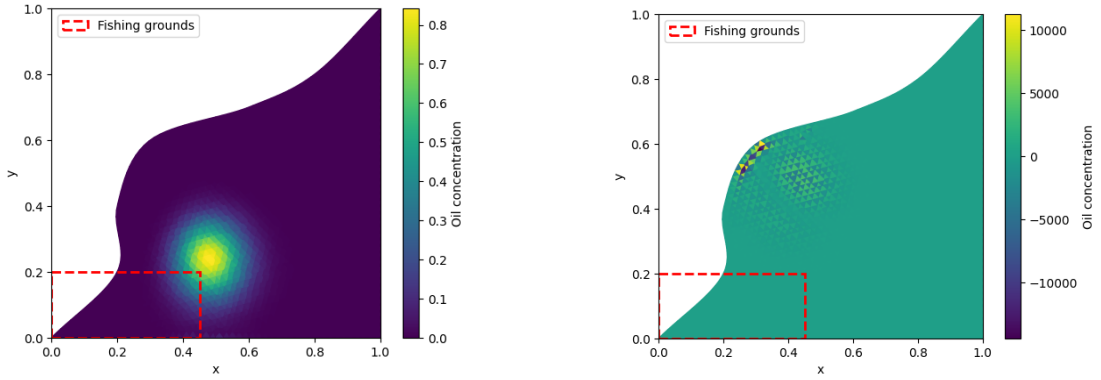

Figure 5.1: *Final oil distribution for the baseline simulation ($\Delta t = 0.002$, left) and for an extreme time step ($\Delta t = 0.100$, right).*

As shown in Figure 5.1, the extreme time step leads to a visibly different oil distribution compared to the baseline case. The spatial pattern suggests overly aggressive transport and loss of physical realism when the temporal resolution is insufficient.

5.3 Discussion

The observed behavior can be explained by stability constraints inherent to explicit finite volume schemes. In practice, the time step must satisfy a CFL-type condition, where Δt is limited by the local cell size and the magnitude of the velocity field. When this condition is violated, the numerical scheme may remain formally stable but produce inaccurate or non-physical solutions, as observed for the largest time step.

The numerical method used in this project is conservative by construction, as oil is exchanged between neighboring cells through fluxes across shared edges. Consequently, total oil mass in the domain is approximately preserved, and deviations in the fishing ground measurements are primarily due to redistribution rather than numerical loss or gain of oil.

Finally, the upwind flux scheme introduces numerical diffusion, which leads to smoothing of sharp gradients in the oil distribution. This artificial diffusion increases with the time step size and contributes to the visibly more diffuse oil fronts observed for larger

Δt. The effects seen in the results are therefore primarily numerical in nature rather than a consequence of physical diffusion.