1-D Ammonium Diffusion Code

Trevor R Hillebrand

created December 15, 2017

This document describes the code `ammonium_diffusion_1D.py`, which I wrote on December 14, 2017 to model the diffusion of chemical species in marine sediments. The chemical species in question here is the ammonium ion, but can be modified for use with any species. The code is written in Python, and makes heavy use of the NumPy package, which enables us to solve linear systems of equations of the form Ax=b. The code uses a finite difference discretization in space, and Crank-Nicolson time-stepping. As of the time of writing, the code only handles Dirichlet (fixed value) boundary conditions. With a little work, it could also handle Neumann (fixed flux) boundary conditions, or even time-varying boundary conditions.

$$\frac{\partial u}{\partial t} - D\frac{\partial^2 u}{\partial x^2} = f(x,t) \quad \text{for } t > 0, x \in (a,b) \tag{1}$$

The initial condition is $u(x,0) = u_0$, and the solution is subject to Dirichlet boundary conditions $u(0,t) = \alpha, u(N,t) = \beta$. Here, we discretize in space and time separately. First, use the centered second-order finite difference approximation in space:

$$\frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{h^2} \approx \frac{\partial^2 u}{\partial x^2},$$

from which we get the second-order finite difference matrix $\mathbf{A}_{fd}$. Since we have Dirichlet boundary conditions, the first and last rows of $\mathbf{A}_{fd}$ should have one and only one non-zero entry, and it should be 1 in the first and last columns, respectively:

$$\mathbf{A}_{fd} = \frac{1}{\Delta x^2}\begin{bmatrix} \Delta x^2 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & 0 & \Delta x^2 \end{bmatrix}$$

Our equation (1) now becomes

$$\frac{\partial \mathbf{u}}{\partial t} - D\mathbf{A}_{fd}\mathbf{u} = \mathbf{f}(x,t) \quad \text{for } t > 0, x \in (a,b) \tag{2}$$

So, the finite difference matrix $\mathbf{A}_{fd}$ is the numerical approximation of the second partial derivative in x. Here I'm using boldfaced text to indicate an array (but I might forget to embolden (?) some things in the future). $\mathbf{A}_{fd}$ is a matrix of size NxN. $\mathbf{u}$ and $\mathbf{f}$(x,t) is an Nx1 column vector. The diffusivity D is left as a scalar here, for simplicity, but it could conceivably vary with depth as well. It's nice when the source term $\mathbf{f}$(x,t) = 0, but we will keep it in here while we work out all the math.

So now we have a system of linear equations for our PDE, but we still have to get rid of the partial time derivative. We use the Crank-Nicolson time integration scheme, which is just a special case of the θ-scheme:

$$\frac{\mathbf{u}^{k+1} - \mathbf{u}^k}{\Delta t} = -D\mathbf{A}_{fd}(\theta\mathbf{u}^{k+1} + (1-\theta)\mathbf{u}^k) + \theta\mathbf{f}^{k+1} + (1-\theta)\mathbf{f}^k$$

when $\theta = \frac{1}{2}$.
The superscripts here indicate the time step. $\mathbf{u}^{k+1}$ is our unknown; we know $\mathbf{u}^k$ at each time step because we have the initial condition $\mathbf{u}^0$. Rearrange the equation, with all the unknowns $\mathbf{u}^{k+1}$ on the left, and the known $\mathbf{u}^k$ on the right:

$$\mathbf{u}^{k+1} + \Delta t D\mathbf{A}_{fd}\theta\mathbf{u}^{k+1} = \mathbf{u}^k - \Delta t D\mathbf{A}_{fd}\left(1-\theta\right)\mathbf{u}^k + \Delta t\left(\theta\mathbf{f}^{k+1} + (1-\theta)\mathbf{f}^k\right)$$

or

$$\left(\mathbf{I} + \theta\Delta t D\mathbf{A}_{fd}\right)\mathbf{u}^{k+1} = \left(\mathbf{I} - \Delta t D(1-\theta)\mathbf{A}_{fd}\right)\mathbf{u}^k + \Delta t\left(\theta\mathbf{f}^{k+1} + (1-\theta)\mathbf{f}^k\right), \tag{3}$$

where the matrix $\mathbf{I}$ is the identity matrix. This looks ugly, but it's just an equation of the form $\mathbf{Ax}$=$\mathbf{b}$, which we can make Python solve for us:

```
for tt in range(1, int(end_time/dt)):
      u_xt[:,tt] = np.linalg.solve((np.identity(N) + theta*dt*D*A_fd), \
      np.matmul((np.identity(N)-dt*(1-theta)*D*A_fd), u_xt[:,tt-1]) \
      + dt*(theta*f[:, tt] + (1-theta)*f[:,tt-1]))
```

As I said above, this algorithm is easy to start because we already have the initial condition, $\mathbf{u}^0$. Crank-Nicolson time-stepping makes it both quite accurate and unconditionally stable. This means you can use any time step you want, but you'll get much better accuracy with smaller time steps. If $\theta \neq 1/2$ the global error of the numerical behaves like $\mathcal{O}(\Delta t + \Delta x^2)$; if $\theta = 1/2$ the global error of the numerical behaves like $\mathcal{O}(\Delta t^2 + \Delta x^2)$. So Crank-Nicolson ($\theta = 1/2$) time-stepping gives you an extra order of accuracy. You'll have to play with it the trade off between time step and accuracy.