Project #2 in FMNN10 and NUMN12

Goals. In this assignment, the goal is to construct solvers for two-point boundary value problems as well as Sturm–Liouville eigenvalue problems. The objective is to learn the basic finite difference methodology by implementing it in detail. As it is too complicated to make the solver adaptive (i.e., the mesh width varies along the solution), we only construct solvers for equidistant grids. Likewise, because nonlinear problems present additional difficulties, we will only consider linear problems.

Prepare yourself before you go to the computer lab! Read the entire instruction, work out answers or formulas that you will need in your programs, and make sure that you have a plan for your work. Get started with the programming before you go to the lab, in order to make the most of your time there.

Part 1. 2pBVP solvers

Theory and problem statement. Consider a simple 2pBVP

$$\begin{aligned} &y'' = f(x, y), \\ &y(0) = \alpha, \quad y(L) = \beta. \end{aligned}$$

Introduce an equidistant grid on $[0, L]$ with $\Delta x = L/(N+1)$. The derivative is discretized with a symmetric finite difference quotient, so that we obtain

$$\begin{aligned} &\frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} = f(x_i, y_i), \\ &y_0 = \alpha, \quad y_{N+1} = \beta. \end{aligned}$$

This is a system of N equations for the N unknowns $y_1, y_2, \ldots, y_N$, which approximate the exact solution values, $y(x_1), y(x_2), \ldots, y(x_N)$. If we denote the system by $F(y) = 0$, then the equations are (note how the boundary values enter)

$$\begin{aligned} F_1(y) &= \frac{\alpha - 2y_1 + y_2}{\Delta x^2} - f(x_1, y_1), \\ F_i(y) &= \frac{y_{i-1} - 2y_i + y_{i+1}}{\Delta x^2} - f(x_i, y_i), \qquad i = 1, \ldots N, \\ F_N(y) &= \frac{y_{N-1} - 2y_N + \beta}{\Delta x^2} - f(x_N, y_N). \end{aligned}$$

In the *nonlinear case*, one would have to solve $F(y) = 0$ iteratively, using Newton's method. However, trying to keep things simple, we will only study *linear problems* here, so as to avoid iterative equation solving.

In fact, we are only going to consider the very simplest linear problems, where $f(x, y) \equiv f(x)$. This means that the right-hand side of the BVP is independent of y. In this case, we see that our system $F(y) = 0$ reduces to a linear system of equations,

$$\frac{1}{\Delta x^2}\begin{pmatrix} -2 & 1 & 0 & \dots & \\ 1 & -2 & 1 & & \\ & 1 & -2 & 1 & \\ & & & \ddots & \\ & \dots & 0 & 1 & -2 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \\ y_N \end{pmatrix} = \begin{pmatrix} -\alpha/\Delta x^2 + f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ -\beta/\Delta x^2 + f(x_N) \end{pmatrix}$$

This is a tridiagonal linear system. The matrix is sparse. Solving the system has low complexity. The LU decomposition runs in $\mathrm{O}(N)$ time ($3N$ operations, to be precise; compare to the $N^3/3$ operations needed for a dense, full matrix). The forward and back substitutions also run in $\mathrm{O}(N)$ time (a total of $2N$ operations). Therefore, the solution effort is moderate even when N is large, and is proportional to the number of grid points.

Task 1.1 Write a MATLAB 2pBVP solver

```
function y = twopBVP(fvec, alpha, beta, L, N)
```

that solves $y'' = f(x)$ with boundary conditions $y(0) = \alpha$ and $y(L) = \beta$ on an equidistant grid having N *interior points*. Test your solver by implementing a right-hand side function of your choice, so that you know what the exact solution is. It is *necessary to verify that the code functions properly in every respect, even when changing the boundary conditions.*

Note that you cannot take a "too simple" right hand side. For example, if you take $f(x) \equiv 1$, the solution is a 2nd degree polynomial, and a 2nd order method therefore solves the problem exactly. To choose a more difficult problem, select the function y itself, and differentiate twice to find the corresponding f. Don't forget that boundary conditions must match. For your chosen problem, **verify that your method works by plotting the error in a log-log diagram in the usual way to demonstrate second order convergence**. Give all details about your computational setup.

Here are some useful hints for constructing your solver:

1. To simplify your work with Task 1.2 let your solver take `fvec` as an input *vector*. That is, before calling `twopBVP`, evaluate the right hand

side function f of the 2pBVP at the N interior grid points $x_1, x_2, \ldots, x_N$ and store the results in `fvec`.

2. Do not plot the result from within `twopBVP` as you will use this solver repeatedly below. Instead, plot the result by giving a plot command in the script (main program) you use to call `twopBVP`.

3. Note that when you plot the solution, you have obtained *y only on interior points*. Make sure to *append the boundary values* at the beginning and end of your solution vector, so that you can plot the solution *all the way from boundary to boundary*.

4. In MATLAB and `scipy.linalg` you will find the function `toeplitz`, which generates tridiagonal matrices of the structure you need. Optionally, you can use the command `diag`, in the following manner. If you have an $N-1$ vector `sup`, an $N-1$ vector `sub` and an N vector `main`, then you can construct a tridiagonal $N \times N$ matrix with subdiagonal `sub`, main diagonal `main` and superdiagonal `sup` by the command

   ```
   A = diag(sub,-1) + diag(main,0) + diag(sup,1);
   ```

   Using this technique, all you need to do is to first generate the three vectors `sub, main, sup`, and install them in their right positions in the matrix. After constructing the matrix, using either technique, you correct the elements in the first and last row of A, if necessary, so that the boundary conditions are properly represented. The commands `toeplitz` and `diag` are very convenient as they allow you to avoid using `for`-loops and instead work with a vectorized code. However these commands have the drawback that you work with full matrices without exploiting the sparse tridiagonal structure. For higher performance, you can check MATLAB's `help` for the commands `sparse` and `spdiags` or the Python module `scipy.sparse`, in particular the `diags` function. Thus, using the sparsity of the tridiagonal matrices, you can choose a matrix representation that makes your solver much faster, and also enables you to use very large values of N. (A better alternative in high precision computations is of course to work with higher order methods, but higher order methods also have sparse representations that should be exploited.)

The Beam Equation. An elastic beam under load is deflected in accordance with its material properties and the applied load. According to elas-

ticity theory, the deflection u is governed by the differential equations

$$\begin{aligned} M'' &= q(x) \\ u'' &= M(x)/(EI), \end{aligned}$$

where $q(x)$ is the load density (N/m); $M(x)$ is the bending moment (Nm); E is Young's modulus of elasticity (N/m^2); I is the beam's cross-section moment of inertia (m^4); and u is the beam's centerline deflection (m).

The beam is supported at its ends, at $x = 0$ and $x = L$. This means that there is no deflection there: $u(0) = u(L) = 0$. Further, assuming that the beam's ends do not sustain any bending moment, we also have the boundary conditions $M(0) = M(L) = 0$. We have then obtained a 2pBVP, of order 4. The task is to solve for the deflection u. Thus, given the load vector q, you first solve a 2pBVP for the bending moment M; then, using your calculated M vector as data, you solve *another* 2pBVP for the deflection u.

Task 1.2 Use your 2pBVP solver to write a script for solving the beam equation for a heavy-duty railway flatcar. Its support beams have length $L = 10$ m and elasticity module $E = 1.9 \cdot 10^{11}$ N/m^2 (construction steel). The beam's web is "taller" at the center section. This is modeled by a cross-section moment of inertia that varies along the beam, according to

$$I(x) = 10^{-3} \cdot \left(3 - 2\cos^{12}\frac{\pi x}{L}\right).$$

Solve the problem for this beam shape, with a load of $q(x) = -50$ kN/m (i.e., a load of five metric tonnes per meter), and **plot the computed deflection**. Don't forget to insert the boundary conditions so that your plot reaches all the way out to the boundary. Please be careful with your SI units – steel is a strong material, so loads are on the order of kN, and deflections on the order of mm. Make sure that your computations are consistent, so that you don't get incorrect results due to mistakes in the choice of units.

Specifically, **make a run for $N = 999$ interior grid points** (corresponding to $\Delta x = 10^{-2}$) and **compute the deflection at the beam's midpoint**. Give your result to eight digits. (This will be used to check that your code is correct.)

Part 2. Sturm–Liouville eigenvalue problems

Theory and problem statement. A fascinating aspect of applied mathematics is that the very same equation may solve problems in the most diverse areas. The Sturm–Liouville problem does just that, and and has applications

in e.g. materials science (oscillations in beams; buckling modes in structural analysis) as well as in microphysics (quantum mechanics).

Sturm–Liouville problems are eigenvalue problems for differential operators. The standard formulation is

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}y}{\mathrm{d}x}\right) - q(x)y = \lambda y$$

$$y(a) = 0; \qquad y(b) = 0$$

where $p(x) > 0$ and $q(x) \geq 0$. The problem is to find *eigenvalues* λ and *eigenfunctions* $y(x)$ satisfying this 2pBVP. Eigenfunctions are often called "eigenmodes" or just "modes," because they can be interpreted as oscillation modes of various frequencies. In quantum mechanics the eigenmodes are called "wave functions," and the eigenvalues represent "energy levels" of the corresponding "states."

Note that the boundary conditions in a Sturm–Liouville problem are almost always homogeneous; apart from the Dirichlet condition above, one also encounters homogeneous Neumann conditions such as $y'(a) = 0$.

Discretization. The Sturm–Liouville problem is discretized by standard symmetric 2nd order finite differences to obtain a matrix eigenvalue problem

$$T_{\Delta x} y = \lambda_{\Delta x} y$$

where $T_{\Delta x}$ is a symmetric, tridiagonal $N \times N$ matrix, which depends on the functions p, q and Δx. The discretization has exactly N *real* eigenvalues $\lambda_{\Delta x} = \lambda + O(\Delta x^2)$, due to symmetry. This reflects the fact that the differential operator is self–adjoint and therefore possesses an infinite sequence of real eigenvalues and orthogonal eigenfunctions.

You generate the matrix using the `diag` command as described before. The eigenvalues of a matrix A are computed in MATLAB using the command `eig(A)`. Depending on how you call this function, you can compute either eigenvalues only, or eigenvalues and the corresponding eigenvectors. The latter are discrete versions of the eigenfunctions, and can be plotted to visualize the eigenfunctions. Use `help` in MATLAB to find out more, and *take care with how the outputs are ordered.* In Python, the equivalent `eig` function resides in `scipy.linalg`.

Task 2.1 Construct a Sturm–Liouville solver for the simple problem

$$u'' = \lambda u$$

with boundary conditions $u(0) = u(1) = 0$. Solve the problem analytically (find the eigenvalues of $\mathrm{d}^2/\mathrm{d}x^2$). Then take your numerical solver and solve the same problem, verifying that it is second order accurate by **plotting the error $\lambda_{\Delta x} - \lambda$ for the first three eigenvalues versus** N, the number of interior grid points. Make sure that the error in all three eigenvalues has the correct slope in a loglog diagram. Specifically, **give the eigenvalues you obtain to eight digits** for $N = 499$, corresponding to $\Delta x = 2 \cdot 10^{-3}$.

In addition, make a separate graph where you **plot the first three eigenmodes**. Note that by the "first" eigenvalues, we refer to those of smallest absolute magnitude. Make sure that you plot the eigenfunction *all the way to the boundaries, at* $x = 0$ *and* $x = 1$. Include the boundary points.

The Schrödinger equation. A quantum particle trapped in a one-dimensional potential $V(x) \geq 0$ has a stationary *wave function* $\psi(x)$ satisfying the stationary *Schrödinger equation*

$$\psi'' - V(x)\psi = -E\psi,$$

where E is the *energy level* of the particle. Assuming that the potential is infinite outside the interval $[0, 1]$, the boundary conditions are $\psi(0) = \psi(1) = 0$. This is a Sturm–Liouville problem for determining the wave functions and energy levels. In addition to the wave functions, the corresponding *probability densities* $|\psi|^2$ are of interest; they give the probability of finding the particle at a certain position x.

Task 2.2 Construct a solver for the stationary Schrödinger equation for any given potential $V(x)$. The solver should **compute the first few eigenvalues** (say up to six to ten eigenvalues), and **plot both the wave functions ψ and the probability densities $|\psi|^2$**. Use separate plots for wave functions and probability densities.

Note that wave functions and probability densities need to be normalized in order to make good plots. Use the same normalization for ψ and $|\psi|^2$ and let $\hat{\psi}$ denote the normalized wave function. As is customary in physics, it is a good idea to plot wave functions and probability densities "at the energy level." This is achieved by plotting the functions $\hat{\psi}_k(x)+E_k$ and $|\hat{\psi}_k(x)|^2+E_k$. This trick "lifts" the wave function (probability density) from zero to the energy level. To make nice plots, you may have to carefully consider how to scale the amplitude of wave functions. Any amplitude normalization is permissible, since the eigenvalue problem is homogeneous. You should choose the normalization from the point of view of visualizing the eigenfunctions and the probability densities as clearly as possible.

Test your code by trying it out for the case $V(x) = 0$ in the interval $[0, 1]$ and $V(x) = \infty$ outside the interval. This is usually referred to as a particle in a potential box. Since the potential is infinite outside the interval, the probability of finding the particle there is zero. This means that $\psi(0) = \psi(1) = 0$, and that the Schrödinger equation reduces to the simple Sturm–Liouville problem

$$\psi'' = -E\psi.$$

As we have used this test case in task 2.1, you just need to verify that the Schrödinger solver works correctly.

Now the task is to compute something you wouldn't be able to do by hand. Thus, other choices of $V(x)$ lead to the study of "potential barriers" and phenomena like tunneling. Modify your Schrödinger code to **calculate wave functions and probability densities for potentials that are nonzero in** $[0, 1]$ **and infinite outside**. For example, one can try the potential barriers $V(x) = 700(0.5 - |x - 0.5|)$ and $V(x) = 800 \sin^2 \pi x$. At what energy levels does the probability to find the particle at $x = 0.5$ become significant? Here you see the well-known effect of getting a "doublet state," i.e., two states of almost the same energy (but different parity) due to the potential barrier inside the box. There are of course more interesting potentials. Can you define a potential with three "wells" so that you get triplet states? Experiment with your code – it will solve any problem, and you can apply it to any potential of your choice. Report at least one interesting case you have constructed yourself.

Notes concerning the report. The same rules apply as in Project 1. 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. Document your discretizations and supply relevant code segments, functions or scripts. Clarify your observations and conclusions. Include references, acknowledge discussions with fellow students or instructors, and sum up what you have learned from this assignment.