School of international Liberal Studies
Waseda University
Advanced Course: Virtual Earth
Spring 2025

Problem Set 3

1. The diffusion of heat or a chemical tracer such as salt is governed by the partial differential equation $\partial T/\partial t = \kappa \partial^2 T/\partial x^2$. The 2nd order differential operator $\partial^2/\partial x^2$ on the right hand side of this equation (in two dimensions it would be $\partial^2/\partial x^2 + \partial^2/\partial y^2$ and in three $\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$), sometimes called the "diffusion operator", is one of the most important objects in math, science and engineering as it shows up again and again in many contexts (even when they have nothing to do with diffusion). We saw in class that the spatially discretized form of this operator can be written as a "sparse" matrix with "-2" on the diagonal and "+1" on either side. It is no exaggeration to say that this matrix–which we'll call the "diffusion matrix"–may easily be one of the most important matrices in the universe! Write a function, `DiffusionMatrix` to construct a "diffusion matrix" of size $n \times n$. That is, `A=DiffusionMatrix(n)` takes an integer n as input and returns a diffusion matrix `A` of size $n \times n$. One way to do this is through a nested `for` loop. But a far more efficient way would be to use builtin functions for constructing and manipulating sparse matrices (in Matlab these are called `sparse` and `spdiags`, but there are likely equivalent ones in NumPy and SciPy).

2. Consider the diffusion of heat in a metal bar of length L with temperature at one end ($x = 0$) held at 0°C and the other ($x = L$) at 10°C. At time $t = 0$ the distribution of temperature along the bar is given by: $T(x, 0) = 0$ for $0 \leq x \leq L/2$ and $T(x, 0) = 10$ for $L/2 < x \leq L$.

   (a) Use finite differences to discretize the spatial derivatives with a grid spacing of Δx and convert the PDE into a system of coupled ODEs.

   (b) Using your spiffy new function `DiffusionMatrix`, write a script to numerically integrate this coupled system in time with the forward Euler scheme. Integrate the system from $t = 0$ to $t = 200$ seconds. For this, take $L = 1$ m, $\kappa = 0.001$ m^{2}s^{-1}, and $\Delta x = 0.05$. Pick a time step Δt such that $r \equiv \kappa \Delta t/(\Delta x)^2 < 0.5$. Plot the numerical solution at 20 s intervals to show how it evolves in time. Now repeat the computation with a larger Δt (or smaller Δx) such that $r > 0.5$. Notice a difference? The parameter r is fundamental in numerical analysis as it puts constraints on the values of Δx and Δt for which a numerical scheme is *stable*. This is known as the "CFL condition", named after Courant, Friedrichs and Lewy who came up with this idea in the 1920s (`https://en.wikipedia.org/wiki/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition`), and the value of r at which

a scheme becomes unstable is known as the "CFL limit". While "explicit" schemes such as forward Euler are cheap and easy, their big drawback is that they are subject to the CFL limit, which means that for a given grid spacing Δx your time step cannot exceed a certain value. "So what?", you might ask. Well, the larger the time step the quicker you can compute the solution. So, generally speaking, you want to take as large a time step as you can get away with.

(c) Now solve the same equation using the backward Euler method and plot the solution at 20 s intervals. Recall from the class notes that this requires solving, at each time step, a linear system of equations with a different right hand side (but the same coefficient matrix). If you used Gauss elimination, you'd have to start from scratch each time. But by first LU-factorizing the coefficient matrix, you can greatly reduce the number of calculations needed to solve the linear system. Almost all codes for solving the diffusion equation exploit this so make sure to incorporate it into your implementation. The Matlab function for LU factorization is: `lu`, i.e., `[L,U]=lu(A)` will compute the LU factorization of matrix `A`. To solve a linear system of equations use Matlab's "backslash" (\) operator: `x=A\b` will calculate the solution `x` to `A*x=b`. There is some very sophisticated math, numerical analysis and computer science behind this operator. There are no doubt equivalent functions available in NumPy and SciPy if you use Python.

Try a few different values of Δt, in particular those for which the forward Euler solution was unstable (i.e., $r > 0.5$). You'll notice that the solution doesn't blow up even for large Δt. That is the great advantage of "implicit" schemes such as backward Euler: even though they are more expensive (requiring a linear solve at each time step) they are generally not subject to the CFL limit, which means you can take much larger time steps. But sadly there's no such thing as a free lunch. The larger the time step the less accurate your numerical solution will be, and if you take too big a time step the solution will be rubbish. That is the basic tradeoff of numerical computation.