Technische Universität Berlin Winter term 2025/26
Institute of Mathematics
Prof. Dr. T. Breiten, Prof. Dr. S. May, Dr. Juliette Dubois

Numerical Mathematics II for Engineers
Tutorial 1

Topics : Second-order difference quotients, 1D Poisson equation, 1D linear advection equation.

Discussion in the tutorials of the week 27–31 October 2025

Disclaimers:

- To test your code, we provide tests. This should help you to locate precisely your errors, and code more efficiently. To run the test, you need to install pytest[1].
- Exercise 1.3 concerns the linear advection equation; this topic will be covered in the lecture 03 on Tuesday 21.10.
- Exercises should be solved in **fixed groups of 3 students**.
- Hand in the solution in **one folder** labeled **hw1_group[group_number]** and containing:
  - **One pdf** for the theoretical questions and comments on the numerical results,
  - **One python file per programming exercise**: e.g. for this homework, you should have two python files: poisson.py and advection.py.
  - Write the group number and all names of your members **in each file**.

Exercise 1.1: 1D Poisson equation

We consider the Poisson problem:

$$-u''(x) = f(x) \quad \text{ for } x \in (0,1), \quad u(0) = a, \quad u(1) = b. \tag{1}$$

Complete the following tasks using the template `Poisson_template.py`. To test your code, run in a terminal `pytest poisson_test.py`.

(a) Given the boundary conditions $a = b = 0$ and the right-hand-side functions $f(x) = 1$ and $f(x) = \sin(\pi x)$, compute the exact solution of (1). Add these to the code (see `get_testproblem()`).
Hint: Integrate f twice.

(b) We want to code up the reduced system (1.21) from the lecture notes. First, we do it the "naive" way using a dense matrix representation. In the given code (`solver_type == "full"`), the diagonal and the off-diagonals are added to a `numpy` (full) matrix and the system is solved with `np.linalg.solve()`.

[1]https://docs.pytest.org/en/stable/getting-started.html. Only the section "Install pytest" is relevant for you!

Remark: In the code, the interval is discretized into $N+1$ intervals (instead of N intervals used in the lecture), leading to a matrix of size $N \times N$ (instead of a $(N-1) \times (N-1)$ matrix in the lecture).

(c) Run the code implemented in (b) to solve the two test problems defined in (a). Solve for the mesh resolutions $N \in [20, 40, 80, 160, 320, 640, 1280]$ and compute the maximum norm of the error $\max_i |u_i - u(x_i)|$. What convergence order do you get?

(d) Add timers around the solution step. What do you observe regarding runtime? The Gaussian elimination for a $N \times N$ matrix is in $\mathcal{O}(N^3)$. Using the runtime, compare the cost of solving the reduced system to a Gaussian elimination: is it the same, better, or worse? *Hint:* For timing, you can use, e.g., the functions `t = time.time()` and `t_solution = time.time() - t`.

(e) We want to examine the question: Does the usage of sparse matrices reduce the time to solution? For this task, you need to implement the case `solver_type == "sparse"`. A useful package for sparse linear algebra in Python is `scipy`. From here, you could use, e.g., the functions `scipy.sparse.csr_matrix()`, `scipy.sparse.diags()`, and `scipy.sparse.linalg.spsolve()`. How do the timings compare with the approach using the dense matrix?

Exercise 1.2: Difference quotient

We consider the second-order finite difference quotient D_2^c

$$D_2^c f(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \approx f''(x)$$

as well as D^+, D^-, and D_1^c to approximate $f'(x)$ from the lecture.

(a) Show that $D^+ D^- = D_2^c$.

(b) Show that $D_1^c D_1^c = \widetilde{D}_2^c$, where $\widetilde{D}_2^c$ is the second-order finite different quotient defined with a meshsize $2h$.

(c) How can you modify D_1^c to obtain D_2^c in part (b)?
Hint: Think about changing h.

Exercise 1.3: 1D linear advection equation

We consider the linear advection equation for $t \in (0,1]$ and $x \in (0,1)$

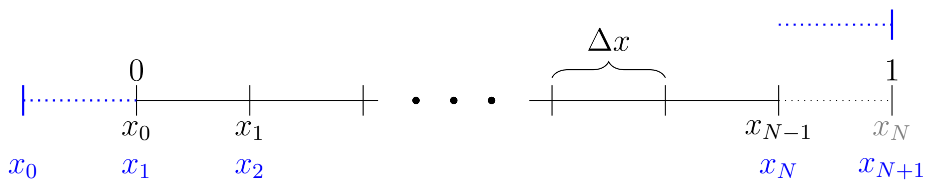
$$u_t + a u_x = 0, \quad a > 0 \text{ constant}, \tag{2}$$

with the periodic boundary conditions $u(t,1) = u(t,0)$ and initial data u_0. For $a = 1$, the exact solution at final time $T = 1$ is given by

$$u(1,x) = u_0(x) \quad \forall\, x \in (0,1).$$

You will solve this equation using two different schemes: FTCS (forward time and centered space) and upwind. The template `advection_template.py` is provided. To test your code, run in a terminal `pytest advection_test.py`.

Implementation of the boundary conditions: An often used approach for implementing boundary conditions is to use *ghost cells*. We describe the idea of ghost cells for periodic boundary conditions:



1. We create a mesh with N intervals, i.e. $\Delta x = 1/N$. The solution vector U is of size $N+1$.
2. Because of the periodic boundary condition, the grid point for $x = 0$ should have the same value as the one for $x = 1$, hence the solution vector has one unknown less: U is of size N. The first and last entries represent grid points located at $x_0 = 0$ and $x_{N-1} = 1 - \Delta x$.
3. Add now two ghost cells (shown in blue color), one on each end of the interval: U is extended to a vector of length $N+2$ (shown in blue color). The first and last entries represent grid points located at $x_0 = -\Delta x$ and $x_{N+1} = 1$. To realize periodic boundary conditions, we initialize our ghost cell values as

$$U_0 = U_N, \quad U_{N+1} = U_1.$$

Now we can begin the exercise:

(a) Complete in the template the function `compute_dt(CFL,a,dx)`, which computes the time step.

(b) Complete in the template the function updating the solution: `update_ftcs(U, a, dt, dx)` for the FTCS scheme, and `update_upwind(U, a, dt, dx)` for the upwind scheme. The update of the ghost cells is already implemented, only the part `U[1:-1]` should be updated in both functions.

(c) Complete in the template the function `compute_error(x, time, dx, uexact, U)`, which computes the maximum norm of the error $\max_i |u_i - u(x_i)|$.

(d) Test the FTCS and the upwind schemes with the smooth initial data

$$u_0(x) = \sin(2\pi x).$$

What do you observe for FTCS and for upwind? Estimate the convergence order of the upwind scheme using a convergence table.
Hint: The number of intervals is defined by $N = 40 \cdot 2^k$ where k is the refinement level. For the convergence table, you can change the value of `parameters["Nrefine"]`, which controls the number of refinement levels. If you set for example `N=40` and `Nrefine=2`, the code will automatically run the simulation (1) for $N = 40$, then (2) for $N = 80$, and then (3) for $N = 160$ (i.e., 2 additional refinement level) and compute convergence orders for you. The results are stored in `results/error.txt`.

(e) We want to test how the choice of the CFL number affects the solution. Solve the equation using the upwind scheme with the same initial data for $N = 40$ and $\nu \in \{0.8, 1., 1.2\}$. What do you observe? Which behavior was expected from the lecture?

Plotting solution in each step in python code is slow. You can increase `plot_freq` to a higher values so that the result is not shown in each time step. When you run the convergence test, you probably want to disable plotting by setting `plot_freq` to zero.