

A Simple Numerical Model in Python: Discretization

Daniel Dauhajre, UCLA 2016

We show Python's capability of solving PDEs through a simple 1D diffusion problem. It is thought of as vertical diffusion(mixing) of a velocity $v(t, z)$ with a vertical diffusion (mixing) coefficient $\kappa_v(t, z)$ with the option to provide a forcing $F(t, z)$. (the code is written relative to these variables because of a specific problem I was working on; you could easily think of v as a concentration C with a diffusion coefficient D in place of κ_v).

$$\frac{\partial v}{\partial t} = \frac{\partial}{\partial z} \left(\kappa_v \frac{\partial v}{\partial z} \right) + F(t, z) \quad (1)$$

The vertical mixing coefficient $\kappa_v(t, z)$ and forcing $F(t, z)$ may be prescribed as having vertical and/or temporal structure if wanted (for the following descriptions, we will assume $F(t, z) = 0$ for simplicity). The system is solved on a staggered grid with an implicit time step, which leads to a tridiagonal matrix system that can be easily solved with Python's sparse matrix library.

**If you google "diffusion numerical solution" you'll see countless write-ups describing why this numerical scheme is chosen over "explicit" schemes which would not require a matrix inversion. However, explicit schemes don't necessarily work for diffusion problems, so we need to perform this matrix inversion to obtain a numerically stable solution.*

In the discretization superscripts n denote time and subscripts k denote vertical level. The spatial grid is a staggered grid (taking its notation from ROMS) with v at ρ -levels and κ_v at w -levels. The ρ -levels span $k = 1 \dots N$ and the w -levels span $k = 1/2 \dots N + 1/2$. If this were an ocean, $k = 1/2$ would correspond to the sea-floor and $k = N + 1/2$ would correspond to the sea-surface. The discretization is written as follows (where $\kappa_v = A$ for simpler sub/superscript notation):

Top: $k = N$

$$\frac{H_N v_N^{n+1} - H_N v_N^n}{\Delta t} = -A_{N-1/2}^n \frac{v_N^{n+1} - v_{N-1}^{n+1}}{\Delta z_{N-1/2}} + \text{SBC}^n \quad (2)$$

Interior: $k = 2 \dots N - 1$

$$\frac{H_k v_k^{n+1} - H_k v_k^{n-1}}{\Delta t} = \left(-A_{k-1/2}^n \frac{v_k^{n+1} - v_{k-1}^{n+1}}{\Delta z_{k-1/2}} + A_{k+1/2}^n \frac{v_{k+1}^{n+1} - v_k^{n+1}}{\Delta z_{k+1/2}} \right) \quad (3)$$

Bottom: $k = 1$

$$\frac{H_1 v_1^{n+1} - H_1 v_1^{n-1}}{\Delta t} = A_{3/2}^n \frac{v_2^{n+1} - v_1^{n+1}}{\Delta z_{3/2}} + \text{BBC}^n \quad (4)$$

Where $H_k = z_{k+1/2} - z_{k-1/2}$ is the vertical spacing between w -cells and $\Delta z_{k+1/2} = z_{k+1} - z_k$ is the vertical spacing between ρ -cells. SBC^n and BBC^n are surface and bottom boundary conditions, respectively that can be added if wanted (in the context of ocean velocity, these are a surface wind-stress and bottom-stress).

What we are after in the above equations is calculating v^{n+1} . You should notice that there are v^{n+1} terms on the R.H.S of the discretizations (the presence of those unknown terms on the RHS is what makes this an “implicit” method). So, we cannot simply do simple algebra to solve for the unknown v^{n+1} . Instead, we group the v^{n+1} terms together based on their vertical placement: $v_1^{n+1}, v_{k-1}^{n+1}, v_k^{n+1}, v_{k+1}^{n+1}, v_{N-1}^{n+1}, v_N^{n+1}$. The rearranging gives the following:

Top: $k = N$

$$\begin{aligned} & v_N^{n+1} \left(H_N + \frac{\Delta t A_{N-1/2}^n}{\Delta z_{N-1/2}} \right) \\ & + v_{N-1}^{n+1} \left(\frac{-\Delta t A_{N-1/2}^n}{\Delta z_{N-1/2}} \right) \\ & = H_N v_N^n + \text{SBC}^n \end{aligned}$$

Interior: $k = 2 \dots N - 1$

$$\begin{aligned} & v_{k+1}^{n+1} \left(\frac{-\Delta t A_{k+1/2}^n}{\Delta z_{k+1/2}} \right) \\ & + v_k^{n+1} \left(H_k + \frac{\Delta t A_{k-1/2}^n}{\Delta z_{k-1/2}} - \frac{\Delta t A_{k+1/2}^n}{\Delta z_{k+1/2}} \right) \\ & + v_{k-1}^{n+1} \left(\frac{\Delta t A_{k-1/2}^n}{\Delta z_{k-1/2}} \right) \\ & = H_k v_k^n \end{aligned}$$

Bottom: $k = 1$

$$\begin{aligned} & v_1^{n+1} \left(H_1 - \frac{\Delta t A_{3/2}^n}{\Delta z_{3/2}} \right) \\ & = H_1 v_1^n + \text{BBC}^n \end{aligned}$$

The terms on the L.H.S of the above equations are an unknown $v_{k'}^{n+1}$ (where k' corresponds to a vertical level) multiplied by known coefficients, which we define as follows (consistent with the variables coded in `TD_main.py`):

$$D_1 = \frac{\Delta t A_{N-1/2}^n}{\Delta z_{N-1/2}}$$

$$D_2 = H_1 + D_1$$

$$D_3 = \frac{\Delta t A_{k+1/2}^n}{\Delta z_{k+1/2}}$$

$$D_5 = \frac{\Delta t A_{k-1/2}^n}{\Delta z_{k-1/2}}$$

$$D_4 = H_k + D_3 + D_5$$

$$D_7 = \frac{\Delta t A_{3/2}^n}{\Delta z_{3/2}}$$

$$D_6 = H_1 + D_7$$

To see what this looks like, let's consider a system where $N = 4$ ($k = 1...4$). The tridiagonal matrix for this system is as follows:

$$\begin{bmatrix} D_6 & -D_7 & & \\ -D_3 & D_4 & -D_5 & \\ & -D_3 & D_4 & -D_5 \\ & & -D_1 & D_2 \end{bmatrix} \begin{bmatrix} v_1^{n+1} \\ v_2^{n+1} \\ v_3^{n+1} \\ v_4^{n+1} \end{bmatrix} = \begin{bmatrix} H_1 v_1^n + \text{BBC}^n \\ H_2 v_2^n \\ H_3 v_3^n \\ H_4 v_4^n + \text{SBC}^n \end{bmatrix} \quad (5)$$

The actual coding work is setting up this matrix in Python. Because this matrix is tridiagonal we can use the sparse matrix library in Python (a subset of the SciPy library) to setup and solve the system for the unknowns ($v_{k'}^{n+1}$). Once we have the matrix setup we use the “`spsolve`” function to obtain solutions at the next time step ($n + 1$).