Final Project: 1D Diffusion in Radial Coordinates

Anastasia Horne

December 2024

1 PDE Overview and Model Introduction

The goal of this project is to create a working python and C code that can handle the 1D diffusion equation in radial coordinates, given that the grid size will increase with increasing r and t because as the radius grows so will the area (the grid) that the heat can diffuse towards. Additionally, we assume a specific set of Neumann boundary conditions at $r = 0.1$ and $r = 100$. Finally, we are given an initial condition. This IC is important because $r = 0$ will only occur when $t = 0$, assuming an instantaneous heating element applied after $t = 0$, thus our model can assume $\boxed{r \neq 0}$. In summary, our model takes the form:

$$\frac{\partial \phi}{\partial t} = K \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial \phi}{\partial r} \right) \tag{1}$$

$$\phi(r = 0.1, t) = 1 \tag{2}$$

$$\phi(r = 100, t) = 0 \tag{3}$$

$$\phi(r, t = 0) = 0. \tag{4}$$

Solving the radial diffusion equation is important for several applications. Properly discretizing and solving this equation could lead to a better understanding of heat diffusion in cylindrical reactors. Additionally, it can help us model battery design (think of a battery as emitting heat), and groundwater flow through a well.

2 Analytical Solution

In order to determine the accuracy of our discretization model we must find a steady-state analytical solution. This means we want to find a function in terms r such that $\frac{\partial \phi}{\partial t} = 0$.

$$0 = K \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial \phi}{\partial r} \right)$$

$$0 = \frac{\partial}{\partial r} \left(r \frac{\partial \phi}{\partial r} \right) \quad \text{(Multiply by k/6);}$$

$$1 = \left(r \frac{\partial \phi}{\partial r} \right) \quad \text{(Integrate both sides)}$$

$$\frac{C}{r} \partial r = \partial \phi \quad \text{(Separate Variables)}$$

$$C \ln r = \phi + D \quad \text{(Integrate both sides)}$$

$$\boxed{\phi = C \ln r - D.}$$

Now, we need to apply our boundary conditions, (2) and (3), to solve for the undefined constants C and D. First, we can apply condition (2), to find,

$$\phi(r = 0.1, t) = 1 = C \ln 0.1 - D \tag{5}$$

Applying condition (3) we see that $\phi(r = 100, t) = 0 = C \ln 100 - D$, and we are able to solve for D, $D = C \ln 100$. We can plug this value for D into (5) and solve for C,

$$C \ln 0.1 - C \ln 100 = 1$$

$$C(\ln 0.1 - \ln 100) = 1$$

$$C(\ln 0.001) = 1$$

$$C = \frac{1}{\ln 0.001}.$$

Now we are able to solve for D,

$$D = \frac{1}{\ln 0.001} \ln 100$$

$$D = \frac{\ln 100}{\ln 0.001}.$$

Given the defined constants, our steady-state analytical solution becomes,

$$\boxed{\phi(r) = \frac{\ln r}{\ln 0.001} - \frac{\ln 100}{\ln 0.001}.} \tag{6}$$

Using *Steady State Analytical Solution*, from "Final_Project.ipynb", we see that our steady-state analytical solution diffuses heat along radii as follows.

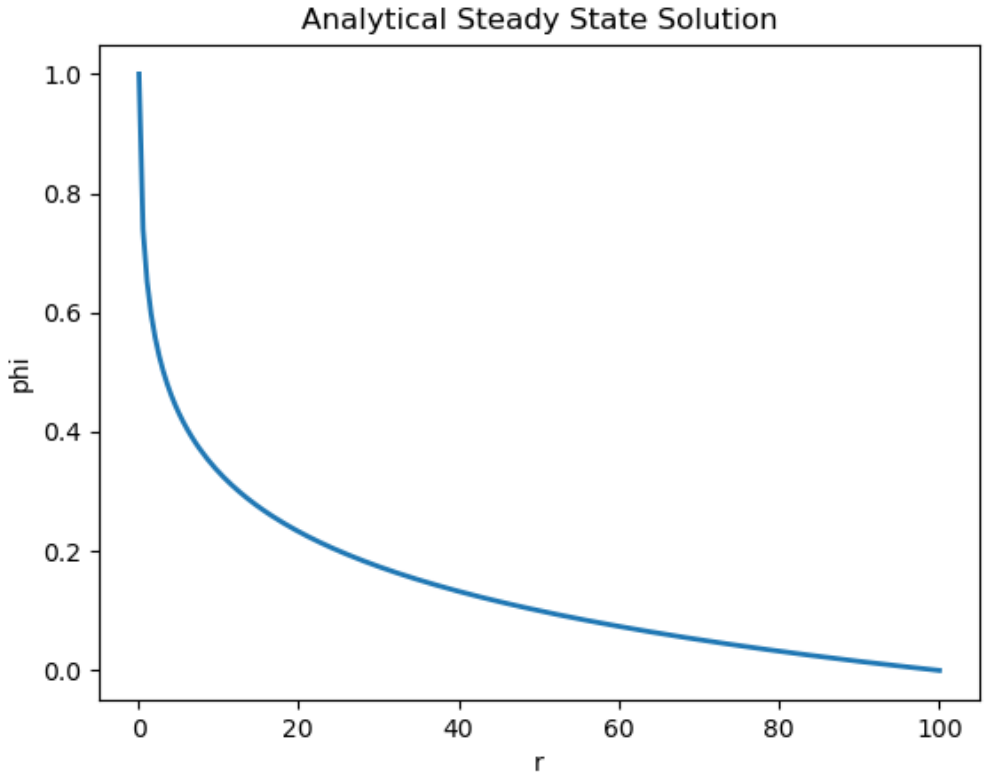

Figure 1: Analytical Steady-State Solution.

We are able to use this solution to see how our discretization changes over time, and how different it is once it reaches steady-state.

3 Discretization

Starting with our radial diffusion equation (1), we first need to discretize in the spatial domain. Since the problem statement assumes an increasing grid size given an increasing r and t, the steps in the spatial domain will not be equal, thus we define the individual spatial steps for the backward, forward, and current/middle steps, given a current step of i.

$$\begin{aligned} r_- &= r_i - r_{i-1} \\ r_+ &= r_{i+1} - r_i \\ r_{mid} &= \frac{r_{i+1} + r_i}{2}. \end{aligned}$$

Now that we have found our discretization steps, we must define the spatial derivatives using central differences.

$$\begin{aligned} \frac{\partial \phi}{\partial r_{i+\frac{1}{2}}} &\approx \frac{(\phi_{i+1} - \phi_i)}{r_+} \\ \frac{\partial \phi}{\partial r_{i-\frac{1}{2}}} &\approx \frac{(\phi_i) - \phi_{i-1}}{r_-} \end{aligned}$$

We will discretize the middle (radial) term at its midpoints.

$$r\frac{\partial\phi}{\partial r_{i+\frac{1}{2}}} \approx r_{mid,i} * \frac{(\phi_{i+1}-\phi_i)}{r_+}$$
$$r\frac{\partial\phi}{\partial r_{i-\frac{1}{2}}} \approx r_{mid,i-1} * \frac{(\phi_i-\phi_{i-1})}{r_-}$$

Now that we have discretized both parts of the PDE in spatial dimensions, we now have to discretize in time using the *forward Euler* approximation method. Using the forward Euler method, (1) becomes,

$$\frac{(\phi_i^{n+1}-\phi_i^n)}{\Delta t} = \frac{K}{r_i}\left(\frac{r_{mid,i}}{r_+}(\phi_{i+1}^{n+1}-\phi_i^{n+1}) - \frac{r_{mid,i-1}}{r_-}(\phi_i^{n+1}-\phi_{i-1}^{n+1})\right)$$

The final step is combining the above and finding like terms.

$$\frac{(\phi_i^{n+1}-\phi_i^n)}{\Delta t} = \frac{K}{r_i}\left(\frac{r_{mid,i}}{r_+}(\phi_{i+1}^{n+1}-\phi_i^{n+1}) - \frac{r_{mid,i-1}}{r_-}(\phi_i^{n+1}-\phi_{i-1}^{n+1})\right)$$
$$(\phi_i^{n+1}-\phi_i^n) = \Delta t\frac{K}{r_i}\left(\frac{r_{mid,i}}{r_+}(\phi_{i+1}^{n+1}-\phi_i^{n+1}) - \frac{r_{mid,i-1}}{r_-}(\phi_i^{n+1}-\phi_{i-1}^{n+1})\right) \text{ (Mult. by } \Delta t)$$
$$(\phi_i^{n+1}-\phi_i^n) = \frac{K\Delta t}{r_i}\left(\left(\frac{r_{mid,i}}{r_+}\phi_{i+1}^{n+1}\right) - \left(\frac{r_{mid,i}}{r_+}\phi_i^{n+1}\right) - \left(\frac{r_{mid,i-1}}{r_-}\phi_i^{n+1}\right) + \left(\frac{r_{mid,i-1}}{r_-}\phi_{i-1}^{n+1}\right)\right) \text{ (Expand the right side)}$$
$$\phi_i^{n+1} + \frac{K\Delta t}{r_i}\left(\frac{r_{mid,i}}{r_+}\phi_i^{n+1}\right) + \frac{K\Delta t}{r_i}\left(\frac{r_{mid,i-1}}{r_-}\phi_i^{n+1}\right) = \phi_i^n + \frac{K\Delta t}{r_i}\left(\frac{r_{mid,i}}{r_+}\phi_{i+1}^{n+1}\right) + \frac{K\Delta t}{r_i}\left(\frac{r_{mid,i-1}}{r_-}\phi_{i-1}^{n+1}\right) \text{ (Comb. like terms)}$$

Now we will define the following coefficients,

$$\alpha_i = \frac{-K\Delta t}{r_- r_i} r_{mid,i-1}$$
$$C_i = \frac{-K\Delta t}{r_+ r_i} r_{mid,i}$$
$$\beta_i = 1 - (\alpha_i + C_i)$$

These give us the final discretized version of (1).

$$\boxed{\phi_i^n = \alpha_i\phi_{i-1}^{n+1} + \beta_i\phi_i^{n+1} + C_i\phi_{i+1}^{n+1}.} \tag{7}$$

We see that (7) takes the form of a tridiagonal matrix.

Now that we have a discretized version of our PDE we can use Python and/or C to find the numerical steady-state approximation of the 1D diffusion equation in radial coordinates. We will first develop a Python implementation of the problem because the author is more familiar with the syntax. Then we will transform the code into C with the help of an artificial learning interface (e.g. ChatGPT, Claude.ai).

4 Overview of Python Code

The 1D solver in Python defines five functions; four functions for the solver, and one function to animate the solution. The first function, *TDMAsolver*, is used to solve the tridiagonal matrix in (7). The second function, *nonuni_grid*, creates a non-uniform grid of r-values. Specifically, the function will return an array of r values from 0.1 to 100 on a log scale. The third function, *Diffusion_1D_Radial*, performs the discretization. This function sets up the coefficients defined above (α, β, C), and iterates over our grid to set up a triadiagonal matrix, which it can then solve. The fourth function, *steady_state*, is used to check if the current approximation is similar to the previous approximation, if 'yes' then the diffusion process will stop. We define all parameters, initialize solution vectors, and apply boundary conditions, then we will run the diffusion function until the steady-state condition is met, saving the diffusion at step t, to a solutions vector each iteration. To see how the diffusion changed over time, we use the final function, *diffusion_1D_animate*, to animate the solution. Referring to the animation in the Python notebook, we see that the temperature curve becomes less and less deep as time moves on. The algorithm found that a steady state was reached at 4189.7 seconds.

We see that at a time slightly after steady-state the heat flow profile is extremely similar to our analytical solution (1 and 2). We notice from both plots that the temperature at $r = 20$ is slightly more than 0.2°C, and the temperature does not reach zero until $r = 100$.

The Python implementation took roughly 70 **seconds** to reach a steady-state solution. The shortest time an iteration has taken is 50 seconds, but it is typically longer than that.

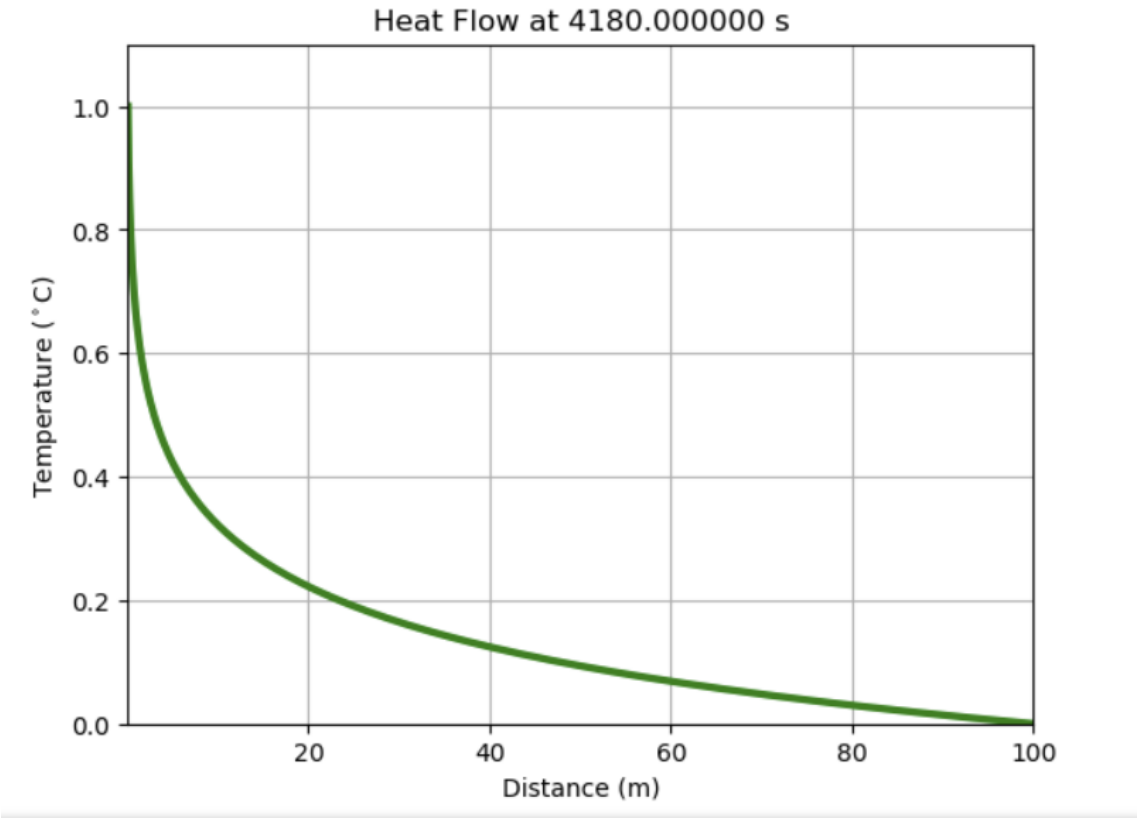

Figure 2: Approximate Steady-State Solution.

5 Overview of C Code

The reason to implement the algorithm in C is to significantly reduce the computation time. After running the C-code, we see that the same steady-state time is found (4189.7s), suggesting that the code is correct.

Additionally, the C-implementation performed as expected. It consistently took **less than** 5 **seconds**, to reach a steady state which is significantly less time than it took the Python implementation.

6 Notes to grader

Unfortunately, I was unable to debug the C code successfully and figure out why the animation created from its solution vector is different than the Python one. What is interesting, is when I run the Python implementation, the solution vector at the time of steady state (found via indexing vector) does not look like the output given by the animation, it looks like a linear line, which would match the C implementation. However, if it did match the C, then neither would match the analytical solution.

Also, the word count is around 1300, which is less than the required 2000, however, that original count does not include the mathematics shown, so I have assumed the math will count for the remaining words.