01_fdm_poisson_1d

November 9, 2016

1 Homework 1

1.1 General Instructions

- To pass this assignment you are asked to submit both the program sources and a short report including results, discussion etc.
- For the theoretical exercises, please include intermediate steps to explain how you arrive at your solution.
- For the computational exercises, you can use either Matlab, Octave (an open source variant of Matlab) or Python to implement your computer programs.
- Please submit the **complete** computer program or script you used to solve the computational problems. Don't overengineer your code, keep it as simple and readable as possible and provide short code comments to help other people understanding your code.
- Please provide also a short summary and discussion of your results including the requested output (e.g. tables, graphs etc).
- You have two options to submit you solution. Either you submit the program source and the report in a separate pdf file or you can include everything in a single jupyter notebook, preferable based on the original homework notebook.
- Up to 3 students can jointly submit the solutions. **Only 1 student from each group** is supposed to submit them. Please indicate the other members of your group in the comment field appearing when you submit your files in Cambro.
- Deadline for submission of your solutions is **19th of November**.

Happy coding!

1.2 Exercise 1

Find at least 4 more "famous" partial differential equations (PDE) with one in each category "Linear PDE, Non-linear PDE, Linear System, Non-linear system". Give a brief description of the underlying phenomena modeled by the PDE.

1.3 Problem 2

In Lecture 2, we introduced the **central finite difference operators**

$$\partial^0 u(x) = \frac{u(x+h) - u(x-h)}{2h} \approx u'(x)$$

and

$$\partial^+ \partial^- u(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \approx u''(x)$$

as an approximation of the first and the second order derivative $u'(x)$ and $u''(x)$, respectively.

Recall that for $u \in C^k([0,1])$, the Taylor expansion of u around x is given by

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2!}u''(x) + \ldots + \frac{h^{k-1}}{(k-1)!}u^{(k-1)}(x) + \frac{h^k}{k!}u^{(k)}(\xi)$$

for some $\xi \in (x, x+h)$. Since $u \in C^k([0,1])$, the remainder term $\frac{h^k}{k!}u^{(k)}(\xi)$ is uniformly bounded with respect to ξ and thus we can simply write

$$u(x+h) = u(x) + hu'(x) + \frac{h^2}{2!}u''(x) + \ldots + \frac{h^{k-1}}{(k-1)!}u^{(k-1)}(x) + \mathcal{O}(h^k)$$

a) Use Taylor expansion to show that for $u \in C^3([0,1])$

$$\|\partial^0 u(x) - u'(x)\|_{C([0,1])} \leqslant Ch^2 \|u^{(3)}\|_{C([0,1])}$$

b) Similarly, demonstrate that

$$\|\partial^+ \partial^- u(x) - u''(x)\|_{C([0,1])} \leqslant Ch^2 \|u^{(4)}\|_{C([0,1])}$$

assuming that $u \in C^4([0,1])$.

1.4 Computational Problem 1

In this problem set you are asked to solve the Poisson problem

$$-u'' = f \quad \text{in } (0,1)$$

numerically for various types of boundary conditions.

a) Start with implementing the finite difference method (FDM) from Lecture 2 using the right-hand side

$$f = \sin(2\pi x)$$

and boundary conditions

$$u(0) = u(1) = 0.$$

Plot the solutions for different mesh sizes $h = 1/N$ with $N = 4, 8, 16, 32, 64$ in the *same* plot. Find the *exact* analytical solution u to the given Poisson problem (Hint: it should be very similar to f) and plot it for $N = 64$ into the same figure. Does your computed discrete solution U converge to u?

b) Next, we switch to a Neumann boundary condition on the left endpoint; that is

$$-u'(0) = \sigma_0, \quad u(1) = 0.$$

Modify your FDM solver to incoporate the Neumann condition based on the one-side apprioximation

$$-u'(0) \approx \frac{u(0) - u(h)}{h} = \sigma.$$

Take the exact solution u from a) and calculate its proper value for σ. Conduct a similar numerical study as in part a). What do you observe regarding the accuracy of the method?

c) Now try to solve the same Poisson problem, now with boundary conditions

$$-u'(0) = \sigma_0, \quad u'(1) = \sigma_1$$

using values σ_0 and σ_1 corresponding to the exact solution u in part a). What happens when you try to solve the linear algebra system? Why?

1.4.1 Useful code snippets

As most of you are familiar with MATLAB but not so familar with Python, we provide a number of code snippets to get you started in Python. Three dots ... indicate places where you have to fill in code. Note that this outline provides only a very rudimentary inefficient implementation to begin with and we will refine our methods while progressing towards more advanced and larger problems.

We start with importing the necessary scientific libraries and define a name alias for them.

```
In [3]: # Arrary and stuff
        import numpy as np
        # Linear algebra solvers from scipy
        import scipy.linalg as la
        # Basic plotting routines from the matplotlib library
        import matplotlib.pyplot as plt
```

Next we define the grid points.

```
In [4]: # Number of equally spaced subintervals
        N = 4
        # Mesh size
        h = 1/N #Important! In Python 2 you needed to write 1.0 to prevent integer
        # Define N+1 grid points via linspace which is part of numpy now aliased as
        x = np.linspace(0,1,N+1)
        print(x)

[ 0.    0.25  0.5   0.75  1.  ]
```

Now define matrix A and right-hand side vector F. We will first fill in the values that will be unchanged for different boundary conditions.

```
In [93]: # Define a (full) matrix filled with 0s.
         A = np.zeros((N+1, N+1))

         # Define tridiagonal part of A by for rows 1 to N-1
         for i in range(1, N):
```

```
        A[i, i-1] = ...
        A[i, i+1] = ...
        A[i, i] = ...

    # Define right hand side. Instead of iterating we
    # use a vectorized variant to evaluate f on all grid points
    # Look out for the right h factors!
    F = ...*np.sin(2*np.pi*x)

    # Note that F[0] and F[N] are also filled!
```

Last step to set up the system is to take the boundary conditions into account by modifying A and F properly.

```
In [ ]: # Left boundary
        A[0,0] = ...
        F[0] = ...

        # Right boundary
        A[N,N] = ...
        F[N] = ...
```

Now we solve the linear algebra system $AU = F$ and plot the results.

```
In [ ]: U = la.solve(A, F)
        #  "x-r" means mark data points as "x", connect them by a line and use red
        plt.plot(x, U, "x-r")
```

With these snippets in place you should be able to solve Computer Problem 1 but don't hesitate to ask if you are wondering about something!

1.5 Computational Problem 2

The goal of this problem is to investigate the numerical error introduced by the FDM more quantitatively and to familiarize us with the **method of manufactured solution**.

The idea is to assess the accurracy and correctness of a PDE solver implementation by constructing a know reference solution which solves the PDE problem at hand. This can be simply done by picking a meaningful and not to boring analytical solution and explicitly calculate the data which need to be supplied, e.g., the right-hand side or boundary values for various boundary condition.

For instance, taking the function $u(x) = x + \sin(2\pi x)$, we can simply calculate that

$$u'(x) = 1 + 2\pi\cos(2\pi x) \tag{1}$$

$$u''(x) = -(2\pi)^2 \sin(2\pi x) \tag{2}$$

and thus u satisfies the Poisson problem

$$-u''(x) = (2\pi)^2 \sin(2\pi x)$$

with boundary conditions

$$-u'(0) = -(1+2\pi), \quad u(1) = 1.$$

With a known reference solution at hand we can compute the error vector $u_i - U_i$ at the grid points $\{x_i\}_{i=0}^N$ for a series of successively refined grid, e.g. by taking $N = 4 \cdot 2^k$ for $k = 0, 1, 2, 3$ etc.

Reducing the mesh size h by half allows us to easily compute the **experimental order of convergence (EOC)**, that is the observed error reduction in the numerical solution when passing from a coarser mesh with mesh size h to to a finer mesh with $h/2$. The EOC can then be compare with the theoretically predicted error reduction (if known).

For instance, if you know that the discretization error $E(h)$ given on mesh with mesh size h and measured in some norm $|\cdot|$ behaves like $\|E(h)\| \sim h^k$, you can conclude that

$$\frac{\|E(h/2)\|}{\|E(h)\|} \sim \frac{(h/2)^k}{h^k} = (1/2)^k$$

when passing from h to half the mesh size $h/2$. Taking the logarithm of the last equation shows

$$k \sim \frac{\ln(\|E(h)\|/\|E(h/2)\|}{\ln 2}.$$

(Verify this!) Thus the EOC is measured by the

$$EOC = \frac{\ln(\|E(h)\|/\|E(h/2)\|}{\ln 2}.$$

Alternatively, you can do a log-log plot of your error as function of h, that is plot $\ln(\|E(h)\|)$ againts $\ln(h)$. Then the slope of this plot should be $\sim k$ if we expect the method to be of convergence order k

a) Use this approach to verify your FDM program developed in Computational Problem 1 a) by computing the error for $N = 4, 8, 16, 32, 64$ in the maximum norm. Give the corresponding log-log plot and report your convergence order. Do you achieve 2nd order convergence?

b) Next, repeat the same experiment for the Poisson problem 1b) with mixed Dirchlet/Neumann boundary conditions. What EOC do you observe? Can you explain it?

1.6 Computational Problem 3

In the final computer exercise you are asked to extend your FDM solver in order to compute a solution to the *Convection-Diffusion problem*

$$-\epsilon u''(x) + bu'(x) = f(x) \quad \text{for } x \in (0,1), u(0) = u(1) = 0,$$

with $b = 1$ and various ϵ tending 0. While for $\epsilon > 0$, the problem is clearly a 2nd order problem, its characteristics change drastically for $\epsilon \to 0$. Formally, the limit equation is given by the **first order** problem

$$bu'(x) = f(x) \quad \text{for } x \in (0,1)$$

and we see immediately that only *one* boundary condition should be required in the limit case. (Convince yourself by assuming that $f = 1$ and trying to compute a solution). It turns out that it is natural to impose a Dirichlet boundary condition $u(0) = u_0$ only at the "inflow point" $x(0)$ and thus the "outflow point" $u(1) = u_1$ becomes "superfluous" when $\epsilon \to 0$. Here, we will study what happens to our FDM solver when we gradually approache this limit case.

a) Compute f such that

$$u(x) = x - \frac{e^{(x-1)/\epsilon} - e^{-1/\epsilon}}{1 - e^{-1/\epsilon}} \tag{3}$$

is an exact solution for $b = 1$ and arbitrary ϵ (Hint: f should not look too complicated…).
b) Start with using the symmetric/central difference operator

$$\partial^0 U_i = \frac{U_{i+1} - U_{i-1}}{2h}$$

to discretize the first order derivative $bu'(x)$. How does the resulting matrix system look like? Adapt your FDM solver from Problem 1 accordingly and verify your implementation employing the method of manufactured solution from Problem 2.

c) Now repeat the numerical experiment and compute a numerical solution U_ϵ for $\epsilon = 0.1, 0.01.0.001$ and at least 4 successively refined grids. For each ϵ provide a plot including the exact solution and the computed approximations. What do you observe?

Report the EOC, this time in a 2 column table with the first column reporting the mesh size and the second column reporting the computed error. Is the error reduced by a factor 4 as expected from a 2nd order convergent method? Can you recover order 2 by making the meshes even finer?

d) Finally, again, conduct the same experiment after replacing ∂^0 by 1) ∂^+ and 2) ∂^-. Describe your observations of the discrete solution behavior. Which variant gives the most satisfying/robust solution for small ϵ?

Finally compute again the EOC. What do you get?

The following cell loads non-default styles for the notebook

```
In [2]: from IPython.core.display import HTML
        def css_styling():
            #styles = open("../styles/custom.css", "r").read()
            styles = open("../styles/numericalmoocstyle.css", "r").read()
            return HTML(styles)

        # Comment out next line and execute this cell to restore the default notebo
        css_styling()

Out[2]: <IPython.core.display.HTML object>
```