[[1]](#footnote-1)

Modeling the Diffusion Equation in 2-D

Major Christopher J. Mihal, AFIT Graduate Student

*Abstract*— The diffusion equation is a method to model the transport of neutrons often used for nuclear reactors. As neutrons are generated, they diffuse through the reactor system; modeling this behavior assists one in reactor design control. Basic modeling of the diffusion equation first occurs in one-dimensional space, time independent, in a homogenous medium, with boundary conditions consisting of vacuums of empty space on both sides. To better approximate reality, this project sought to model the diffusion equation in two dimensions (Cartesian coordinates), with added reflective boundaries. This provides a better method to approximate the diffusion of neutrons throughout a fuel pin, as the x- and y-axis of a fuel pin are symmetric. The z-axis is extremely long in comparison and is modeled as infinite in both directions. Modeling of the 1-D diffusion equation was done via utilizing the Thomas Algorithm. Modeling the 2-D equation was attempted via a Gauss-Seidel. Both methods were done entirely using the Python programming language.

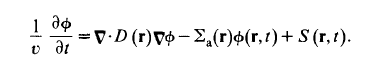
*Index Terms*— Diffusion, Finite-Difference Method, Gauss-Seidel model, Python, Thomas Algorithm

# INTRODUCTION

T

HE diffusion equation is a partial differential equation (PDE) used for assisting nuclear engineers in modeling what is occurring inside of a nuclear reactor. Modeling neutron diffusion aids nuclear engineers by quantifying the neutron population throughout the system allowing for reaction rates to be calculated and predicting criticality of the reactor. As a part of NEN685, all students demonstrated how to model the diffusion equation in one dimension (1-D) utilizing various methods. This project sought to expand on that work and model the diffusion equation in two dimensions (2-D), as well as modifying boundary conditions for a more complete product.

The time-independent diffusion equation isgiven as



where v is…[1]. Removing the time dependence of the equation is often done since nuclear reactors attempt to maintain criticality at a very constant rate.

Modeling the diffusion equation was a lengthy process with much trial and error in conducting the actual code. Initially I attempted to code before understanding the underlying problem, and what the application of the diffusion equation was. This led to a rocky start with the 1-D model, although I made a lot of progress on the ancillary code to accompany the main partial differential equation solver. Once I gathered more information and understood the underlying concepts of why the 2-D model was superior quantitatively, I began multiple methods to code the problem. I will discuss all of my attempts and the reasons for their failures, as each one demonstrated (unintentionally on my part, of course) more information about the diffusion equation and how to model it.

The primary tool for this project was the Python programming language. All code was input into Python. My primary reference was *Nuclear Reactor Analysis* by James J. Duderstadt and Louis J. Hamilton, an invaluable tome for every aspiring nuclear engineer. Much credit goes to U.S. Army LTC Edward Hobbs, NENG Instructor at AFIT, for his many hours of assistance imparting his knowledge to me.

# INITIAL Modelling in 1-D

It is logical to begin modelling the diffusion equation in 1-D and then progressing to 2-D. This allows testing of relevant sections of code that can be common across both models – which I will discuss in section VII – as well as laying the groundwork for the equation to build upon.

Modelling the equation in 1-D involves several factors. First, one must recognize that the x-axis is the only axis in which particles are moving in this model. The initial neutron source is a planar source centered on the y-axis; this keeps the model symmetrical for simplicity’s sake. Taking Equation (1) and applying the initial conditions of time independence permits one to begin coding. Crucial to the code is utilization of the finite volume method, which allows a more complete picture of neutron diffusion for the model.

The finite volume method adds a condition that, rather than measuring from the edge of a cell to the edge of the next cell, we instead measure from the center of each cell to the center of the next. This permits coding in non-homogeneous media, as coding at each cell boundary would create gross inaccuracies as the neutrons change mediums.

Vacuum boundaries are the simplest boundaries to model and conceptualize; once a neutron crosses into one of these boundary regions, it disappears and is no longer part of the system.

Solving the diffusion equation in 1-D is relatively straightforward. Given vacuum boundary conditions leads to the solution

where Φ is the total neutron flux, S is the source term, ∑a is the macroscopic absorption cross section for the medium, x is the length of the x-axis, a is the boundaries of the system, and L is a constant related to the relationship between the diffusion coefficient D and ∑a – essentially, it is the length that neutrons will travel before they are absorbed.

These conditions lead to a graph that closely resembles the shape of a cosine curve, as shown in Figure 1 below. Note that the Thomas Algorithm provides a very close approximation to the values calculated by numerically solving for Equation 2 above.

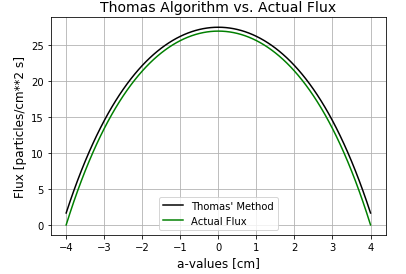


Fig. 1. Comparing the Thomas-Algorithm method to the actual flux solution of the diffusion equation in 1-D, with vacuum boundaries on either side at a=-4,4.

## 

# EVOLVING THE 1-D MODEL

## The next step in the evolution of the equation involves changing the boundary conditions. A reflective boundary is one in which, rather than the neutrons entering the boundary area and exiting the system, they reflect back and remain within the system. Applying that condition to one boundary (the left boundary, in this case) provides the following graph, where the Thomas Algorithm method is plotted against Python’s built-in linear algebra solver:

Fig. 2. Applying the reflective boundary condition to the left boundary. Note that the algorithm matches Python’s linear algebra solver exactly.

## IV EVOLVING TO 2-D: FIRST ATTEMPTS

## Initial Thoughts

With a complete 1-D solver, the next step is to progress to 2-D. This was not as simple of a process as I first assumed. I would once again start with uniform vacuum boundaries and then add in the reflective boundaries once I got the solution I desired. My very first instinct was to simply attempt to solve the x- and y-variables of the PDE separately and then combine the two; unsurprisingly this was both simple and very incorrect. One cannot assume that both variables are independent of each other; this is a closed system, after all, and so naturally the neutrons will flow in both the x- and y- axes simultaneously, not independently of one another. Armed with this knowledge, I resolved to apply the Thomas Algorithm I used to successfully model the 1-D diffusion equation to the full 2-D problem. I was to learn the hard way that this also would not be as straightforward as I’d hoped.

## The Thomas Algorithm

The Thomas Algorithm is a general algorithm to solve tridiagonal matrix problems. In 1-D it is very applicable, as it uses the relationships of the three diagonals of the matrix to reduce the problem down to a single diagonal which can then be solved, as illustrated below in Fig. 3 [1].

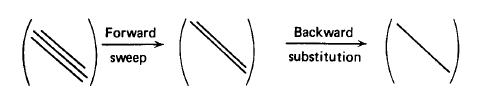


Fig. 3. The Thomas Algorithm method of solving a tridiagonal matrix, represented visually.

However, due to the nature of 2-D matrices, it is not relevant, as 2-D matrices are not tridiagonal at all, but contain seven diagonals, as shown in Fig. 4 below.

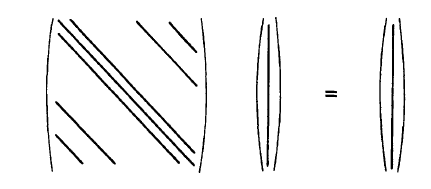


Fig. 4. A 2-D equation will have a 3-D matrix with a total of seven diagonals, rendering the Thomas Algorithm incapable of solving the PDE model [1].

Despite my brute forces attempts to force the data to conform to an algorithm that it could not, I was completely unable to solve the equation in 2-D using the Thomas Algorithm, as expected. Unfortunately, I was unable to comprehend *why* I could not make the solution work, rather focusing on the fact that I could not. I spent an excessive amount of time tinkering with the code in a vain hope to make it work. Finally, after another read-through of Duderstadt, I understood what the problem was. Because the geometry did not line up properly, I would need to use a different iterative method to solve the 2-D diffusion equation.

# Approximating the solution: the gauss-seidel iterative method

Armed with my newfound knowledge, I set to use a different iterative method that could operate in multidimensional space. The Gauss-Seidel model is one such method that can be used to solve for very large matrices, and seemed ideal for what I needed. As one can see in the attached code, however, I was unsuccessful in my application of the Gauss-Seidel method. This may be due to my inaccurate means of setting up my matrix; as I also get errors with Python’s linear algebra solver, it logically follows that my matrix is incorrectly set up, but I am unable to find a means of rectifying the issue. If this issue were solved, the Gauss-Seidel method would converge iteratively to close approximations of the solution; instead, the Gauss-Seidel method exponentially increases to infinity across the entire matrix, which is obviously incorrect.

This also made it impossible for me to use the methods I used to check my solution in the 1-D version. While I was able to plot an approximate graph of the numerical solution – shown below, although it too is not entirely accurate – I could not calculate absolute or relative error for a system that does not work. Unfortunately, due to inaccuracies in the matrix itself, I also could not get Python’s linear algebra solver to provide a solution to check myself against.

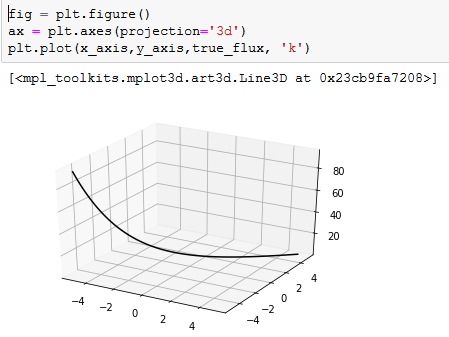


Figure 5. 3-D plot of the numerical solution of the 2-D diffusion equation with reflective boundaries at the right and top edge and reflective boundaries on the bottom and left edge.

While the PDE solver does accurately model the system, there are some conditions worth noting about the code overall. In the 2-D environment, boundary conditions are that the left and bottom of the system be vacuum boundaries, while the top and right be reflective. I kept the source centered on the y-axis.

# ANCILLARY CODE

## Though the PDE solver needs some work, the rest of the code is extremely useful. A number of sections of the code are valuable both for setting up the solution and for general usage regardless of the situation being modelled.

## Python Libraries Utilized

This code used a number of Python’s built-in libraries to run, such as numpy, matplotlib.pyplot, mpl\_toolkits.mplot3d, time, sys, copy and datetime.

## User-Input Variables

## Most important is the user-inputted variable and variable validity check that the code performs. Both 1-D and 2-D systems begin with a number of variables that the user must input. An example is below in Figure 6.

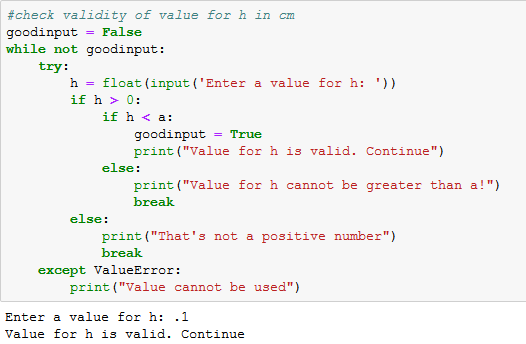


Figure 6. An example of a user-input verified by the code. In this case, the code verifies that the cell length is both positive and less than the overall length of the system, a. Note: when the value for h is less than or equal to 0.01, Python delivers a memory overflow error.

This ensures that the values the user inputs are realistic and capable of entering into the equation without causing unnecessary errors. The code checks all relevant user inputted variables – system length, cell length, source strength, medium macroscopic cross section, and the transport coefficient, and subsequently uses these values to calculate all other values, such as number of times the Gauss-Seidel model must iterate (n) and the neutron mean free path length (L), among others.

## Length of Code Execution

The code contains a section, utilizing Python’s datetime library, that determines the length of time it took for the code to run. This can be handy information for exceedingly large or complex problems for one to visualize exactly how much computing power was required.

## Print to Output File

While a Python notebook can display output directly in the file – especially within a Jupyter notebook – it is useful to have a log of code solutions in an external file for future reference. This code contains the shell of an output file produced by the code itself, allowing for an externally-generated file to be stored for future reference.

# CONCLUSION

The diffusion equation is a crucial tool for nuclear reactor analysis. Modeling how neutrons transport throughout the reactor system is complex, and complexity multiplies as one adds more conditions to better approximate reality. A 2-D diffusion model is a useful midpoint between the initial 1-D model most graduate students are introduced to and the more complex, fully 3-D model of reality. Future work on this code would include, of course, fixing the PDE solver, as well as adding more than one material for neutrons to diffuse through. Modifications to the source, as well as adding multiple sources, would also be very beneficial. One could even go from strictly isotopic sources to directed sources.

References

1. J. L. Duderstadt and L.J. Hamilton, *Nuclear Reactor Analysis,* 1st ed., John Wiley and Sons, Inc. New York, NY, USA: McGraw-Hill, 1976, pp. 150–210.

[2] Ryan McClarren, *Computational Nuclear Engineering and Radiological Science Using Python*, Academic Press, Notre Dame, IN, 2018, pp. 350-360.

[3] David Bodansky, *Nuclear Energy Principles, Practices and Prospects*, Springer Science+Business Media, Seattle, WA, 2004. pp153-168.

[4] Jake VanderPlas, *Python Data Science Handbook*, O’Reilly Media, 2016. Accessed via <https://jakevdp.github.io/PythonDataScienceHandbook/04.12-three-dimensional-plotting.html>

1. Date submitted for review: 12 December 2018. This work was initiated by the Air Force Institute of Technology as a graduate student assignment for NENG685 – Methods of Neutral Particle Transport.

   Major Christopher J. Mihal is a U.S. Army FA52 Nuclear and Counterproliferation Officer and a student at the Air Force Institute of Technology. He can be contacted at [Christopher.mihal@afit.edu](mailto:Christopher.mihal@afit.edu) [↑](#footnote-ref-1)