2D Diffusion Solver
NE155 Final Project

Ruijie Shao

May 10, 2016

1 Introduction

This code is a solver for the diffusion equation in two dimensions. It has two vacuum conditions along the left and bottom boundaries, and two reflecting conditions along the right and top boundaries. The code is split up into three parts:

- the input reader, which parses the data file and creates the variables that will be fed into the solver
- the solver itself, which runs the data through all the algorithms and solves for the final answer
- the output writer, which takes the final data and formats it into an understandable form

There is a main overarching script that executes these three functions in the above order, and is the one that the user will run to begin the program. It also takes in two string arguments for the names of the input and output files to control where the program reads and writes the data.

The input reader first finds the file in the path that the user specifies and begins to read it. Since the file has a predefined format, as discussed in **Section 4**, the input script ensures that there are only $3(n^2) + 4$ given data values, and checks for other possible errors such as D, Σ_a, or S being negative. Then, it will pass on the values for length n, cell width h_x or δ, cell height h_y or ϵ, tolerance ε_t, diffusion coefficient D, cross section Σ_a, and source strength S to the diffusion solver.

The solver uses the arguments given to it to solve for the flux ϕ. It does this by using the discretized five-point difference equation, which is derived in **Section 2**. By creating the matrix **A** and vector $\vec{S}$, it uses the successive over-relaxation algorithm, discussed in **Section 3**, to solve the system to the provided tolerance.

Now that the values for flux have been found, the output script reads that off into a matrix that is written to a file specified by the user. It also plots the flux on a grid so that the user has a visual representation, as well as the numbers in the file.

2 Mathematics

To solve for our diffusion equation, we begin by using the 2D finite volume method. To do this, we will start with our diffusion equation (1), integrate it over the 4 areas around the point (2), and then split the integral up into three terms. We then apply Gauss's Theorem to the first part of the integral (3).

$$-\nabla(D(\vec{r})\nabla\phi(\vec{r})) + \Sigma_a(\vec{r})\phi(\vec{r}) = S(\vec{r}) \tag{1}$$

$$\int_V d\vec{r}\,[-\nabla(D(\vec{r})\nabla\phi(\vec{r})) + \Sigma_a(\vec{r})\phi(\vec{r}) = S(\vec{r})] \tag{2}$$

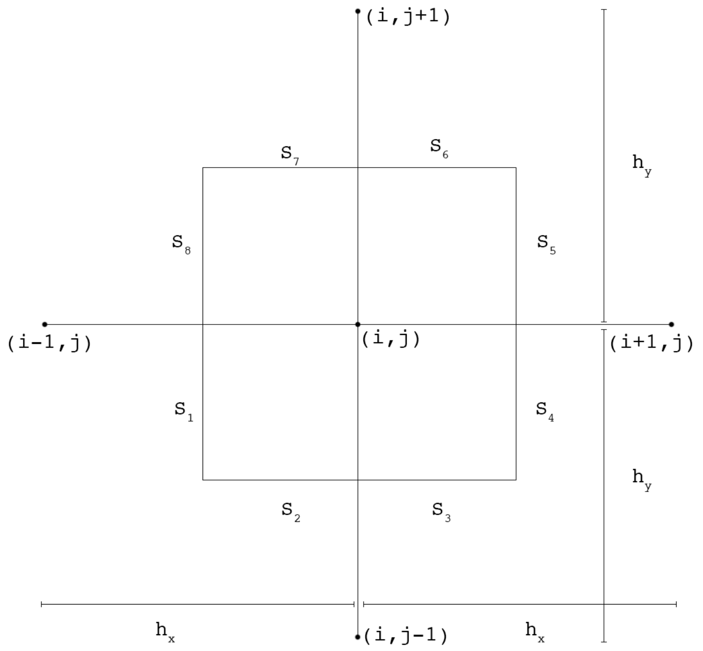

Figure 1: Diagram showing the logistics of a single cell in a finite volume system.

$$\int_V d\vec{r}\ [-\nabla(D(\vec{r})\nabla\phi(\vec{r}))] = \int_S d\vec{S}\ [D(\vec{r})\frac{\partial}{\partial\hat{n}}\phi(\vec{r})] \tag{3}$$

where

$D(\vec{r})$ is the diffusion coefficient, function of position

$\Sigma_a(\vec{r})$ is the absorption cross section, function of position

$S(\vec{r})$ is the source, function of position

$\phi(\vec{r})$ is the flux we are solving for

Now we apply (3) to each edge and use the midpoint rule to integrate and find the streaming term [2]. There are no varying values for δ and ϵ, since one of the conditions we specified was that the mesh was evenly spaced. **Figure 1** shows the positions of the surfaces relative to each other, and why certain ones are calculated together.

$$-\int_{S_2+S_3} d\vec{S}D(\vec{r})\frac{\partial}{\partial\hat{n}}\phi(\vec{r}) = \frac{\phi_{i,j}-\phi_{i,j-1}}{\epsilon}(D_{i,j}+D_{i+1,j})\frac{\delta}{2} \tag{4}$$

$$-\int_{S_4+S_5} d\vec{S}D(\vec{r})\frac{\partial}{\partial\hat{n}}\phi(\vec{r}) = \frac{\phi_{i+1,j}-\phi_{i,j}}{\delta}(D_{i+1,j}+D_{i+1,j+1})\frac{\epsilon}{2} \tag{5}$$

$$-\int_{S_6+S_7} d\vec{S}D(\vec{r})\frac{\partial}{\partial\hat{n}}\phi(\vec{r}) = \frac{\phi_{i,j+1}-\phi_{i,j}}{\epsilon}(D_{i,j+1}+D_{i+1,j+1})\frac{\delta}{2} \tag{6}$$

$$-\int_{S_8+S_1} d\vec{S}D(\vec{r})\frac{\partial}{\partial\hat{n}}\phi(\vec{r}) = \frac{\phi_{i-1,j}-\phi_{i,j}}{\delta}(D_{i+1,j}+D_{i+1,j+1})\frac{\epsilon}{2} \tag{7}$$

Next, we integrate the second term of (2) to find the absorption term and the final term of (2) to find the source term.

$$\int_V d\vec{r}[\Sigma_a(\vec{r})\phi(\vec{r})] = \phi_{i,j}(\Sigma_{a,i,j} + \Sigma_{a,i+1,j} + \Sigma_{a,i,j+1} + \Sigma_{a,i+1,j+1})(\frac{\delta\epsilon}{4}) = \Sigma_{a,ij} \tag{8}$$

$$\int_V d\vec{r}[S(\vec{r})] = \phi_{i,j}(S_{i,j} + S_{i+1,j} + S_{i,j+1} + S_{i+1,j+1})(\frac{\delta\epsilon}{4}) = S_{ij} \tag{9}$$

Combining equations (4), (5), (6), (7), (8), and (9) together, we get our discretized five-point difference equation.

$$a^{ij}_{i-1,j}\phi_{i-1,j} + a^{ij}_{i+1,j}\phi_{i+1,j} + a^{ij}_{i,j-1}\phi_{i,j-1} + a^{ij}_{i,j+1}\phi_{i,j+1} + a^{ij}_{i,j}\phi_{i,j} = S_{ij} \tag{10}$$

where

$$\begin{aligned}
a^{ij}_L &= a^{ij}_{i-1,j} = -\frac{D_{i,j}\epsilon + D_{i,j+1}\epsilon}{2\delta} \\
a^{ij}_R &= a^{ij}_{i+1,j} = -\frac{D_{i+1,j}\epsilon + D_{i+1,j+1}\epsilon}{2\delta} \\
a^{ij}_B &= a^{ij}_{i,j-1} = -\frac{D_{i,j}\epsilon + D_{i+1,j}\epsilon}{2\delta} \\
a^{ij}_T &= a^{ij}_{i,j+1} = -\frac{D_{i,j+1}\epsilon + D_{i+1,j+1}\epsilon}{2\delta} \\
a^{ij}_C &= a^{ij}_{i,j} = \Sigma_{a,ij} - (a^{ij}_L + a^{ij}_R + a^{ij}_B + a^{ij}_T)
\end{aligned}$$

Now we must plug the elements of the discretized equation into a matrix to make it easier to solve [2]. This results in an $n^2 \times n^2$ matrix for **A**, an $n^2 \times 1$ vector for $\vec{\phi}$, and an $n^2 \times 1$ vector for $\vec{S}$.

$$A = \begin{bmatrix} D_0 & U_0 & 0 & 0 \\ L_0 & D_1 & U_1 & \vdots \\ 0 & L_1 & \ddots & U_{n-2} \\ 0 & \dots & L_{n-2} & D_{n-1} \end{bmatrix} \quad \vec{\phi} = \begin{bmatrix} \begin{bmatrix} \phi_{0,0} \\ \phi_{1,0} \\ \vdots \\ \phi_{n-1,0} \end{bmatrix} \\ \begin{bmatrix} \phi_{0,1} \\ \phi_{1,1} \\ \vdots \\ \phi_{n-1,1} \end{bmatrix} \\ \begin{bmatrix} \vdots \end{bmatrix} \\ \begin{bmatrix} \phi_{0,n-1} \\ \phi_{1,n-1} \\ \vdots \\ \phi_{n-1,n-1} \end{bmatrix} \end{bmatrix} \quad \vec{S} = \begin{bmatrix} \begin{bmatrix} S_{0,0} \\ S_{1,0} \\ \vdots \\ S_{n-1,0} \end{bmatrix} \\ \begin{bmatrix} S_{0,1} \\ S_{1,1} \\ \vdots \\ S_{n-1,1} \end{bmatrix} \\ \begin{bmatrix} \vdots \end{bmatrix} \\ \begin{bmatrix} S_{0,n-1} \\ S_{1,n-1} \\ \vdots \\ S_{n-1,n-1} \end{bmatrix} \end{bmatrix} \tag{11}$$

where

$$D_i = \begin{bmatrix} a^{0,i}_C & R^{0,i}_R & 0 & 0 \\ a^{1,i}_L & a^{1,i}_C & a^{1,i}_R & \vdots \\ 0 & a^{2,i}_L & \ddots & a^{n-2,i}_L \\ 0 & \dots & a^{n-1,i}_L & a^{n-1,i}_C \end{bmatrix} U_i = \begin{bmatrix} a^{0,i}_T & 0 & 0 & 0 \\ 0 & a^{1,i}_T & 0 & \vdots \\ 0 & 0 & \ddots & 0 \\ 0 & \dots & 0 & a^{n-1,i}_T \end{bmatrix} L_i = \begin{bmatrix} a^{0,i}_T & 0 & 0 & 0 \\ 0 & a^{1,i}_T & 0 & \vdots \\ 0 & 0 & \ddots & 0 \\ 0 & \dots & 0 & a^{n-1,i}_T \end{bmatrix}$$

Now that we have the discretized, matrix form of the diffusion equation, we can run it through a successive over-relaxation algorithm to solve for the values of the $\vec{\phi}$ vector.

3 Algorithms

The main algorithm implemented in this code to solve it was the successive over-relaxation algorithm [1], which was run with a ω value of 1.1

$$\text{For } i = 1, 2, ..., n$$

$$\vec{\phi}_i^{(k+1)} = (1-\omega)\vec{\phi}_i^{(k)} + \frac{\omega}{A_{ii}}\left(\vec{S}_i - \sum_{j=1}^{i-1} A_{ij}\vec{\phi}_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij}\vec{\phi}_j^{(k)}\right) \tag{12}$$

where

$$\vec{\phi}^{(m)} \text{ is the } m^{\text{th}} \text{ iteration of the flux}$$

$$\omega \text{ is the relaxation factor constant}$$

$$A \text{ is the } n^2 \times n^2 \text{matrix from (11)}$$

$$S \text{ is the } n^2 \times 1 \text{ source vector from (11)}$$

This successive over-relaxation algorithm is run until the final ϕ value does not deviate from the previous iteration by more than the tolerance ε_t, which is specified by the user as one of the arguments from the input file.

The ω value chosen here is not the optimal value, but after several tests for the timing, I concluded that using the successive over-relaxation algorithm with $\omega = 1.1$ was marginally faster than using Gauss-Seidel or Jacobi. This small amount of time saved is very important in the overall picture, however. The iterative solver portion of the code was the most time and resource intensive, so choosing the best possible algorithm here was crucial for optimization, especially since solving larger problems multiplies that time difference to a very noticeable amount.

4 Code Usage

To use the code, an input file with the data is needed. The input should have the following data, in the exact order:

- matrix dimension n
- cell width h_x or δ
- cell height h_y or ϵ
- tolerance ε_t
- matrix of diffusion coefficient D that is size $n \times n$
- matrix of absorption cross section Σ_a that is size $n \times n$
- matrix of source values S that is size $n \times n$

As long these values are in order and each value is separated by either a space or a line break, the code should be able to read the data perfectly. The user will then also specify the input file name and the output file name in the arguments, and the program will write the output to the output file specified. The output will be in the form of a matrix of the phi values. This may be hard to interpret, however, so the program will also provide a graph of the result. The graph is to scale of each cell being a square, so providing different values for δ and ϵ will result in a skewed graph.

5 Test Problems and Results

I ran a few datasets through the script, and the outputs all looked reasonable. The first dataset I used had a constant value for all D, Σ, and S values, and an n value of 32. The resulting ϕ was completely flat in the center and near the reflecting boundaries, but dipped steeply as it approached the vacuum boundaries, as seen in **Figure 2**.

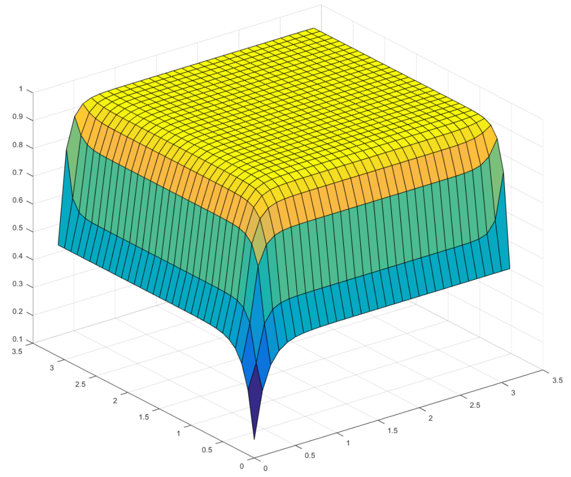

Figure 2: Output of data with constant D, Σ_a, and S.

The other dataset I used was my attempt at modeling the conditions in a reactor. I kept the n of 32, and for the most part, kept the values for D, Σ_a, and S constant. In the center, however, I put higher values of S and Σ_a, and around the boundary of those values I increased the values for D. **Figure 3** shows the end result of running those numbers through the program.

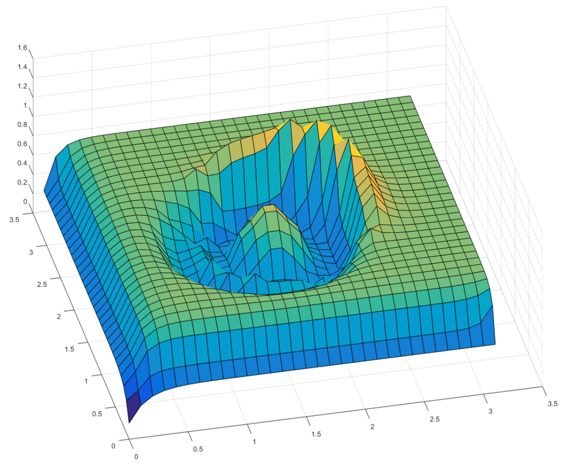

Figure 3: Output of data with values of what I imagined a reactor would be like.

6 Conclusion

This script calculates flux using diffusion coefficient, absorption cross section, and source strength data through three main steps. It first processes the information given and assigns them to variables for the rest of the code to read.

After the variables have been initiated, the solver begins to form the matrices needed to solve the problem. The solver runs through an n^2 algorithm to create the A matrix and the $\vec{S}$ vector in one go. The flux ϕ can technically be solved for now; however, since we have boundary conditions, we have to readjust each boundary to fit those conditions. Now that we have put all our information in, we can run the system through an iterative solver until we reach the tolerance specified by the user. Successive over-relaxation was chosen in this code, because it converges the fastest, which will make a big difference when solving larger problems.

After the flux has been solved for to the required tolerance, it is plotted on a grid to show the heatmap of values, as well as exported to a data file for future usage. The graph is interactive, and allows the user manually rotate and zoom, to examine points of interest.

References

[1] Rachel Slaybaugh, *Iterative Solutions for Linear Systems*, rachelslaybaugh.github.io, March 2, 2016.

[2] Rachel Slaybaugh, *2D Finite Difference/Volume Methods*, rachelslaybaugh.github.io, March 16, 2016.