Department of Engineering Mathematics

EMAT30008: SCIENTIFIC COMPUTING
Part 3: Finite Difference Methods and PDEs

1 Overall aim

Remember that the overall aim of the coursework for this unit is to integrate the work you do from each week into a single piece of software. The software should be a general numerical continuation code that can track, under variations of a parameter in the system,

- limit cycle oscillations of arbitrary ordinary differential equations (of any number of dimensions), and
- steady-states of second-order diffusive PDEs.

Moreover, you should provide time simulation codes for both ODEs and PDEs to enable the solutions to be verified.

2 Steps to get there

It's best to start with the time simulation code, and work up from there. Start slowly, change one thing at a time, and keep your code as modular as possible.

Some suggestions for dividing the overall task into manageable pieces are given below. You could aim to complete items 1-3 by the end of week 21, item 4 by the end of week 22, and then focus on items 5 and 6 after that. You should be able to work on item 7 throughout!

If it helps, please feel free to team up and share ideas (see, e.g., `https://en.wikipedia.org/wiki/Pair_programming`), for items 1–3.

1. First, start off by playing with the Python code for the forward Euler scheme, as available on Blackboard. You should experiment with different discretisations (choices of the mesh parameters `mx` and `mt`) as well as different problem parameters (L, T, and κ), so you're confident you know how it works. Use the code to verify the stability criterion for yourself.

2. What is the steady state of the system in the code, and does it make sense to you? What happens if we change the initial condition — for example, to $u(x,0) = \sin^p(\pi x)$ for some integer p?

3. Modify the Python code to replace the component-wise forward Euler method with the matrix/vector approach (see page 15 of the lecture notes). Make sure the solutions you obtain are the same as from the component-wise method.

4. Modify the code to use backward Euler, and Crank-Nicholson (nothing complicated; just ordinary linear solve will do), and check that you get the same solutions but with no stability problems.

5. Adapt the code so that it can deal with more general parabolic PDE problems. Some extensions that you might wish to consider (there's no need to do them all!):

- non-homogenous Dirichlet boundary conditions, so (for example) $u(0,t) = u_0(t)$
- homogenous (or not) Neumann boundary conditions, so (for example) $\dfrac{\partial u}{\partial x}(0,t) = 0$,
- more general boundary conditions, e.g. a Robin (mixed) boundary condition $\alpha u + \beta \dfrac{\partial u}{\partial x} = g$ at $x = 0, L$, or periodic boundary conditions $u(0,t) = u(L,t)$ for all t,
- heat sources inside the domain, so the PDE becomes $\dfrac{\partial u}{\partial t} = \kappa \dfrac{\partial^2 u}{\partial x^2} + f(x,t)$,
- non-constant diffusion coefficient, so the PDE becomes $\dfrac{\partial u}{\partial t} = \dfrac{\partial}{\partial x}\left[\kappa(x)\dfrac{\partial u}{\partial x}\right]$,
- other parabolic PDE problems, whether linear or nonlinear; e.g.,
  - the cable equation $C_m \dfrac{\partial u}{\partial t} = g_m \dfrac{\partial^2 u}{\partial x^2} + g_L(u_L - u)$, where $C_m > 0$, $g_M > 0$, $g_L > 0$, and u_L are constants
  - Fisher's equation $\dfrac{\partial u}{\partial t} = \kappa \dfrac{\partial^2 u}{\partial x^2} + ru(1-u)$ where $\kappa > 0$ and $r > 0$ are constants.

6. Adapt your numerical continuation code (from the ODEs part of the course) so that it can also track, under variations of a parameter in the system, steady-states of second-order diffusive PDEs.

7. Turn the code into *good* Python code, that you can call to solve a parabolic PDE problem. You should be applying sound software engineering principles, as for the ODE part of the course, throughout.

   You could, for example,

   - vectorize the code, rather than using loops,
   - break it down into subroutines/internal function calls,
   - create a function library,
   - improve the computational efficiency of the linear algebra; e.g., linear solve for a tridiagonal matrix, sparse matrix operations, etc.
   - add visualisation functionality; e.g., plotting, snapshots, animation, diagnostics, etc.,
   - make the design easily adaptable for future modifications,
   - make sure the code is fully documented.

   It would also be a good idea to check your code is working by testing against known (analytic) solutions wherever possible. Some information on explicit solutions of the 1D heat equation (and other PDEs) can be found here: `https://people.maths.bris.ac.uk/~mazvs/supplementary.pdf`.