NE 155 Final Project Report

Alexander Blank

Introduction

This code uses methods learned in NE 155 to solve the neutron diffusion equation in a two-dimensional non-multiplying system. This paper outlines the discretization of the diffusion equation into a linear system and the algorithm used to solve the system. The code is at

https://github.com/AlexanderBlank/NE-155/tree/master/Project

diffusion_solver.py can be imported as a module and given a mesh of arbitrary physics values. It will construct a matrix and solve it to give the flux at the edges of the mesh.

Mathematics

The equation we're solving is

$$-\nabla \cdot (D\nabla\phi(x,y)) + \Sigma_a(x,y)\phi(x,y) = S(x,y),$$

where D is the diffusion coefficient, ϕ is neutron flux, Σ_a is the absorption cross section, and S is the value of the fixed neutron source. This means that the net neutron diffusion out of a point, which is proportional to the negative concentration gradient, and the loss of neutrons to absroption are balanced by the neutron source. This differential equaton can be turned into a system of linear equations by discretizing it using the finite volume method, where physics values (D, Σ_a, and S) are defined on mesh cells and flux is defined on mesh edges. For each mesh vertex, the diffusion equation is integrated over a rectangle made up of the nearest corners of the adjacent cells. These discretized equations for each vertex form a linear system. In general, these equations look like this:

$$a_{i,j-1}\phi(x_i, y_j - \epsilon_j) + a_{i-1,j}\phi(x_i - \delta_i, y_j) + a_{i,j}\phi(x_i, y_j) + a_{i+1,j}\phi(x_i + \delta_{i+1}, y_j) + a_{i,j+1}\phi(x_i, y_j + \epsilon_{j+1}) = S_{i,j},$$

where δ and ϵ are, respectively, the mesh cell widths and heights. The factor $a_{i,j}$, the dependence of $\phi_{i,j}$ on itself, includes the absorbtion cross section and diffusion out of the point to its neighbors. Because the diffusion of neutrons from a point (x_1, y_1) to point (x_2, y_2) is equal and opposite to the diffusion of neutrons from (x_2, y_2) to (x_1, y_1), $a_{i,j}$ includes the negative of each diffusion term contributing to $\phi_{i,j}$. This means

$$a_{i,j} = \Sigma_{a,i,j} - a_{i-1,j} - a_{i,j-1} - a_{i+1,j} - a_{i,j+1}.$$

The value of the absorption coefficient for a vertex can be calculated simply by summing $\Sigma_a A$ for each quarter-cell, where $A_{i,j} = \frac{1}{4}\delta_i \epsilon_j$ is the area of the quarter-cell.

$$\Sigma_{a,ij} = \Sigma_{a,i,j} A_{i,j} + \Sigma_{a,i+1,j} A_{i+1,j} + \Sigma_{a,i,j+1} A_{i,j+1} + \Sigma_{a,i+1,j+1} A_{i+1,j+1}$$

The source term is calculated the same way. To find the a values, the integral of the divergence term can be re-written as a line integral using the divergence theorem to get

$$\int_A -\nabla \cdot (D\nabla\phi) dA = \int_C -D\frac{\delta\phi}{\delta\hat{n}} ds,$$

which can be evaluated for the left, right, bottom, and top neighbors to obtain the values of a. The derivative of ϕ in a direction is approximated as the difference in the values of $\phi(x, y)$ on either edge of a cell divided by the length of that cell dimension. This gives, for the right side, for example,

$$\int_{\text{right}} -D\frac{\delta\phi}{\delta\hat{n}} ds = \frac{\phi_{i,j} - \phi_{i+1,j}}{\delta_{i+1}} \int_{\text{right}} D ds = \frac{\phi_{i,j} - \phi_{i+1,j}}{\delta_{i+1}} \left(D_{i+1,j} * \frac{1}{2}\epsilon_j + D_{i+1,j+1} * \frac{1}{2}\epsilon_{j+1} \right)$$

This equation then gives the coefficient $a_{i+1,j}$ for the influence of the flux $\phi_{i+1,j}$ of the right neighbor:

$$a_{i+1,j} = -\frac{1}{2\delta_{i+1}} \left(D_{i+1,j}\epsilon_j + D_{i+1,j+1}\epsilon_{j+1} \right)$$

The same procedure yields similar equations for the other three directions. If the vertex is at an edge of the mesh with a fixed flux condition (like a vacuum boundary), the discretized equation will simply say that flux must be equal to the fixed flux at that edge: $\phi(x, y) = S_{\text{edge}}$.

In this code, if two intersecting boundaries are assigned a different fixed flux, the flux at the corner is taken to be the average of the two.

For a reflecting boundary, the equations are similar to the general case, but all terms that would correspond to cells past the boundary are excluded. The result of the contour integral at an edge or a corner looks the same is it would look if we had a material with $D = 0$ on the other side of the reflecting boundary, since this would enforce zero current at the edge.

A more thorough explanation is available in the NE 155 course notes[1] and in Nuclear Reactor Analysis.[2]

Algorithms

Since the flux at each vertex only depends on the flux in the vertex and the flux in its 4 neighboring vertices, the resulting $(M * N)$ by $(M * N)$ matrix is 5-banded. Each of those bands is stored as a 1-dimensional array. This banded matrix is solved using successive over-relaxation (SOR). At each iteration k of SOR, the flux $\phi_{i,j}$ at a point is updated according to this equation:

$$\phi_{i,j}^{(k+1)} = (1-\omega)\phi_{i,j}^{(k)} + \frac{\omega}{a_{i,j}}\left(S_{i,j} - a_{i-1,j}\phi_{i-1,j} - a_{i+1,j}\phi_{i+1,j} - a_{i,j-1}\phi_{i,j-1} - a_{i,j+1}\phi_{i,j+1}\right) \tag{1}$$

The first term is the weighted value of the current flux and the second term is the weighted solution for $\phi_{i,j}$ assuming the flux at all neighboring points is correct. The value of the wieghting factor ω has a very strong effect on performance. The optimal value of omega is approximated using the following equations from Numerical Recipes.[3]

$$\omega = \frac{2}{1+\sqrt{1-\rho_{\text{Jacobi}}^2}}$$

$$\rho_{\text{Jacobi}} = \frac{\cos\frac{\pi}{M} + \left(\frac{\Delta x}{\Delta y}\right)^2 \cos\frac{\pi}{N}}{1+\left(\frac{\Delta x}{\Delta y}\right)^2},$$

where $\frac{\Delta x}{\Delta y}$ is the ratio of cell width to height. In this code, arbitrary cell widths and heights are allowed, so the ratio of the average cell width to average cell height is used to select ω. The advantage of the Gauss-Seidel method and SOR over Jacobi iteration is that half of the values of ϕ used in Equation 1 have already been updated by the time we get around to updating $\phi_{i,j}$. This can be improved on by taking advantage of the fact that even $i*j$ values depend only on odd $i*j$ values and vise-versa. That is, if the flux values were on a checkerboard, the flux on red squares would depend only on black squares and the flux on black squares would depend only on red squares. If we first update the red squares and then update the black squares, the black squares will be calculated entirely using already updated values. This optimization is called red-black SOR.

For large numbers of mesh cells, this code will first merge adjacent cells together to form a coarser mesh with averaged physics values and solve that. This coarse approximate solution is then interpolated to get an initial guess for the full resolution flux. This is applied recursively until the grid has only a small number of mesh cells.

The time it takes to complete one iteration is $O(M*N)$. The number of iterations needed to solve the problem depends on the specific problem and on the choice of ω. For a square N by N matrix, the number of iterations scales linearly with N for large values of N, meaning that the runtime is $O(N^3)$. This is verified emperically.

Code Use

The code is able to support arbitrary D, Σ_a, and S values in each mesh cell and arbitrary row heights and column widths. diffusion_solver.py is meant to be imported as a Python module so that physics values can be easily defined using python code. The module contains a class `diffusion_solver` that takes a two-dimensional array of `PhysicsVals`, an array of row heights, an array of column widths, and four boundary condition definitions. The `PhysicsVals` class has properties `D`, `Sigma_a`, and `S` for that mesh cell. Calling the `diffusion_solver`'s `solve()` method will have it construct the banded matrix corresponding to the discretized equations and solve for the flux. There are no sanity checks on input physics values since whether or not a set of inputs is insane is nontrivial to determine and setting some conservative restrictions to guarantee that the code won't break would needlessly constrain user freedom. Examples of code use and output are presented in demo.py and demoOutput.txt, respectively. The solver has functions to print version information, inputs, and flux. It also prints mesh dimensions, computation time, and the number of iterations for each grid that it solves.

Additional code use informaiton is included in README.txt and demo.py.

Test Problems and Results

The file tests.py contains four test cases. The first three verify that the code works properly by running it with simple problems where the flux can be determined analytically and comparing the analytical solution with the numerical solution. The first two cases are the problems from the homework where the problem is one-dimensional and the source is in the first case uniform and in the second case varies as $\cos(x)$. The 2D solver can be used as a 1D solver by assigning reflecting boundary conditions to two opposite sides, which models the case of an infinite slab geometry. For the third test, the code is run with a uniform source and vacuum boundaries on four sides and compared to the analytical solution to the two-dimensional diffusion equation. The last test verifies that reflecting boundaries actually do what they're supposed to do by running the same problem once with reflecting boundaries on the top and right edges and once with the mesh values actually reflected. All of these tests use randomized inputs. The code passes all of these tests. However, these tests are fairly simple and do not necessarily mean that the code will correctly solve more complex problems. As of now I am not aware of any bugs.

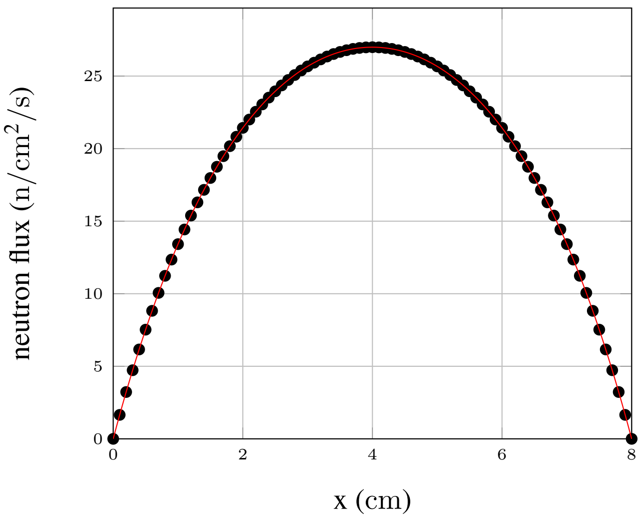

Figure 1: Solution points for one-dimensional fixed source problem vs. analytical solution.

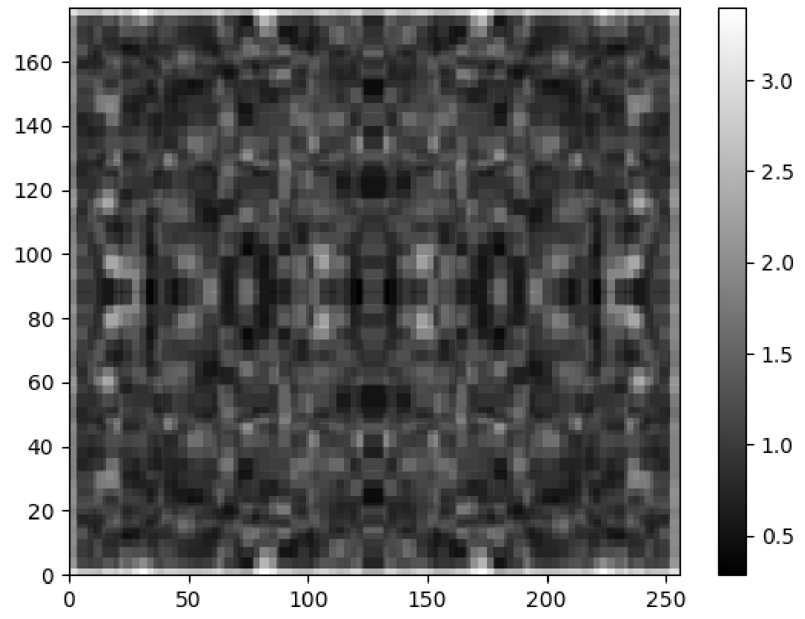

Figure 2: Solution for randomized physics values reflected over the centerlines.

References

[1] Slaybaugh, R. N. NE 155 course materials. https://github.com/rachelslaybaugh/NE155. 2017.

[2] Duderstadt, James J. Louis Hamilton. Nuclear Reactor Analysis. 1976.

[3] Press, William H.; Saul A. Teukolsky. William T. Vetterling. Brian P. Flannery. Numerical Recipes: The Art of Scientific Computing. 3rd edition. Chapter 20.5. 2007.