Project #3 in FMNN10 and NUMN20

Goals. In this project, we will work with parabolic and hyperbolic partial differential equations, and combine elementary techniques from 2p-BVPs with time-stepping methods from IVPs. The goal is to gain experience with the method of lines and stability constraints of Courant-Friedrichs-Lewy (CFL) type on the time-step Δt.

In the first part we shall primarily work with the parabolic diffusion equation $u_t = u_{xx}$. In the second part we will work with conservation laws such as the hyperbolic advection equation $u_t = -u_x$. In the third part, we combine the two and consider the parabolic convection–diffusion equation $u_t = \epsilon u_{xx} - u_x$. Finally, in the fourth part we change the problem to the parabolic viscous Burgers equation, $u_t = \epsilon u_{xx} - uu_x$.

The objective is to gain an elementary understanding of and experience with "evolution equations" (time dependent partial differential equations), and the many varying properties one encounters in such problems, from dissipation to wave propagation, including shock formation. In addition, we will work with different types of boundary conditions, and it is important to implement boundary conditions correctly in order to get the correct results.

We will only work in one space dimension, and on equidistant grids. We will use explicit and implicit time-stepping methods, in order to study the properties of some important elementary finite difference methods. We will observe some undesirable phenomena such as numerical instability and numerical damping, and learn how to cope with them.

Part 1. The diffusion equation

The diffusion equation is

$$\begin{aligned} u_t &= u_{xx} \\ u(t,0) &= u(t,1) \;=\; 0 \\ u(0,x) &= g(x) \end{aligned}$$

and its method-of-lines semi-discretization is

$$\dot{u} = T_{\Delta x} u$$

on an equidistant grid on $[0,1]$, with $\Delta x = 1/(N+1)$ and initial condition $g(x)$ sampled on the grid. Generate the grid using `linspace`, so that you have

N internal points on $(0,1)$. The matrix $T_{\Delta x}$ is the usual Toeplitz symmetric tridiagonal matrix that approximates $\partial^2/\partial x^2$ to second order accuracy on the grid. In Project 2 you have generated this (and similar operators). Go back to your old programs and retrieve the code segments you need in order to create $T_{\Delta x}$. You will need (some representation of) the operator $T_{\Delta x}$ in order to solve equation systems involving this matrix.

Next, get your function `eulerstep` from Project 1 and make sure that it takes one time step, and that it works in the matrix–vector case. It will probably be necessary to make some minor modifications so that you can call `eulerstep` with a command of the type

```
unew = eulerstep(Tdx,uold,dt)
```

in order to approximate the vector-valued equation $\dot{u} = T_{\Delta x}u$ using the explicit Euler method, $u^{m+1} = u^m + \Delta t \cdot T_{\Delta x}u^m$.

Task 1.1 Write a script that constructs $T_{\Delta x}$, assuming `N` interior points on the grid and solves the semidiscretization $\dot{u} = T_{\Delta x}u$ by the Euler method.

1. Visualize the computational procedure by plotting the numerical solution $u(m\Delta t, n\Delta x)$ over the t, x plane. Basically, you should apply the following procedure. You determine your spatial grid size Δx and time step Δt from the number N of internal mesh points on the x-interval $[0,1]$ and the number M of time steps to go from $t = 0$ to $t = t_{\text{end}}$, respectively. Always run your experiments by varying N and M, *not* by varying Δx and Δt directly. Then construct a vector

   ```
   xx = linspace(0,1,N+2)
   ```

   and a vector

   ```
   tt = linspace(0,tend,M+1)
   ```

   and run `[T,X]=meshgrid(tt,xx)` to generate the independent variables for 3D graphics in MATLAB. If you are not familiar with these commands, check MATLAB's `help`. In Python, the `meshgrid` function exists directly in the base `scipy` module.

   When you solve the PDE, save the solution vectors over the meshed grid. Don't forget to insert the boundary values! When the computation is done, make a 3D color graph using either the `mesh` or the `surf` command. (Use `help` in MATLAB to find out how they work.)

In Python, this is slightly more involved. See e.g. `https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html` for a tutorial on how to do this in `matplotlib`. The functions roughly equivalent to `mesh` and `surf` are called `plot_wireframe` and `plot_surface`.

2. Determine experimentally the CFL condition on the time step Δt for the explicit Euler method applied to the semi-discretization. Choose your initial condition $g(x)$ as you like, but make sure that they are compatible with the boundary conditions at $x = 0$ and $x = 1$, respectively, and make sure that $g(x)$ is *not an eigenfunction of* $\partial^2/\partial x^2$.

3. Try to obtain representative plots of the solution both when you fulfill the CFL condition and when you violate it. In the latter case, don't overdo it; try to see what happens when you violate the CFL condition by a small amount only, so that the numerical instability is clearly visible.

Task 1.2 Next implement the Crank–Nicolson method. This is the same as applying the trapezoidal rule to the equation,

$$u^{n+1} = u^n + \frac{\Delta t}{2}\,(T_{\Delta x}u^n + T_{\Delta x}u^{n+1}).$$

Implement a function

```
unew = TRstep(Tdx,uold,dt)
```

Here you will need to solve a linear equation system. Do this by \ in MATLAB or by `numpy.linalg.solve` in Python.

Solve the same diffusion equation as in Task 1. Verify that you can now obtain stable solutions where you previously observed instability due to a violated CFL condition. Try to verify that you can use much larger time steps Δt than before, but remember to vary time step and Courant number by varying N and M. Check that this works no matter what initial conditions you use.

Part 2. The advection equation

The linear advection equation is

$$u_t + au_x = 0,$$

with appropriate initial and boundary conditions. We will only work with *periodic boundary conditions.* This means that the solution satisfies $u(t, 0) =$

$u(t, 1)$ for all times t. That is, the solution "wraps around" from $x = 1$ to $x = 0$, and vice versa.

With periodic boundary conditions, we represent the solution u *on the entire grid, including the "boundaries"* throughout the entire computation. Let the solution vector at time t be an N-vector and introduce an equidistant grid on $[0, 1]$, such that $0 = x_1, \ldots, x_j = (j-1)\Delta x, \ldots, x_N = (N-1)\Delta x$ and $x_{N+1} = 1$.

This implies that $\Delta x = 1/N$, and that we are going to compute the solution at the N points $x_1, \ldots x_N$. The solution at the right boundary, $x_{N+1} = 1$, is u_{N+1} and is identified with u_1 at the left boundary, $x_1 = 0$. Make sure you have understood exactly what this implies (make a sketch of the grid!), as your code will differ from what you did in the parabolic case.

Initial conditions are always needed, and have the form

$$u(0, x) = g(x).$$

Note that the initial condition must satisfy $g(0) = g(1)$ and $g'(0) = g'(1)$ when periodic boundary conditions are used. *Do not use a trigonometric initial condition such as* $\sin 3\pi x$.

One of the simplest methods of order 2 for the advection equation is the *Lax–Wendroff scheme.* It can be derived using a Taylor series expansion. Thus,

$$u(t + \Delta t, x) = u(t, x) + \Delta t u_t + \frac{\Delta t^2}{2!} u_{tt} + \mathrm{O}(\Delta t^3).$$

From the differential equation $u_t + au_x$ we get $u_{tt} = -au_{xt}$, as well as $u_{tx} = -au_{xx}$. Therefore we have $u_{tt} = a^2 u_{xx}$ provided that the solution u is sufficiently differentiable. Inserting this into the Taylor series expansion we get

$$u(t + \Delta t, x) = u(t, x) - a\Delta t u_x + \frac{a^2 \Delta t^2}{2!} u_{xx} + \mathrm{O}(\Delta t^3).$$

This gives the Lax–Wendroff scheme,

$$u_j^{n+1} = \frac{a\mu}{2}(1 + a\mu)u_{j-1}^n + (1 - a^2\mu^2)u_j^n - \frac{a\mu}{2}(1 - a\mu)u_{j+1}^n,$$

where $\mu = \frac{\Delta t}{\Delta x}$. Note that the method coefficients are not symmetric – instead, they change with the flow direction as determined by the sign of a.

Task 2.1 Implement a Lax–Wendroff solver for the scalar advection equation with periodic boundary conditions. Write a function

```
function unew = LaxWen(u, amu)
```

that takes a step of size Δt starting from the "initial condition" `u`. The parameter `amu` is the product $a\Delta t/\Delta x$. Write a script for running your Lax–Wendroff solver. Test your solver using some pulse-like initial data, e.g.

$$g(x) = \mathrm{e}^{-100(x-0.5)^2}.$$

Note that this function does not satisfy the boundary conditions exactly, but the discrepancy is small enough that it is dominated by the $\mathcal{O}(\Delta x^2)$ error. Check how this pulse propagates for 5 units of time, and verify that the method works both for positive and negative a. Pay close attention to how you implement the periodic boundary conditions (sketch the grid and the computational stencil before you write the program).

Produce plots of the solution for at least two different CFL numbers. In addition, plot the L^2 norm (RMS norm) of the solution vs time $t \in [0, 5]$ for $a\Delta t/\Delta x = 1$ as well as for $a\Delta t/\Delta x = 0.9$, and explain the difference. Is the Lax-Wendroff method conserving the norm of the solution? Does the amplitude remain constant, or is it damped? Motivate why one actually wants to run the code at the CFL stability limit.

Part 3. The convection–diffusion equation

The linear convection–diffusion equation is

$$u_t + a \cdot u_x = d \cdot u_{xx}$$

with appropriate initial and boundary conditions at both boundaries. Here a is the convection velocity, and d is the diffusivity. Introduce an equidistant grid on $[0, 1]$ with $\Delta x = 1/N$ and work with periodic boundary conditions only, as you did with the advection equation. This means that the boundary conditions are $u(t, 0) = u(t, 1)$. Initial conditions have the form $u(0, x) = g(x)$. Pick $g(x)$ such that $g(0) = g(1)$ and $g'(0) = g'(1)$. (The latter condition could very well be approximate, like when you use the pulse data from the previous task.)

It is important to note that due to the diffusion term, the problem is always stiff. Therefore, it is preferable to work with implicit methods for the time-stepping. We choose the trapezoidal method, which is second order in time. Combined with a second order in space discretization, one can expect to obtain a reasonably good solver.

There are some unexpected difficulties associated with convection–diffusion. A straightforward method of lines discretization gives (the description below disregards the periodic boundary conditions, which you will have to implement – the matrices will be of *circulant* type)

$$\dot{u} = (d \cdot T_{\Delta x} - a \cdot S_{\Delta x})\, u.$$

Here $T_{\Delta x}$ is symmetric, while $S_{\Delta x}$ is skew-symmetric. The former has negative real eigenvalues, and the latter purely imaginary eigenvalues. As the total differential operator has negative real eigenvalues, it is important that the discretization mimics this property. In the MOL equation above, it is easy to see that the subdiagonal matrix elements are

$$\frac{d}{\Delta x^2} + \frac{a}{2\Delta x}$$

while the superdiagonal elements are

$$\frac{d}{\Delta x^2} - \frac{a}{2\Delta x}.$$

Given that a and d are positive, it can be proven that the matrix $d \cdot T_{\Delta x} - a \cdot S_{\Delta x}$ has real eigenvalues if and only if the subdiagonal and superdiagonal elements have the same sign. We must therefore require

$$\frac{d}{\Delta x^2} > \frac{a}{2\Delta x}.$$

The ratio $\mathrm{Pe} = |a/d|$ is known as the *Péclet number*. It measures the balance between the convective and diffusive terms. At high Péclet numbers, the solution is convection dominated; for small Pe diffusion dominates. The condition above can be written

$$\mathrm{Pe}\Delta x < 2,$$

and expresses that you have to choose Δx small when the Péclet number is large, i.e., when convection dominates. However, when you choose a small Δx, it becomes problematic to use an explicit time-stepping method, as this will have to fulfill a CFL condition, which roughly is

$$d \cdot \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}.$$

Therefore, explicit time stepping methods are not well suited to convection–diffusion problems.

The quantity $\mathrm{Pe}\Delta x$ is called the *mesh Péclet number* and is somewhat similar to the Courant number, as both put restrictions on the discretization parameters. The mesh Péclet condition, however, is independent of time-stepping and stability; it is a matter of whether the discretization has properties similar to those of the differential operator. If the mesh Péclet condition is violated, one typically observes "non-physical" oscillations in the numerical solution. This means, as a matter of course, that the error is large, due to the discrete equation being a poor approximation to the differential operator.

Task 3.1 Write a convection–diffusion solver

```
function unew = convdif(u,a,d,dt,dx)
```

for the convection–diffusion equation on $[0, 1]$ and for $t \in [0, 1]$. The single step from `u` to `unew` should be carried out using the Trapezoidal Rule. Boundary conditions should be periodic.

When writing a script to test the solver, use $a = 1$ and the diffusivity $d = 0.1$. Feel free to choose your own parameters once you get your code to work, but **take notice:** $d \geq 0$. (Why?) You can, however, choose the sign of a as you please. Your `convdif` code should work well for $\mathrm{Pe} \sim 0 - 1000$. The script should keep the mesh Péclet number in check, so that you can experiment with your code, both within and outside theoretical limitations.

Start with some suitable initial condition. You can try the same initial pulse as before if you like, but you might get more exciting plots by choosing a less symmetric initial function. Make sure your initial function fulfills the periodic boundary conditions. Plot and report your tests, demonstrating that you have a working code.

Part 4. The viscous Burgers equation

The viscous Burgers equation is $u_t + uu_x = d \cdot u_{xx}$. Note that this problem is nonlinear, so *a simple matrix-vector representation won't work.* It is a nonlinear convection–diffusion equation.

We are going to build a solver for this equation using the tools you have developed above. However, we are going to find out that due to the nonlinearity the problem is far more complicated than one might think.

To obtain a discretization that is of second order at $d = 0$, note that you cannot just replace the advection speed a by u in your Lax–Wendroff solver. Instead, you have to start anew from

$$u(t + \Delta t, x) \approx u(t, x) + \Delta t u_t + \frac{\Delta t^2}{2!} u_{tt}$$

Using the *inviscid Burgers equation* $u_t = -uu_x$, express u_{tt} in terms of space derivatives (u, u_x and u_{xx}) and *replace the time derivatives* with these expressions to obtain the Lax–Wendroff discretization of $u_t = -uu_x$, where all space derivatives are approximated by *symmetric* finite differences. Derive this scheme!

The scheme will look a little more complicated than it did for $u_t = -au_x$, and will lead to a recursion

$$\texttt{unew} = \mathrm{LW}(\texttt{uold}),$$

where you need to construct the map $u \mapsto \mathrm{LW}(\texttt{u})$. We are not going to run this scheme; we first need to *add the diffusion term.* Because the latter is stiff, we will treat it by the Trapezoidal Rule. This means that we will add the diffusion as follows,

$$\texttt{unew} = \mathrm{LW}(\texttt{uold}) + d \cdot \frac{\Delta t}{2}(T_{\Delta x}\texttt{unew} + T_{\Delta x}\texttt{uold}).$$

Note that $T_{\Delta x}$ is a *circulant matrix* to account for the periodic boundary condition.

Because the scheme is now implicit, we have to solve for `unew`, from the linear system

$$\left(I - d \cdot \frac{\Delta t}{2} T_{\Delta x}\right) \texttt{unew} = \mathrm{LW}(\texttt{uold}) + d \cdot \frac{\Delta t}{2} T_{\Delta x}\texttt{uold}.$$

You have then obtained a simple second order discretization of the viscous Burgers equation, where the nonlinear convective part is treated by the explicit Lax–Wendroff method, and the linear diffusion part is taken care of by the Trapezoidal Rule. Note that while the scheme *is* second-order convergent, the above reasoning does *not* constitute a rigorous proof of this fact. For this project, however, we omit such a verification.

Task 4.1 Implement the Lax-Wendroff scheme, and write a script for the repeated time stepping, where you collect the solution data for plotting. Choose a fairly low diffusivity (start with say $d = 0.1$) so that you can see some wave propagation. Use some initial condition of your choice (for instance the pulse data, but make sure it has a high enough amplitude) and use periodic boundary conditions. Try to see if you can produce some waves with very steep gradients (near shocks) without the method going unstable. Plot at least one interesting example. It is advisable to take N in the range 200 – 300, and to use at least $M = 1000$ time steps. You have a well functioning code if you can manage a diffusivity of $d = 0.01$. Your code might go a little bit lower, but note that it won't work for $d = 0$ if the initial condition has a high enough amplitude to generate shock waves.

Your code is not designed to solve problems with discontinuous solutions, so you can expect to experience difficulties and instability. You may have to experiment quite a lot with the parameters and initial data in order to obtain a numerical solution.

Be sure to identify every plot with a title, axis labels and a caption explaining what you see in the plot and what conclusions you can draw from it. Include relevant functions, scripts and code segments to document your solver.

Acknowledge help from fellow students and instructors, literature references etc., and summarize what you have learned both in this assignment, and about the numerical solution of differential equations in this course.