Series 1

ETH zürich

Advanced Numerical Methods for CSE
Last edited: October 17, 2019
Due date: November 1, 2019

Template codes are available on the course's webpage at `https://metaphor.ethz.ch/x/2019/hs/401-4671-00L/`.

Exercise 1 Linear advection equation

Consider the one-dimensional linear advection equation with variable coefficients

$$\begin{cases} \partial_t u(x,t) + a(x)\partial_x u(x,t) &= 0 & \text{for } (x,t) \in \mathbb{R} \times [0,\infty) \\ u(x,0) &= u_0(x) & \text{for } x \in \mathbb{R} \end{cases} \tag{1}$$

where $u_0 : \mathbb{R} \to \mathbb{R}$ is a given function. Let $a : \mathbb{R} \to \mathbb{R}$ be given by

$$a(x) = \begin{cases} 0, & \text{if } x \leq 0, \\ -x, & \text{if } 0 < x \leq 1, \\ -1, & \text{if } x > 1. \end{cases} \tag{2}$$

1a)

Use the method of characteristics to find the solution $u(x,t)$, for $(x,t) \in \mathbb{R} \times [0,\infty)$, and a generic initial condition u_0.

1b)

Determine now the solution $u(x,t)$ of the one-dimensional linear conservation law

$$\begin{cases} \partial_t u(x,t) + \partial_x\big(a(x)u(x,t)\big) &= 0 & \text{for } (x,t) \in \mathbb{R} \times [0,\infty) \\ u(x,0) &= u_0(x) & \text{for } x \in \mathbb{R} \end{cases} \tag{3}$$

for $(x,t) \in \mathbb{R} \times [0,\infty)$, and a generic initial condition u_0.

Hint: Make use of the characteristics found in **1a)**.

1c)

For the two solutions computed in **1a)** and **1b)**, respectively, what can be said about $\int_{\mathbb{R}} |u(x,t)|\,dx$ and $\max_{x\in\mathbb{R}} |u(x,t)|$ as a function of time t?

1d)

Let $u(x,t)$ be a solution of the conservation law (3), where we now consider a general coefficient $a \in C^1$. Assume that $u(x,t)$ is C^1 and that $\lim_{|x|\to\infty} u(x,t) = 0$ for all $t \geq 0$ (you may assume $u(x,t)$ to be well-behaved at $x = \pm\infty$ to avoid technical difficulties). Show that $u(x,t)$ is conserved, in the sense that $\int_{\mathbb{R}} u(x,t)\,dx = \int_{\mathbb{R}} u_0(x)\,dx$ for all $t \geq 0$. Is this also true for the linear advection equation?

Exercise 2 Initial value problem for Burgers' equation

2a)

Find the unique entropy solution $u : \mathbb{R} \times [0, \infty) \to \mathbb{R}$, $(x, t) \mapsto u(x, t)$ of the Burgers equation

$$\partial_t u + \partial_x \left(\frac{u^2}{2} \right) = 0, \tag{4}$$

with the following initial data

$$u_0(x) = \begin{cases} 0, & (x < -1), \\ 1, & (-1 < x < 0), \\ 1 - x, & (0 < x < 1), \\ 0, & (x > 1). \end{cases} \tag{5}$$

Exercise 3 Riemann problems for nonlinear fluxes

We consider the Riemann problem for the non-linear conservation law

$$\partial_t u + \partial_x f(u) = 0. \tag{6}$$

where $u : \mathbb{R} \times [0, \infty) \to \mathbb{R}$, $(x, t) \mapsto u(x, t)$, and initial data of the form

$$u_0(x) = \begin{cases} u_L, & (x < 0), \\ u_R, & (x > 0). \end{cases} \tag{7}$$

For each of the following flux functions $f(u)$ and initial data u_0, determine the unique entropy solution:

3a)

$$f(u) = u(1-u), \quad u_0(x) = \begin{cases} 1/2, & x < 0, \\ 0, & x > 0. \end{cases}$$

3b)

$$f(u) = u(1-u), \quad u_0(x) = \begin{cases} 0, & x < 0, \\ 1/2, & x > 0. \end{cases}$$

3c)

$$f(u) = \exp(u), \quad u_0(x) = \begin{cases} 0, & x < 0, \\ 1, & x > 0. \end{cases}$$

3d)

$$f(u) = \exp(-u), \quad u_0(x) = \begin{cases} 1, & x < 0, \\ 0, & x > 0. \end{cases}$$

Exercise 4 Finite Volume Method for scalar equations in 1D

In this exercise we implement finite volume methods for the one-dimensional, scalar hyperbolic conservation law

$$u_t + f(u)_x = 0, \tag{8}$$

with flux $f(u) = \frac{1}{2}u^2$ on a uniform grid. The cell-centers are denoted by x_i and the interfaces by $x_{i+1/2}$. The semi-discrete formulation of FVM is

$$\frac{d}{dt}U_i + \frac{1}{\Delta x}\left(F_{i+1/2} - F_{i-1/2}\right) = 0 \tag{9}$$

where $F_{i+1/2}$ is the numerical flux through the interface $i+1/2$ and U_i the approximate cell-average of u. The numerical fluxes we consider are two point fluxes, given by

$$F_{i+1/2} = F(U^-_{i+1/2}, U^+_{i+1/2}) \tag{10}$$

with traces $U^{\pm}_{i+1/2}$, i.e. approximations of the value of $u(x_{i+1/2}, t)$ from the left and right of the interface.

Hint: The templates are a simplified version of our research codes. Therefore please read the README.md which comes with the code. It will walk you through the individual parts of the exercise.

Hint: The subproblems are ordered, and it's recommended you solve them in the stated order.

Hint: The file config.json is used to configure the simulation. The path the this file has been hard-coded. Depending on your setup you might need to change edit src/ancse/config.cpp to used the right path for your system. This is likely the case if you're using VS2019. The path should be relative to the executable.

4a)

Implement the following numerical fluxes:

- Rusanov's
- Lax-Friedrichs
- Roe
- Godunov
- Enquist-Osher

Hint: The file include/ancse/numerical_flux.hpp contains an example of how to implement the central flux, called CentralFlux.

Hint: Write tests to check for the correctness of your implementation. You can find the tests in tests/test_numerical_flux.cpp.

4b)

Implement piecewise linear reconstruction of the trace values $U^{\pm}_{i+1/2}$, with the following slope-limiters:

- minmod
- minabs
- superbee
- monotonized central
- van Leer's limiter

Hint: The file include/ancse/reconstruction.hpp contains an example of how to implement piecewise constant reconstruction, called PWConstantReconstruction.

Hint: Consider implementing this using a template class which accepts the slope-limiter as a template parameter.

Hint: Implement tests in tests/test_reconstruction.cpp.

4c)

Complete the loop that applies your fluxes and numerical reconstructions in fvm_rate_of_change.hpp.

Hint: Boundary conditions haven't been yet implemented, but assume they are in place when writing this class. That is, you can trust the ghost cells to contain appropriate values, and you only need to update the central, non-ghost cells here.

4d)

Implement the second-order strong stability preserving Runge-Kutta (SSP2) scheme.

Hint: The files include/ancse/runge_kutta.hpp and src/ancse/runge_kutta.cpp contains an example of how to implement forward Euler, c.f. ForwardEuler.

4e)

Implement periodic and outflow boundary conditions.

Hint: Please implement your boundary conditions by deriving from BoundaryCondition, found in include/ancse/boundary_condition.hpp.

4f)

Implement the CFL condition. This should be a new class that derives from CFLCondition, found in include/ancse/cfl_condition.hpp, which implements the computation of a CFL-satisfying dt for a generic problem.

Hint: Please remember to test your implementation.

4g)

To enable selecting the different schemes at run time we need to implement factories for each component.

Start by registering your implementation of SSP2, see src/ancse/runge_kutta.cpp. Note, there is already an example of what to do for ForwardEuler .

Next, register your two boundary conditions in src/ancse/boundary_condition.cpp and the CFL condition in src/ancse/cfl_condition.cpp.

Finally, the hardest part: registering the numerical flux and reconstruction. You'll find an example of how to do this in src/ancse/fvm_rate_of_change.cpp.

4h)

You now have a code with which you can discover the behaviour of a large number of numerical schemes.

Here are some ideas of what you can do:

- Observe how the numerical schemes behave on a smooth test case, e.g.

$$u(x, t = 0) = u_0(x) = \sin(2\pi x) \tag{11}$$

  on a domain $D = [0, 1]$ with periodic boundary conditions.

- The Roe flux is known to produce an entropy violating shock instead of a rarefaction. You can see this behaviour by looking at the following two step functions

$$u_0(x) = \begin{cases} -1 & x < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{12}$$

and

$$u_0(x) = \begin{cases} 1 & x < 0.5 \\ -1 & \text{otherwise.} \end{cases} \tag{13}$$

Use a domain $D = [0, 1]$ and outflow boundary conditions. Choose the final time such that the waves don't reach the boundary.

- You can investigate which solves are stable in the presence of discontinuities. Compare different slope-limiters. As initial condition you could again pick a step function and use outflow boundary conditions.
- How sharp is the CFL condition?

Hint: Use piecewise linear reconstruction with minmod, Rusanov's flux and SSP2 timestepping on 2048 cells as a reference solution.

Hint: When assessing the performance of a scheme use a moderate number of cells, in the range of 10s to 100s.

4i)

Optional. Since you now have a 1D version of our research codes, you can do a number of other interesting things.

- Implement the third-order strong stability preserving method. You can read up on this in the review paper by Gottlieb, Shu and Tadmor:

  `https://doi.org/10.1137/S003614450036757X`

  Of particular interest is the introduction and Section 4 "Nonlinear SSP Runge–Kutta Methods".

  This task is not very hard and the paper contains some surprising (and depressing) results about strong stability preserving Runge-Kutta schemes.

  One natural question is: why can't we just use standard RK methods such as *the* fourth-order Runge-Kutta method?

- Implement fifth order WENO. You could read this well-written review paper by Shu:

https://doi.org/10.1137/070679065

Keep in mind that you are implementing a finite volume method, not a finite difference method. In particular, pay attention to the difference of WENO *interpolation* and WENO *reconstruction*.

Only the introduction, Section 2.1 "WENO Interpolation" and Section 2.2 "WENO reconstruction" are needed for this task.

This task is interesting because you will learn about the difference of cell-averages and point-values, which only becomes important at third-order. Furthermore, WENO is one of the most powerful reconstruction schemes available; and once you have the formulae it should be straight forward to implement it in this code.

- You could parallelize the code by using either OpenMP or MPI. Both are currently standard ways of parallelizing numerical codes.
- Writing the output to JSON is not scalable. The two real options are HDF5 and NetCDF. You could implement a subclass of SnapshotWriter which would use either one of these libraries to write the output.

  This task requires some substantial coding and debugging, but these libraries are currently the two most popular ways of writing simulation output to a file. Therefore, it's worth learning them.

Exercise 5 Linear transport equation in 2D

Until now we have considered the scalar linear advection equation only in 1D, i.e. $u_t + au_x = 0$, $u(x,0) = u_0(x)$. In this exercise, we are going to study the natural generalization to more than one dimension. In other words: given $\Omega \subset \mathbb{R}^2$ and $T > 0$, we want to find $u : \Omega \times [0,T] \to \mathbb{R}$ such that

$$\partial_t u(\mathbf{x},t) + \mathbf{a}(\mathbf{x}) \cdot \nabla_{\mathbf{x}} u(\mathbf{x},t) = 0, \qquad \forall (\mathbf{x},t) \in \Omega \times [0,T] \tag{14}$$

$$u(\mathbf{x},0) = u_0(\mathbf{x}), \qquad \forall \mathbf{x} \in \Omega \tag{15}$$

which can be written as $\partial_t u + a_1 \partial_x u + a_2 \partial_y u = 0$, with $\mathbf{a} = (a_1, a_2) \in C^1(\Omega; \mathbb{R}^2)$ the advecting velocity. We call $\nabla_{\mathbf{x}} u := (\partial_x u, \partial_y u)$, $\mathbf{x} = (x,y)$, and $\cdot$ denotes the Euclidean scalar product.

For this problem, we take $\Omega = [a,b] \times [c,d] \ni \mathbf{0}$, and we study the specific choice of

$$\mathbf{a}(x,y) := (y, -x). \tag{16}$$

This choice corresponds to the **rotation of a solid** around the origin.

5a)

Characteristic curves in this case are still defined as the curves $\boldsymbol{\gamma} = (\gamma_1, \gamma_2) : [0,T] \to \mathbb{R}^2$ such that

$$\frac{d}{dt} \boldsymbol{\gamma}_{\mathbf{x_0}}(t) = \mathbf{a}(\boldsymbol{\gamma}_{\mathbf{x_0}}(t)) \tag{17}$$

$$\boldsymbol{\gamma}_{\mathbf{x_0}}(0) = \mathbf{x_0}. \tag{18}$$

Find the explicit expression of characteristic curves for problem (14) with velocity $\mathbf{a}$ chosen as in (16), and draw a sketch.

5b)

Let $C_{ij} = [x_i, x_{i+1}) \times [y_j, y_{j+1}) \subset \Omega$, and let $\mathbf{a}$ be an arbitrary **divergence free** advection velocity, i.e. $\nabla_{\mathbf{x}} \cdot \mathbf{a} = 0$. Derive from eq. (14) the following equality:

$$\partial_t \frac{1}{|C_{ij}|} \int_{C_{ij}} u(\mathbf{x},t) d\mathbf{x} + \frac{1}{|C_{ij}|} \int_{x_i}^{x_{i+1}} \left(a_2(x, y_{j+1}) u(x, y_{j+1}) - a_2(x, y_j) u(x, y_j)\right) dx \tag{19}$$

$$+ \frac{1}{|C_{ij}|} \int_{y_j}^{y_{j+1}} \left(a_1(x_{i+1}, y) u(x_{i+1}, y) - a_1(x_i, y) u(x_{i+1}, y)\right) dy = 0 \tag{20}$$

where $|C_{ij}| = (x_{i+1} - x_i)(y_{j+1} - y_j)$ is the volume of C_{ij}.

Hint: Remember the **divergence theorem** (or Gauss' theorem): if $V \subset \mathbb{R}^d$ is a compact domain with piecewise smooth boundary, and $F \in C^1(U)$ for U an open set containing V, then

$$\int_V (\nabla_{\mathbf{x}} \cdot F) dV = \int_{\partial V} (F \cdot \nu) dS \tag{21}$$

where $\nu(\mathbf{s})$ is the unit **outward-pointing** vector normal to ∂V at point $\mathbf{s}$.

Hint: What do you know about $\nabla_{\mathbf{x}} \cdot (u\mathbf{a})$?

5c)

We will use (19) to implement a **first-order** finite volume numerical scheme for solid rotation in two dimensions. For that, set a domain $\Omega = [x_{min}, x_{max}) \times [y_{min}, y_{max})$, and fix parameters N_x, N_y. Let $\Delta x = (x_{max} - x_{min})/N_x$, $\Delta y = (y_{max} - y_{min})/N_y$. For $i \in \{0, \ldots, N_x\}$, $j \in \{0, \ldots, N_y\}$, define the **Cartesian grid**

$$x_i := x_{min} + i\Delta x, \qquad y_j := y_{min} + j\Delta y.$$

The grid is depicted in Figure 1.

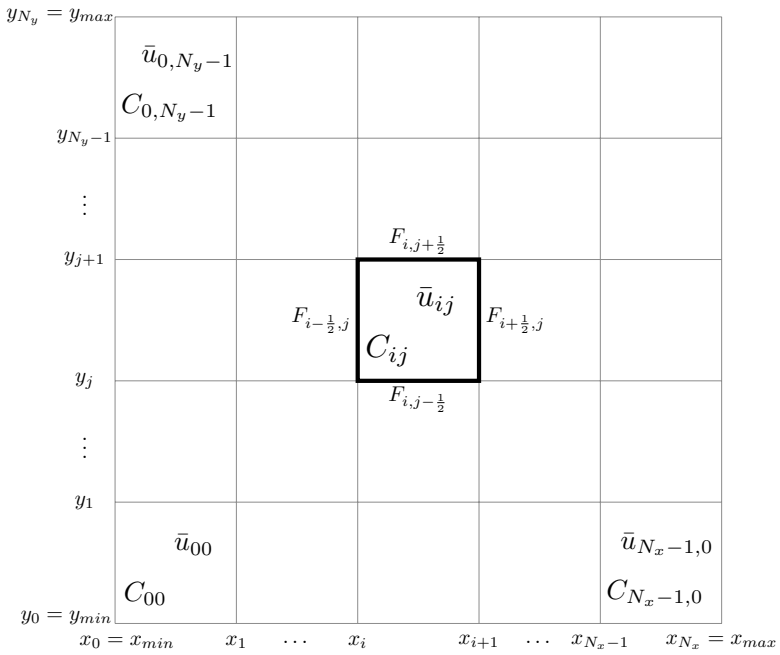

Figure 1: The Cartesian grid.

In file `linear_transport.cpp`, complete function `applyBoundaryConditions`, that fills the first and last row and column of an `(Nx+2) x (Ny+2)` matrix with the appropriate values so that **periodic boundary conditions** are used.

5d)

We want to solve problem (14)-(15), with $\Omega = [-1, 1]^2$, $T = 2$, and

$$u_0(\mathbf{x}) := \chi_{[-0.3,0.3]^2} = \begin{cases} 1 & \text{if } \mathbf{x} \in [-0.3, 0.3]^2 \\ 0 & \text{otherwise} \end{cases}$$

Write an initialization loop for variable `u` in `main.cpp` using provided function `ic`.

Hint: You can use point evaluations $U^0_{i,j} := u_0(x_i + \frac{1}{2}\Delta x,\ y_j + \frac{1}{2}\Delta y)$ rather than compute cell averages. Why is this OK? Would it be an acceptable strategy for a higher-order scheme?

Hint: You may find useful the attached script `plot_init.py`, which plots the initial condition.

5e)

To conclude the implementation, we need to code a discretization of (19)-(20), which depends on several non-trivial integrals at the boundary. For that, use the following (at least) **first-order** accurate estimates:

- $\int_a^b f(x)dx \approx (b-a) f\left(\frac{a+b}{2}\right)$ (midpoint quadrature rule)
- For all $(x, y) \in C_{ij}$, $u(x, y) \approx \bar{u}_{ij}$ (follows from 2D midpoint quadrature rule)
- Remember to apply an **upwind criterion** to choose an approximation of u at cell interfaces!

Using the above approximations, find numerical fluxes $F_{i,j+\frac{1}{2}}$, $F_{i,j-\frac{1}{2}}$, $F_{i+\frac{1}{2},j}$, $F_{i-\frac{1}{2},j}$ where e.g.

$$F_{i,j+\frac{1}{2}} \approx \int_{x_i}^{x_{i+1}} a_2(x, y_{j+1}) u(x, y_{j+1}) dx,$$

and use them to implement function `updateUpwind`, which takes the value of the solution at time-step t^n (as `u_old`) and updates `u` with the values at time t^{n+1}.

Finish your implementation by calling your function in the main loop of the program.

Remark: as a CFL condition, we can use the following (sub-optimal) generalization [1] of 1D CFL:

$$\left| \frac{\Delta t}{\Delta x} \max_{\mathbf{x}\in\Omega} a_1(\mathbf{x}) \right| + \left| \frac{\Delta t}{\Delta y} \max_{\mathbf{x}\in\Omega} a_2(\mathbf{x}) \right| \leq 1$$

[1] A derivation of this expression, together with a less restrictive alternative, can be found e.g. in R. Leveque's *Finite Volume Methods for Hyperbolic Problems*, chapter 20.

Remark: You can modify the parameters of the simulation in file `config.json`. What happens with the result of the simulation as you increase the number of points? How does the runtime of the program scale?

Remark: Note that the path to `config.json` has been hard-coded. You might need to adjust this for you setup, e.g. if you're using VS2019.

Remark: You can generate an animation of your solution with the provided Python script `sol_movie.py`.