Joined Lab! C++ meets PDEs

Fall 2025

Lab sheet #6: Wave equation

26.11.2025

This is a **joined C++ / PDEs lab**, but the grade will be solely used for the PDEs course. Therefore, the **submission** should be done via the **PDEs course Moodle website**.

- **Submission platform:** Only submissions via the Moodle page will be accepted. Submissions via Email are not allowed unless there are any technical issues at (and only at) UGA's site. You can do multiple uploads and the **last upload will be used for grading**.
- **Deadline:** The deadline for the submission is **09.12.2025**, one minute before midnight. This is a **strict** deadline!
- **Working in groups:** Regarding the programming part, you're encouraged to work in groups of up to 3 people! Please keep in mind that those of you who know the answer should **not tell the answer straight ahead**, but **help others to find the answer "on their own"**.

  Write in the first lines of your code with whom you collaborated with this code.

  Regarding the **report**, it **must be written individually**, but you can reuse the graphics. Also, please mention in the first lines of your report with whom you collaborated on the programming part. Also, the **individual assignment (last section) has to be different** for each team member.
- **Submission format:** For the report, provide a **.pdf** file (no Libreoffice/Word or any other format!).

  Provide the report as well as the source code within a single **.zip** or **.tar.bz2** file using **well structured directories** within it.
- **Quality of submission:**
  Regarding the report, the writing quality, the quality of figures (title, legends, labels), equations and the readability will be taken into account of the grading. **Think about using Latex**! The report should be self-explanatory without the source code.

  Grammar and spelling will be taken into account. There are no excuses of not using at least a spellchecker!

  Regarding the source code, also its **ease of use** and **readability** of the code will be part of the grading. It's better to have less things working correctly rather than having a lot of things "done", but not working correctly.
- **Program specific:** Regarding the source code, it must be possible to compile everything with just a single "make".

  The **maximum runtime** of the program will be limited to **5 minutes** (without plotting time). If your program takes longer, you need to optimize it!

This worksheet is related to what your future employer (either in the industry or in one of the research labs) will expect from you later on in the job. It is very likely that you will have to familiarize yourself with the developments of others and extend them. This is exactly the case for this last worksheet.

To motivate you a little bit: You will see "nice looking" animations of waves!

A submission of a **short video** of the simulations you're developing is highly welcomed!

1) Linear 1D linear wave equation

We study the 1D linear shallow-water equation (SWE) given in its first-order formulation

$$\begin{aligned} h_t &= -\overline{h} \cdot v_x \\ v_t &= -g \cdot h_x \end{aligned}$$

with the water height h, the velocity v and an average state $\overline{h}$ around which we linearized the equation. This equation can be derived from the full non-linear SWE, neglecting the non-linear terms. We assume a **periodic boundary** and a domain of size $\Omega = [0; E]$, hence

$$\begin{aligned} h(x) &= h(x + n \cdot E) \\ v(x) &= v(x + n \cdot E) \end{aligned}$$

with $n \in \mathbb{Z}$. The initial condition is given by

$$\begin{aligned} h(x) &= \exp(-(x - E/2)^2) \\ v(x) &= 0. \end{aligned}$$

This equation should be discretized with a 2nd-order accurate centered finite difference method and a suitable time-stepping method to get at least a 2nd-order accuracy overall. With periodic boundary conditions, we have in total $N + 1$ points, including the periodic boundary points. However, we should only store N points! The degrees of freedom (the discrete values of h and v) is then related to locations $x_i = i \cdot \frac{1}{N}$.

2) Implementation

2.1) First steps

A framework is provided on the course website.

First, compile the framework with

```
$ make
```

and then run it with

```
$ ./main
```

This will create a number of output text files that contain the simulation state data based on the default parameters of the simulation.

You can visualize the data by calling

```
$ ./gen_animation.py
```

which takes all these output files and does a nice animation. Note that the scale in this animation is not always right since some rescaling of the values is required for a good visualization later on.

2.2) Overview

Make yourself acquainted with the provided framework, which we will extend to be able to solve 1D wave propagation phenomena.

Advice: Start at the `main(...)` function and **work your way through** to what happens over one execution of the program.

- `main.cpp`: The place where everything comes together.
- `Config.hpp`: All configuration variables to specify which equations should be used, the parameters, and the discretization are gathered here.
- `DiscConfig.hpp`: A configuration that is specific to the spatial discretization.
- `GridData.hpp`: A storage container for the spatial values of the discretized variable (e.g., array of height values).
- `Operators.hpp`: Implementation of differential operators.
- `TimeStepperBase.hpp`: Class with time stepping interface.
- `TimeStepperRK?.hpp`: Implementation of particular time stepper

Call

```
$ ./main -h
```

and have a look into `Config.hpp` how to use program parameters to set the parameters of simulations.

Hint: Invest at least half an hour to make yourself acquainted to the framework.

2.3) Function parameter/argument namings

The code uses a particular prefixed of parameters (arguments):

- `i_` is used to denote variables only used for input (constant).
- `o_` is used to denote variables only used for output (needs to be writable).
- `io_` is used to denote variables used for input and output (needs to be writable).

E.g., if we hand over the time stamp, we should call this variable `i_timestamp`. If we hand over a reference to a class where data should be written, we use `o_someclass`.

These prefixes strongly improve the readability of the code!

3) Differential operator

3.1) Assignment: 1st order differential operator

Extend the method `Operators::diff1(...)` to compute a first-order derivative with centered differences.

Hints:

- Be careful to do it right. Otherwise, you'll have serious problems later on!
- Take care of the periodic boundaries.

4) Time integration

The entire time integration is abstracted with a `TimeStepperBase` class. This allows the implementation of different time-stepping methods and using them as long as they provide the required functionality from `TimeStepperBase`.

4.1) df/dt implementation

Extend in file `TimeStepperBase.hpp` the method `df_dt()` to compute the time derivatives.

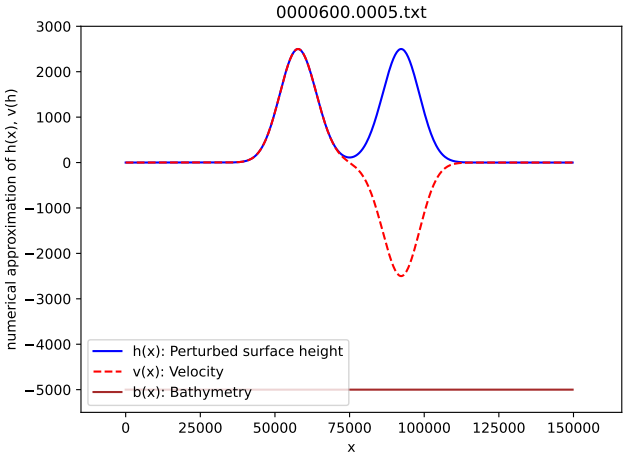
Note that we cope with two variables (height and velocity) rather than a single one!

4.2) Forward Euler

In the file `TimeStepperRK1.hpp` extend the method `TimeStepperRK1::time_integrate()` to time-integrate with a forward Euler scheme.

Hint: Forward Euler is known to be unstable in this case, but it's just to see that the results make any sense for small time step sizes.

Outcome: You should be able to see a (gravity-driven) wave propagating to the left and right side in the animation, see the following image.



4.3) 2nd order Runge-Kutta

In the file `TimeStepperRK2.hpp` extend the method `TimeStepperRK2::time_integrate()` to time integrate with a 2nd-order accurate explicit Runge-Kutta method. You can either use a version from the PDEs script or you can "Google" for different Runge-Kutta schemes.

Outcome: The simulation should look stable (but mathematically, it's not!) also for larger time step sizes.

4.4) Forward Euler / 4th order Runge-Kutta

Once the results look promising, implement the classical 4th-order Runge-Kutta method (which requires only 4 evaluations of the time tendencies).

Compare the results visually with the previous time integrators.

Outcome: The simulation should also be stable for even larger time step sizes.

5) Validation

This is a crucial step in basically all numerical developments.

Although the results are nice-looking, they might still be wrong.

One way to figure out potential (but not all) bugs hidden somewhere is to run convergence benchmarks.

5.1) Analytical solution

You can make a critical observation of the wave, namely that it resembles again very closely to the initial condition after a certain time. This allows us to take just the initial condition as a reference solution for error plots if we stop the simulation at this particular point in time and compute the error.

The wave speed will depend on the gravitational acceleration and the average surface height. Set these parameters to different values to determine the time it takes for a full propagation across the domain. The parameters (e.g., average height) you should use for the convergence

benchmark should differ from "1".

You should write your own unit test for this validation, which you can conveniently execute every time you change your program.

Optional (for experts): Try to compute an analytical solution at arbitrary points in time.

5.2) Convergence plots

Once given the analytical solution, create convergence error plots to ensure a certain order of accuracy.

6) Individual assignments (choose one)

Before you continue, create a **backup of your code**. You can choose one assignment of the following ones.

Each **group member** has to **choose a different assignment**!

6.1) Grid staggering (for those hating computational modes)

So far, we have implemented the differential operators solely based on the alignment of the discrete variables at the same points x_i. E.g., a $v(x_i)$ is available at the point x_i where we also store the height $h(x_i)$.

However, this is not the best choice for our problem due to so-called "computational modes" (see PDEs exercise). A better choice is to place the discrete values of one variable (in our case, the velocity) between the discretization points of the other ones (in our case, height).

The differential operator for v at x_i then reads

$$\frac{\partial}{\partial x} v(x_i) \approx \frac{v(x_{i+\frac{1}{2}}) - v(x_{i-\frac{1}{2}})}{\delta x} \tag{1}$$

and the one for h computed at $x_{i+\frac{1}{2}}$ reads

$$\frac{\partial}{\partial x} h(x_{i+\frac{1}{2}}) \approx \frac{h(x_{i+1}) - h(x_i)}{\delta x}. \tag{2}$$

- Implement this scheme
- Allow to choose the non-staggered or staggered grid layout via the command line parameter (by extending Config.hpp).
- Compare long-time running results between the staggered and non-staggered grid (maybe using the solitary wave described below). The staggered grid should provide better results.
- Make sure that all other assignments are compatible with this one (in particular, the solitary wave one)!

6.2) Solitary wave (for those loving to flip the pen for math)

Once you have implemented the wave equations correctly, you should see two waves that travel independently to each other. One to the left and one to the right.

- This assignment is to find an initial condition for the velocity v so that only one single wave travels to the right. This requires a little bit of understanding in the direction of the method of characteristics where we like to have a solution of the form $h(x,t) = h(x - ct, 0)$.
- Allow to choose the previous benchmark or this new solitary wave via the command line parameter (See Config.hpp).

6.3) Extension to bathymetry (for those interested in Tsunamis)

Search for the internet on how to include a bathymetry term in the shallow-water equations. Include a bathymetry in the form of a small bump and observe how the wave is changed while propagating over such a small bump.

Allow to choose between different bathymetries via the command line parameter (See Config.hpp).

Hint:

- You need to make the bump sufficiently large to observe something.
- You can only simulate the flooding of land area with this method if you implement various other things. Make sure that there is always water (surface height).

6.4) Extension to nonlinearities (for those interested in nonlinear interactions)

The shallow-water equations we have dealt with so far are entirely linear and have been derived from the nonlinear shallow-water equations.

- Implement some nonlinear parts of the nonlinear shallow-water equations (Internet...) and observe changes.
- Allow to choose between the linear and nonlinear equations via the command line parameter (See Config.hpp).

6.5) CMAKE (for those loving computer science)

The GNU "make" system, which we used so far, is not the only one to support us in developing programs. Another very popular one is "CMAKE". Look at their website `https://cmake.org/` and tutorials to replace GNU make with CMAKE.