Using Fast Fourier Transform to Solve Differential Equation with Periodic Boundary Conditions

Raaghav Thatte, Reese Karo, William Mahnke

June 2024

1 Introduction

Differential equations are a field of mathematics with many applications in mathematical modeling, physics, and engineering. A *partial differential equation* (PDE) is a differential equation which solves for a multi-variable function using specific partial derivatives of two or more independent variables. PDEs have many applications in both physics and engineering, including but not limited to the study of vibrations, heat conduction, general relativity, and more.

Partial differential equations can be classified as linear or nonlinear in both first and second orders. An equation is classified by linearity based on if the derivatives are linear terms, and is classified by order based on if the derivatives are first or second partial derivatives.

Our project aims to utilize the Fast Fourier Transform to solve the one-dimensional heat equation, with the goal of understanding how to a real-world application of the FFT.

William Mahnke and Reese Karo took the role of coding, ensuring the FFT and Inverse FFT was applied correctly in solving the heat equation. Raaghav Thatte took the role of writing the report.

2 Methodology

For this study we will focus solely on the *one dimensional heat equation* with periodic boundary conditions, which for some temperature function $u(x,t)$ on the interval $[a,b]$ is of the form

$$u_t = \alpha^2 u_{xx}$$
$$u(a,t) = u(b,t)$$
$$u_x(a,t) = u_x(b,t)$$

where t is time and x is space. Clearly, the one dimensional heat equation is second-order linear.

We utilize the *Fast Fourier Transform* (FFT), a computationally efficient form of the Discrete Fourier Transform (DFT), to solve the problem. The DFT transforms a vector $x = [x_0, \ldots, x_{n-1}]^T$ into another n-dimensional vector $y = [y_0, \ldots, y_{n-1}]$ through the relationship:

$$y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega^{jk}, \quad \text{where } \omega = e^{-i2\pi/n} \tag{1}$$

FFT improves upon DFT by decomposing the n-point transform into two smaller $n/2$-point transforms, which significantly reduces the computational expense from $O(n^2)$ to $O(nlogn)$. This optimization is crucial for the practical application of solving differential equations numerically.

We apply the FFT to $u(x,t)$ to get $\hat{u}(\kappa,t)$, where $\hat{u}$ is a vector of Fourier coefficients and κ is now a spatial frequency. When applied to the spatial derivatives, we now get $u_x \to i\kappa\hat{u}$ and $u_{xx} \to i^2\kappa\hat{u} = -\kappa\hat{u}$. After applying the FFT to the heat equation itself, time derivative becomes $\hat{u}_t \to -\alpha^2\kappa^2\hat{u}$, which are n decoupled ordinary differential equations. We then will simulate the decoupled ODEs in the Fourier system, and then apply the Inverse FFT to find our original $u(x,t)$ with respect to space and time.

3 Computational Setup

We begin by initializing the settings and discretization, creating the wave frequencies, and creating the initial vector for the initial conditions. Then we creating a set of points for the time domain and also create functions for the calculating $-\alpha^2\kappa^2\hat{u}$ for all values of kappa (or omega in the code) and for the Runga-Kutta method to solve the simplifed ode for the values of kappa. We then perform the reverse operations use '$numpy.fft.ifft()$' to get the computational solution. We tested the results against scipy's '$odeint()$' to see how far we could puss our RK method. We then used '$matplotlib$' and '$mpl_toolkits.mplot3d$' to help graph the solution.

We also compared the results to the Backwards Difference Method we developed in M104B.

4 Code

In our Jupyter notebook, we set up the computational environment similarly to Steve Brunton's Python file from [1]. Our code was inspired by his as a sanity check using similar techniques; however, for uniqueness, we implemented the fourth-order Runge-Kutta (RK4) method, which approximates and solves ODEs with great accuracy and stability. More information can be found in Professor Jea-Hyun Park's notes.

By utilizing Python's NumPy package for vectorized coding, SciPy's ordinary differential equation integration function to solve the ODEs we derived by

calculating the Fourier transform of our initial condition, as well as the partial derivatives with respect to space and time, we set up our computations. We first introduced a tridiagonal function discussed in Math 104B, where the function takes a sub-diagonal value (below the main diagonal), diagonal value, and super-diagonal value (above the diagonal). This function is used for the backward difference method, which will be implemented as another sanity check and comparison to the Fast Fourier Transform method for solving PDEs like the heat equation.

Initially, we set the initial condition for each spatial value to 1, then evolved the solution in time to observe the diffusion process. After validating this with Steve Brunton's code from [1], which uses Scipy's 'odeint' function, we chose a different initial condition, taking the form $u(x, m; 0) = \frac{e^{-\frac{x^2}{2m}}}{\sqrt{4\pi m}}$. This can be found in Figure 1 in Section 5. To compare the results to our Fast Fourier Transform method, we implemented the backward difference method using `np.linalg.solve` to solve a matrix of positional points for the next time step.

5 Results

Using the Runga Kutta method to solve the family of ODEs proved to be relatively successful. However, the numerical solution begins to fail and "blow up" when the difference in frequency domain becomes incredibly small. But with acceptable values for '*dt*', the graph shows smooth results. Using '*odeint*' proved to have similar results to our ODE estimator but was able to handle smaller values for the time discretization. Additionally, the results from using the Backwards Difference Method showed the same smooth answer as the previously mentioned methods.

6 Conclusion

The improvement in computational time using the FFT creates a strong argument for using it to numerically solve PDEs by also using an ODE solver. However, the results of this project show that there are other efficient methods from solving PDEs, namely using a Finite Difference Method like the Backwards Difference Method. The BDM is stable for all discretization values while the ODE solver used in the FFT method can become unreliable for increasing fine discretizations of time. The FFT is heplful, but other methods can be considered better when handling more complicated systems and investigating finer intervals of time. Below we have the backwards FDM method of the same initial condition and heat equation settings as the Fourier Transform found in 1,

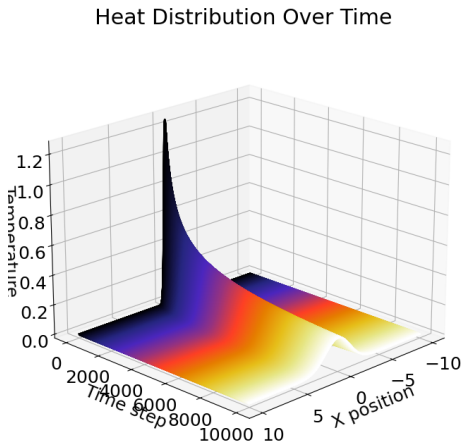

Figure 1: Solution to the heat equation using FFT with an initial condition of $\frac{e^{-x^2/(0.1)}}{\sqrt{0.2\pi}}$

6.1 Future Work

Future work could compare the results of using FFT to solve PDEs using a Neural Network to compare accruacy and computational speed. While FFT is reliable for higher level PDEs, there are complicated PDEs that could be only solved using neural nets or artificial intelligence, or with a much higher order of accuracy. Using one of these aternatives could also be beneficial in the case of a unusually-shaped region.

References

[1] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. Cambridge University Press, 2019.

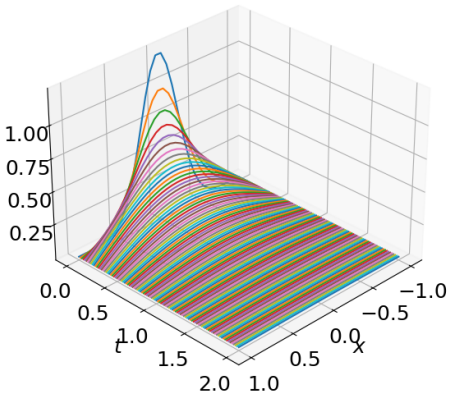

Figure 2: Solution to the heat equation using Backwards FDM method with an initial condition of $\frac{e^{-x^2/(0.1)}}{\sqrt{0.2\pi}}$
, however, with different coloring and spatial domain, but the same solution.