4/22/2024

Project 1 Report

MATH 453

Cameron Robinson

Table of Contents

Table of Figures

Project 1

Part 1

The objective of part one was to implement the power method to calculate the dominant eigenvalues of the matrix shown in Figure 1. Figure 2 shows the implementation of the power method. First, a new eigenvector is calculated by multiplying the matrix A with the previous eigenvector, x_{k-1}. Then the eigenvector is normalized before being used to calculate the eigenvalue by multiplying the transpose of the eigenvector with Ax_k. This process is iterated until the difference between the previous and current eigenvalues is less than the tolerance, or a maximum number of steps is reached. Since the power method is used to find the largest eigenvalue, the smallest eigenvalue can be found by performing the power method on A^{-1}. The code for the power method is in Figure 25 in the appendix.

$$A=\begin{bmatrix} 2 & 1 & -1 & 3 \\ 1 & 7 & 0 & -1 \\ -1 & 0 & 4 & 2 \\ 3 & -1 & -2 & 1 \end{bmatrix}$$

Figure 1: Matrix for Part 1

$$x_k = Ax_{k-1}$$

$$x_k = \frac{x_k}{|x_k|}$$

$$\lambda = x_k^T A x_k$$

Figure 2: Power Method Implementation

Figure 3 shows the solutions calculated with the power method and a built-in eigenvalue solver. The last line also shows the calculated error between the two solutions. Both the eigenvalues and eigenvectors calculated by each method are very close to each other. The eigenvalues are within a millionth of each other, while absolute value of the eigenvector components are slightly further apart from each, but still within 1/1000 of each other.

```
Built-In:
(7.227187681124078+0j)
[-0.1773186 +0.j -0.97960517+0.j  0.08297932+0.j
0.04523562+0.j]

Estimated:
[[7.22718754]]
[[ 0.17730267]
 [ 0.97960515]
 [-0.08300938]
 [-0.04524338]]
Error: 1.4036758066993116e-07
```

Figure 3: Estimated and Actual Eigenvalues and Eigenvectors

Since the power method can be used to find both the largest and smallest eigenvalues for a matrix, it can be used to solve a second-order ODE-BVP because it can be made into a system of first-order ODEs with two eigenvalues. The calculated eigenvalues can be substituted into one of the general-form ODE solutions, depending on the types of eigenvalues (repeated, distinct, complex). The solution's constants would still need to be calculated using the boundary conditions. A downside of this method is that the system could have repeated eigenvalues and the power method requires them to be ordered.

Part 2

The objective of part two was to gain a better understanding of PDEs by researching how they are different from ODEs, their different classifications, and some physical examples of them.

The most significant difference between ODEs and PDEs is that ODEs contain functions and derivatives in terms of one variable, while PDEs contain functions and derivatives with multiple variables [1]. Similarly, the solutions found by solving a PDE are typically dependent on multiple variables, while the solutions to ODEs depend on one variable.

Linear second-order PDEs can be classified into three general categories: parabolic, elliptic, and hyperbolic. The general form of a PDE and how to classify them is shown in Figure 4, where the coefficients (A, B, C, etc.) are functions of x and y. A parabolic PDE generally models systems that diffuse over time, such as the heat equation [2]. Parabolic PDEs often have some time-dependent variable and their solutions tend to smooth-out as time increases [4]. Hyperbolic PDEs typically model the movement of physical quantities, such as the wave equation [3]. These types of PDEs also tend to have a time-dependent variable, but the solutions will often retain their discontinuities rather than smooth-out like parabolic PDEs. Lastly, elliptic PDEs tend to model systems after they have reached steady-state (equilibrium), like how Laplace's equation models the steady-state heat equation. Unlike parabolic and hyperbolic PDEs, the solutions to elliptic PDEs do not have discontinuities, so they are smooth everywhere. The solutions to elliptic PDEs are often similar to parabolic PDEs as their solutions approach equilibrium.

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu = 0$$

$$If\ B^2 - AC = 0, then\ PDE\ is\ parabolic.$$

$$If\ B^2 - AC < 0, then\ PDE\ is\ elliptic.$$

$$If\ B^2 - AC > 0, then\ PDE\ is\ hyperbolic.$$

Figure 4: General Form for PDEs and Classification

As mentioned, physical examples of elliptic, parabolic, and hyperbolic PDEs are Laplace's equation, the heat equation, and the wave equation respectively. These PDEs are illustrated in Figure 5. The heat equation represents the diffusion of heat over some medium. As time increases, the heat distribution will eventually reach equilibrium, which can be represented by Laplace's equation. Laplace's equation has no time dependency since the heat distribution has already reached equilibrium. The solution simply shows the temperature at different physical locations. The solution to the wave equation models the

movement of a wave over time. It represents the displacement of the medium that the wave is travelling through at a specific location and time.

$Heat\ Equation\ (1D)$:

$u_t = ku_{xx}$

$Laplace's\ Equation$:

$0 = u_{xx} + u_{yy}$

$Wave\ Equation\ (1D)$:

$u_{tt} = c^2 u_{xx}$

Figure 5: Heat, Wave, and Laplace's Equations

Boundary-value problems (BVPs) are PDEs and ODEs where the solutions are constrained along the boundary of a curve or line (e.g., the ends of a rod, or the edge of a disk). Initial-value problems (IVPs) are PDEs and ODEs where the solutions are constrained at a specific point [5]. Boundary constraints are given for different values of the dependent variables, while initial-value constraints are given for the same value of the dependent variable. For example, the 1D heat equation has both initial-value and boundary-value constraints. The boundary-value constraints are on the two ends of the rod that the heat radiates through, while the initial-value constraint specifies the temperature along the entire rod when the time-variable is zero. There are multiple types of boundary conditions; two of which are Dirichlet and Neumann. Dirichlet boundary conditions specify the value of the solution to the PDE or ODE along the boundary (e.g., u(a, t) = A, u(b, t) = B, y(a) = A, y(b) = B) [6], while Neumann boundary conditions specify the value of the solution's derivative along the boundary (e.g., u_x(a, t) = A, u_x(b, t) = B, y'(a) = A, y'(b) = B) [7].

Part 3

The objective of part three was to use the forward-time centered-space finite difference method (FTCS) to solve the 1D heat equation shown in Figure 6. The figure also contains the exact solution to the PDE. As mentioned in part two, the heat equation is a parabolic, second-order, linear PDE. One method to solve these types of PDEs is through finite difference methods.

Figure 7 shows the implementation of the FTCS method. This method uses the temperatures at the previous time-step to estimate the temperature of each location at the next time-step. Each new temperature estimate at the next time-step uses three values at the previous time-step: one at the same location on the rod, one to the left, and one to the right. The downside of this method is that the time-step is restricted by the condition: $h \leq \frac{l^2}{2k}$, where h is the time-step, l is the space-step, and k is the conductivity constant. This means that the method needs to take many more steps in time than in space, which can greatly slow down the method for small space-steps. The code for the FTCS method is shown in Figure 26.

$Heat\ Equation$

$u_t = k\, u_{xx}$

$u(0, x) = \sin(\pi x)(1 + 2\cos(\pi x))$

$u(t, 0) = u(t, 1) = 0$

$Solution$:

$u(t, x) = \sin(\pi x)\, e^{-\pi^2 t} + \sin(2\pi x)\, e^{-4\pi^2 t}$

Figure 6: Heat Equation for Part 3 and Solution

$FTCS$

$u(t, x) = u(t - h, x) + \frac{kh}{l^2}(u(t - h, x + l) - 2u(t - h, x) + u(t - h, x - l))$

$h - time\ step$

$l - space\ step$

$k - conductivity\ coefficient$

Figure 7: Forward-Time Centered-Space Method Implementation

Figure 8 - Figure 10 contain solution graphs estimated by the FTCS method with 5, 10, and 20 steps in space. Figure 11 contains a graph of the actual solution. As the number of x-steps increases, the FTCS method converges towards the actual solution. The 5-step graph does not appear to be very accurate, but the 20-step graph is very similar to the actual solution.

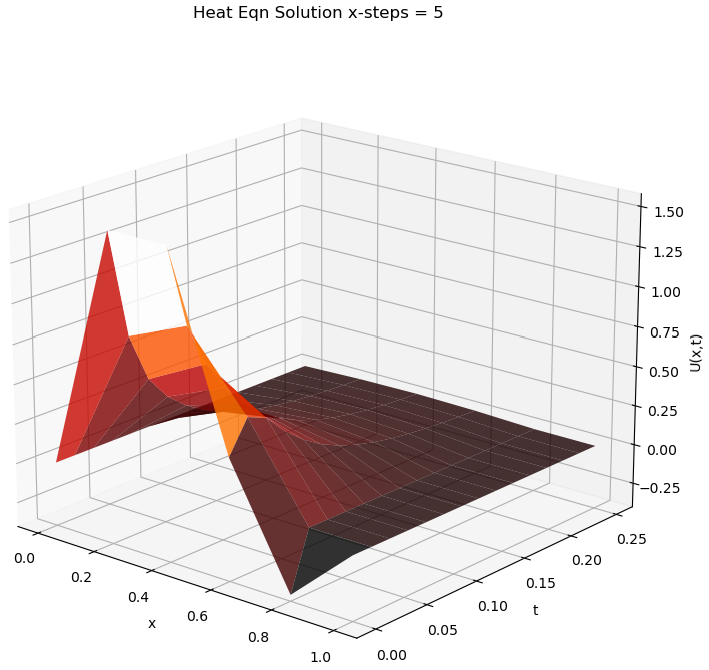

Figure 8: Graph for Part 3 Solution w/ Steps = 5

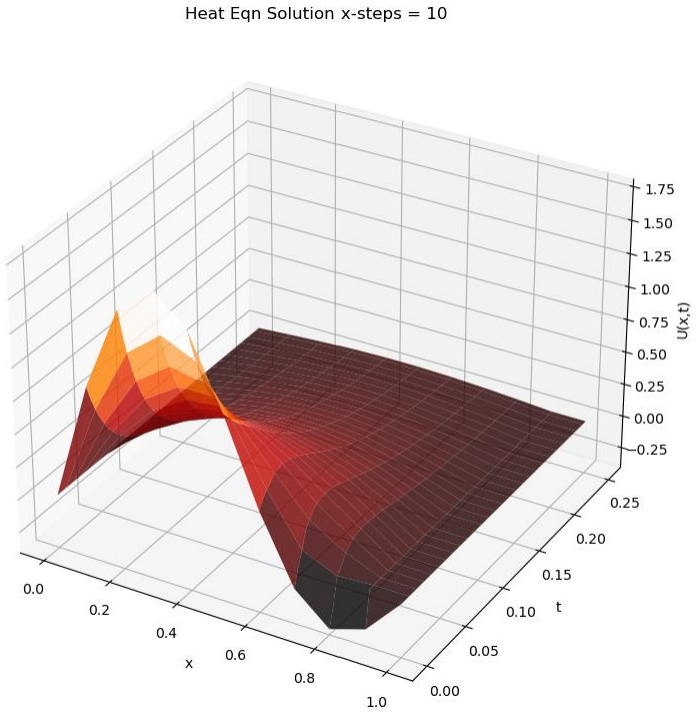

Figure 9: Graph for Part 3 Solution w/ Steps = 10

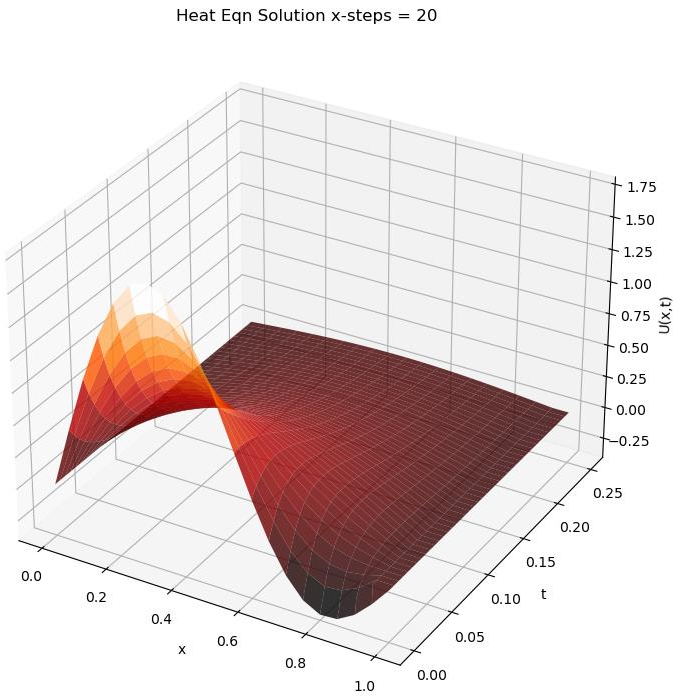

Figure 10: Graph for Part 3 Solution w/ Steps = 20

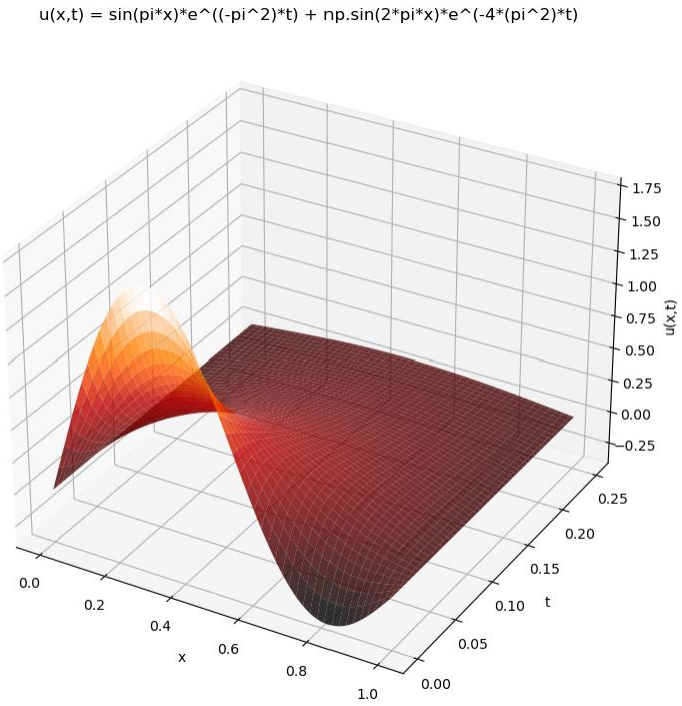

Figure 11: Actual Solution to PDE (Part 3)

Figure 12 contains graphs of the errors calculated by finding the difference between the 5, 10, and 20-step estimates and the actual solution. These graphs do a better job of showing the convergence of the method as the number of x-steps double. Since the FTCS method has error $O(h) + O(l^2)$, the solutions

should become 2 - 4x more accurate when the number of steps double. This improvement is most obvious between the 5-step and 10-step graphs, though some improvement is still visible between 10 and 20-step graphs as well. I am not sure why the method does not perform very well at the front right corner of the graph. The error still drops to zero at the boundaries and at t=0, but the error quickly grows around those points. The code to create the graphs for part three is shown in Figure 27 and Figure 28.

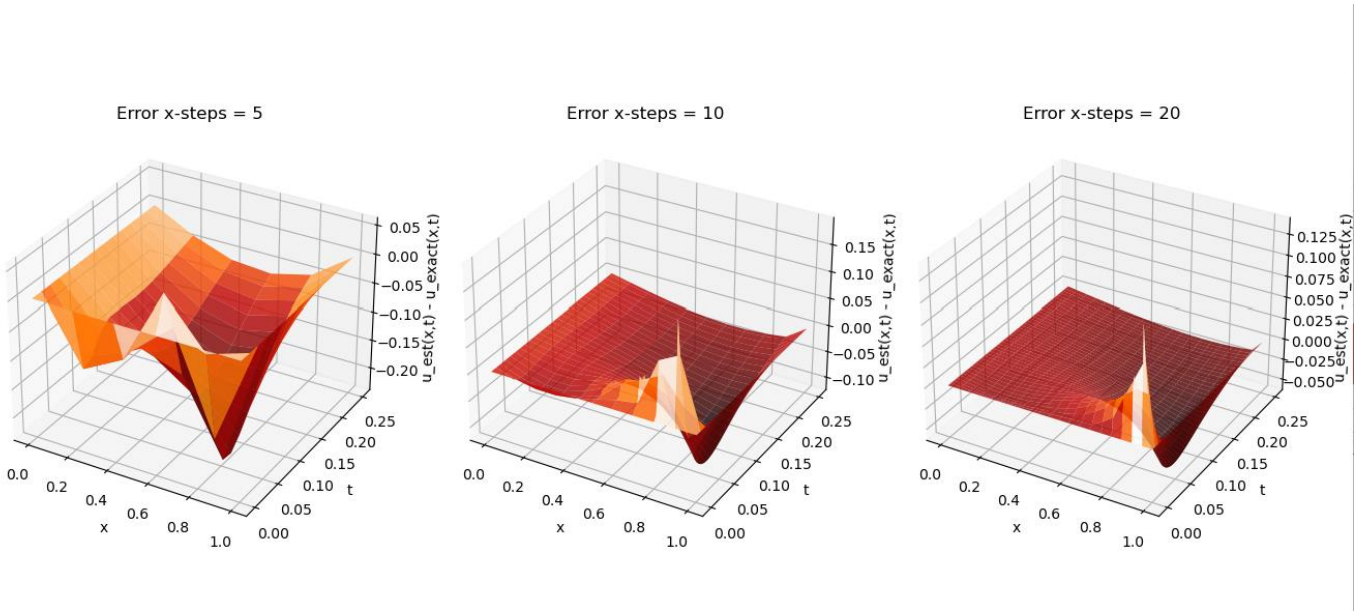

Figure 12: Error Graphs for Part 3

Part 4

The objective of part four was the same as part three, the only difference is that the backward-time centered-space finite difference method (BTCS) is used to solve the heat equation instead. The benefit of this method is that the time-steps do not have the same restrictions as the FTCS method, however, all of the temperatures at a certain time-step must be solved at the same time. This is accomplished by creating a matrix, A, and solving Ax = b at each time step, where the x-vector contains the unknown temperatures, and the b-vector contains the temperatures at the previous time-step. The first and last rows of A are all zeros except for the top-left and bottom-right elements, which are one. The middle rows of A are tri-diagonal with each row containing the same three elements: -r, 1 + 2r, -r, where $r = hk/l^2$ (h, l, and k are the same as part three). An example 5x5 matrix is shown in Figure 13; the number of rows and columns is equal to the number of x-steps + 1. The code to for the BTCS method is shown in Figure 29 in the appendix.

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -r & (1+2r) & -r & 0 & 0 \\ 0 & -r & (1+2r) & -r & 0 \\ 0 & 0 & -r & (1+2r) & -r \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 13: Example A Matrix for Part 4 and 5

Figure 14 - Figure 16 show graphs of the solutions calculated by the BTCS method. Overall, the graphs look nearly identical to the ones from FTCS. As the number of steps increase, the solutions appear to converge towards the actual solution shown in Figure 17 –which is the same solution graph from part three.

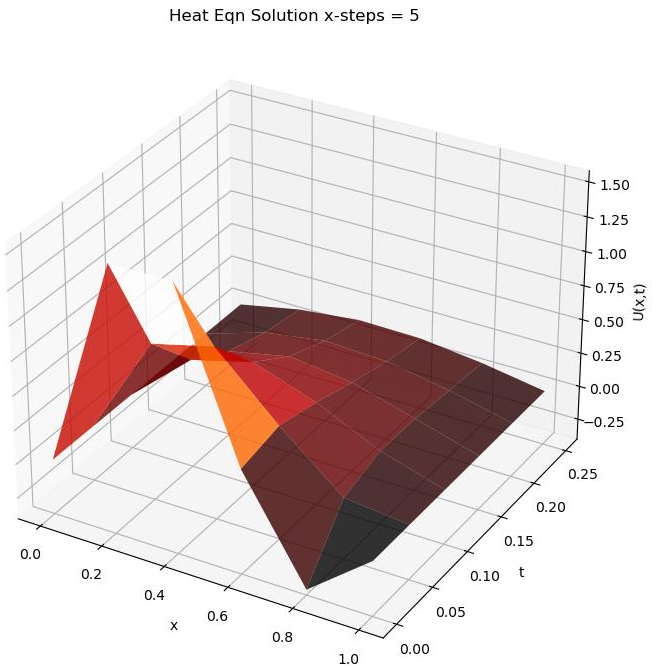

Figure 14: Graph for Part 4 Solution w/ Steps = 5

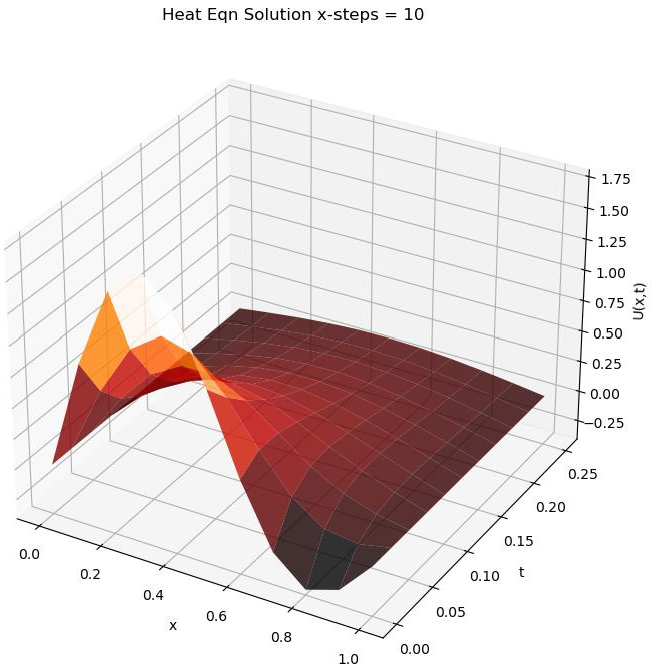

Figure 15: Graph for Part 4 Solution w/ Steps = 10

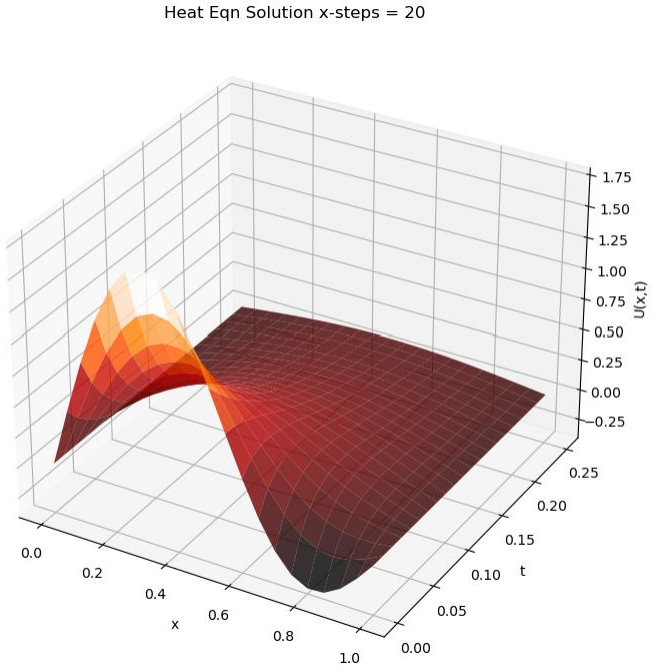

Figure 16: Graph for Part 4 Solution w/ Steps = 20

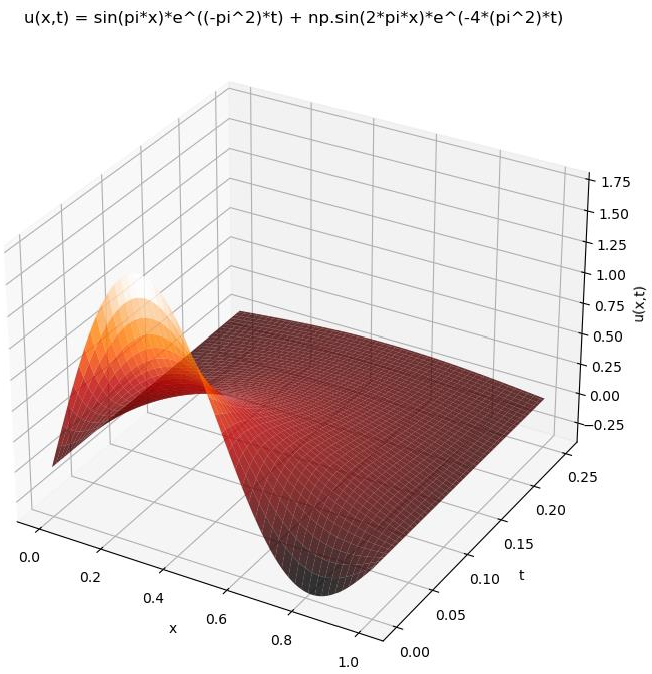

Figure 17: Actual Solution for PDE (Part 4)

Figure 18 contains three subplots of the error for each of the estimated graphs above. At first glance, the error from the BTCS appears to be more than the FTCS method, however, the largest error for each graph tends to be equal to or smaller than in the FTCS error graphs. For example, in the BTCS 20-step graph, the largest error value is about 0.08, while in the FTCS graph, it is about 0.125. Aside from the largest errors, however, the BTCS method does tend to have more error simply because the number of time-steps is a fraction of the number used in the FTCS method.

The error of the BTCS method is $O(h) + O(l^2)$, which means that as the number of steps doubles, the amount of error should decrease by about 2 – 4x. Looking at the scales on the error graphs, this does appear to be the case to some extent since the max error is 0.25, 0.10, 0.08 for the 5, 10, and 20-step graphs respectively. The code to create all of the graphs for part four is shown in Figure 30 and Figure 31.

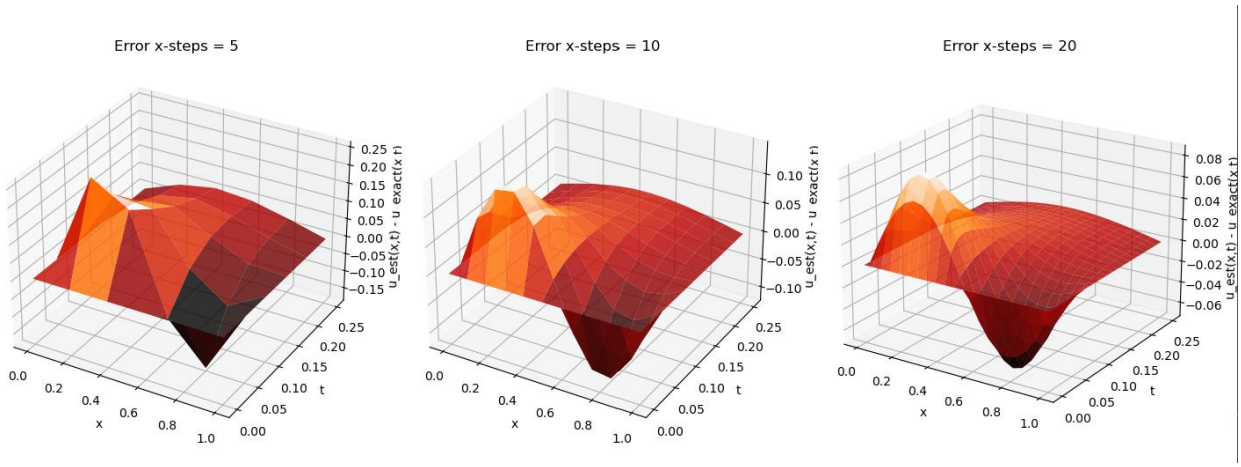

Figure 18: Error Graphs for Part 4

Part 5

The objective of part five was to implement the Crank-Nicolson (C-N) method to solve the same heat equation from parts three and four. This method combines the FTCS and BTCS methods to create a method that is more accurate than the previous methods and does not have stability constraints. The process of the C-N method is practically identical to the BTCS method. The A matrix has the same form, and Ax = b is still being solved for all the temperatures in space at a specific time. The differences between the methods are the values of r and b. The values of r and b for the C-N method are shown in Figure 19. The elements of the b vector are equal to the right-hand side of the equation, while the left-hand side represents the Ax in Ax = b. The code for the C-N method is shown in Figure 32 in the appendix.

$$r = \frac{hk}{2l^2}$$

$$-r\,u(t+h,x-l) + (1+2r)u(t+h,x) - r\,u(t+h,x+l) = r\,u(t-h,x+l) + (1-2r)u(t-h,x) + r\,u(t-h,x-l)$$

Figure 19: Crank-Nicolson Method Implementation

Figure 20 - Figure 22 contain the graphs of the solutions calculated with the C-N method at 5, 10, and 20 steps in both space and time. Looking at the graphs, they are pretty much identical to the solutions calculated in parts three and four. Like the other methods, the solutions converge towards the actual solution (shown in Figure 23) as the number of steps increase.

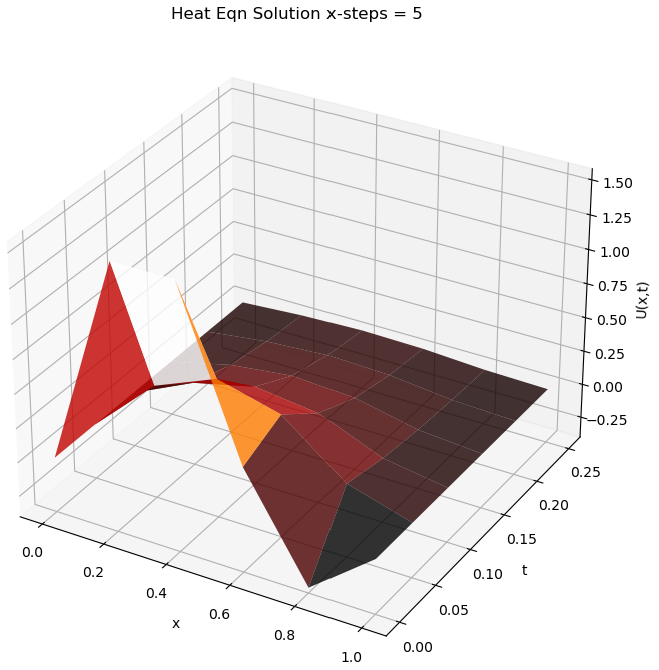

Figure 20: Graph for Part 5 Solution w/ Steps = 5

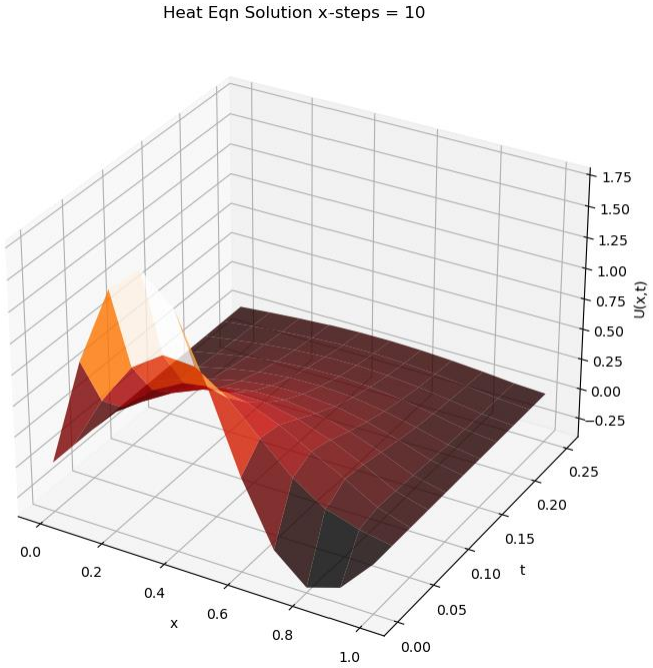

Figure 21: Graph for Part 5 Solution w/ Steps = 10

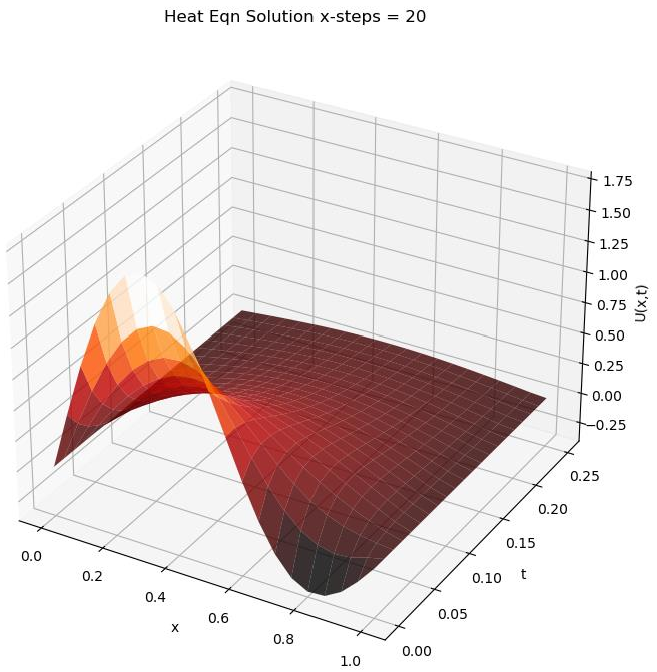

Figure 22: Graph for Part 5 Solution w/ Steps = 20

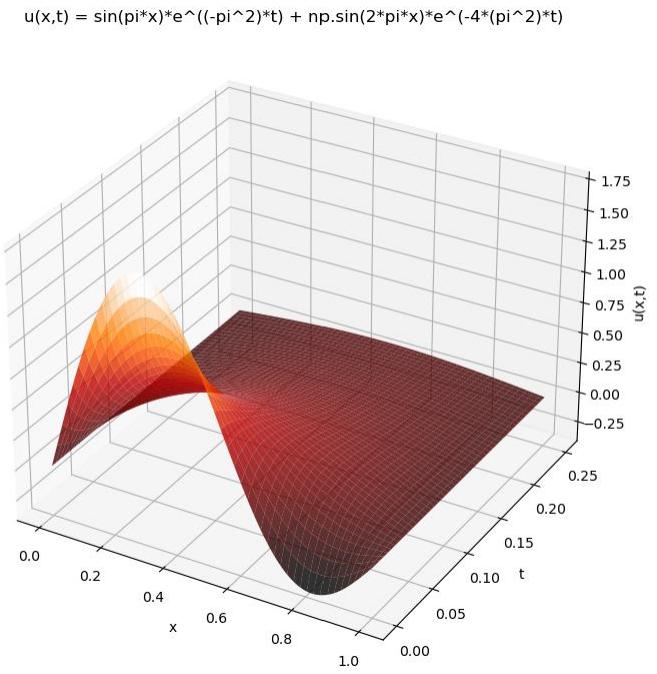

Figure 23: Actual Solution to PDE (Part 5)

Figure 24 contains the error graphs for the Crank-Nicolson method at 5, 10, and 20 steps. Since this method has $O(h^2) + O(l^2)$ accuracy, the solutions should converge faster than the FTCS and BTCS methods. This does appear to be the case since the maximum error for each graph is less than the maximum error in the other methods. Looking at the scales on the graphs, the error seems to decrease by about 2 – 4x when the number of steps doubles.

Something that is interesting about the error is that it has features of both the FTCS and BTCS error graphs. The graphs tend to have one or two large spikes like in the FTCS graphs, and the error in the first couple of time-steps is shaped like a sinewave like the BTCS graphs. These similar features make sense because the C-N method combines the FTCS and BTCS methods. The code to create the graphs for this section is shown in Figure 33 and Figure 34.

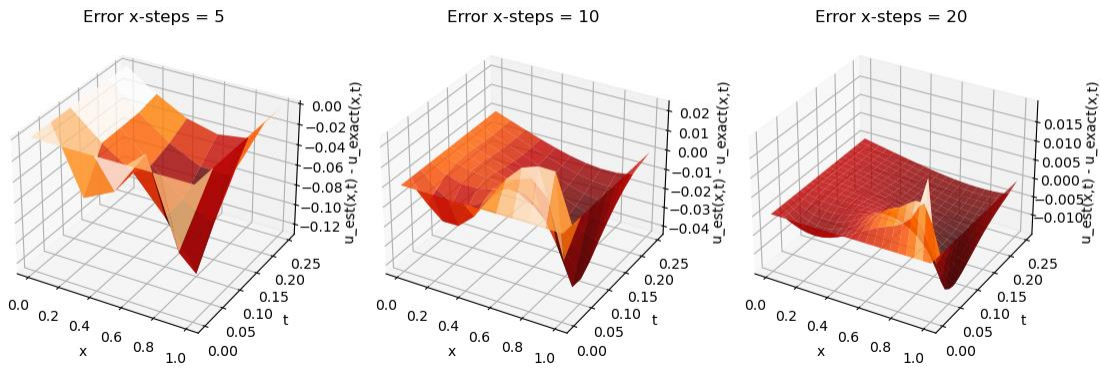

Figure 24: Error Graphs for Part 5

Appendix

```python
from math import e
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def PowerMethod(A, x0, tol, max_steps):
    steps = 0
    prevEigenVal = 0
    eigenVal = 0
    #Repeat while improvement is larger than the tolerance, and the steps is less than the max steps
    while (steps == 0 or ((abs(prevEigenVal - eigenVal) > tol) and (steps < max_steps))):
        #Multiply A and current eigenvector
        Ax = np.dot(A, x0)              #x = Aq
        x0 = Ax / np.linalg.norm(Ax)    #q = x/|x|
        prevEigenVal = eigenVal
        eigenVal = np.dot(Ax.T, x0)
        steps += 1
    return eigenVal, x0

A = [[2, 1, -1, 3],
     [1, 7, 0, -1],
     [-1, 0, 4, 2],
     [3, -1, -2, 1]]

x = [[1], [0], [0], [0]]
A_inv = np.linalg.inv(A)
print("Built-In: ")
eigs1, eVecs = np.linalg.eig(A)
print(eigs1[-1])
print(eVecs[:, 3])
print("\nEstimated: ")
eigs, eVecs = PowerMethod(A, x, 0.00001, 100)
print(eigs)
print(eVecs)
print("Error: " + str(abs(eigs1[-1] - eigs)[0,0]))
```

Figure 25: Power Method Code

```python
def f(x):
    return np.sin(np.pi*x)*(1 + 2*np.cos(np.pi*x))

def g(t):
    return 0

def h(t):
    return 0


def FiniteDiffPDE(x, t, f, g, h, k, stepsX):
    delX = (x[1] - x[0])/ stepsX
    # (tn - t0)/stepsT <= h^2 / (2*alpha) --> (tn - t0)*((2.1*alpha)/h^2) = stepsT
    stepsT = int(np.ceil((t[1] - t[0]) / ((delX**2)/(2.1*k))))
    delT = (t[1] - t[0])/ stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((stepsT + 1, stepsX + 1))
    x_points = np.linspace(x[0], x[-1], stepsX + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)
    #Fill in boudary values
    for i in range(stepsT+1):
        u[i, 0] = g(delT*i)
        u[i, -1] = h(delT*i)
    #Fill initial value (t = 0 for all x)
    for j in range(stepsX+1):
        u[0, j] = f(j*delX)

    for i in range(1, stepsT+1):#Already know values at time 0 (f(x))
        for j in range(1, stepsX-1):#Already know values at x = 0 and x = L (g and h)
            u[i, j] = u[i - 1, j] + ((k*delT)/(delX**2)) * (u[i - 1, j + 1] - 2*u[i - 1, j] + u[i - 1, j - 1])

    return x_points, t_points, u
```

Figure 26: Code for FTCS Method

```
L = [0,1]
time = [0, 0.25]

fig1 = plt.figure(1, figsize=(12,14))
ax = plt.axes(projection = '3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig2 = plt.figure(2, figsize=(12,14))
ax = plt.axes(projection = '3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig3 = plt.figure(3, figsize=(12,14))
ax = plt.axes(projection = '3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

#Exact
fig4 = plt.figure(4, figsize=(12,14))
ax = plt.axes(projection = '3d')
x = np.linspace(0, 1, 100)
t = np.linspace(0, 0.25, 100)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T), cmap='gist_heat', alpha=0.8)
ax.set_title('u(x,t) = sin(pi*x)*e^((-pi^2)*t) + np.sin(2*pi*x)*e^(-4*(pi^2)*t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u(x,t)')
plt.show()
```

Figure 27: Part 3 Graphing Code

```
#Error
fig5 = plt.figure(5, figsize=(12,14))
ax = fig5.add_subplot(1, 3, 1, projection='3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
ax = fig5.add_subplot(1, 3, 2, projection='3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
ax = fig5.add_subplot(1, 3, 3, projection='3d')
x, t, u = FiniteDiffPDE(L, time, f, g, h, 1, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
```

Figure 28: Part 3 Error Graphing Code

```python
def f(x):
    return np.sin(np.pi*x)*(1 + 2*np.cos(np.pi*x))

def g(t):
    return 0

def h(t):
    return 0


def BTCS_SolvePDE(x, t, f, g, h, k, stepsX, stepsT):
    delX = (x[1] - x[0])/ stepsX
    delT = (t[1] - t[0])/ stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((stepsT + 1, stepsX + 1))
    x_points = np.linspace(x[0], x[-1], stepsX + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)
    #Fill in boudary values
    for i in range(stepsT+1):
        u[i, 0] = g(delT*i)
        u[i, -1] = h(delT*i)
    #Fill initial value (t = 0 for all x)
    for j in range(stepsX+1):
        u[0, j] = f(j*delX)

    r = -delT*k/(delX)**2
    #Create A matrix
    A = np.zeros((stepsX + 1, stepsX + 1))
    A[0, 0] = 1
    A[-1, -1] = 1
    for i in range(1, stepsX): #Fill in A matrix rows
        A[i, (i-1):(i+2)] = [r, 1 - 2*r, r]

    for i in range(1, stepsT+1):#Already know values at time 0 (f(x))
        u[i, :] = np.linalg.solve(A, u[i - 1, :]) #Solve Ax = b for unknown rows in u (temp along rod at a certain time)

    return x_points, t_points, u
```

Figure 29: BTCS Method Code

```
L = [0,1]
time = [0, 0.25]

fig1 = plt.figure(1, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 5, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig2 = plt.figure(2, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 10, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig3 = plt.figure(3, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 20, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

#Exact
fig4 = plt.figure(4, figsize=(12,7))
ax = plt.axes(projection = '3d')
x = np.linspace(0, 1, 100)
t = np.linspace(0, 0.25, 100)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T), cmap='gist_heat', alpha=0.8)
ax.set_title('u(x,t) = sin(pi*x)*e^((-pi^2)*t) + np.sin(2*pi*x)*e^(-4*(pi^2)*t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u(x,t)')
plt.show()
```

Figure 30: Part 4 Graphing Code

```
#Error
fig5 = plt.figure(5, figsize=(12,8))
ax = fig5.add_subplot(1, 3, 1, projection='3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 5, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
ax = fig5.add_subplot(1, 3, 2, projection='3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 10, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
ax = fig5.add_subplot(1, 3, 3, projection='3d')
x, t, u = BTCS_SolvePDE(L, time, f, g, h, 1, 20, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
```

Figure 31: Part 4 Error Graphing Code

```python
def f(x):
    return np.sin(np.pi*x)*(1 + 2*np.cos(np.pi*x))

def g(t):
    return 0

def h(t):
    return 0


def CrankNicolson(x, t, f, g, h, k, stepsX, stepsT):
    delX = (x[1] - x[0])/ stepsX
    delT = (t[1] - t[0])/ stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((stepsT + 1, stepsX + 1))
    x_points = np.linspace(x[0], x[-1], stepsX + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)
    #Fill in boudary values
    for i in range(stepsT+1):
        u[i, 0] = g(delT*i)
        u[i, -1] = h(delT*i)
    #Fill initial value (t = 0 for all x)
    for j in range(stepsX+1):
        u[0, j] = f(j*delX)

    r = (k*delT)/(2*(delX)**2) #Likely need to include k in this calc
    #Create A matrix
    A = np.zeros((stepsX + 1, stepsX + 1))
    A[0, 0] = 1
    A[-1, -1] = 1
    for i in range(1, stepsX): #Fill in A matrix rows
        A[i, (i-1):(i+2)] = [-r, 1 + 2*r, -r]

    for i in range(1, stepsT+1):#Already know values at time 0 (f(x))
        for j in range(1, stepsX-1):#Already know values at x = 0 and x = L (g and h)
            #Calc b in Ax = b
            u[i, j] = r*u[i - 1, j + 1] + (1-2*r)*u[i - 1, j] + r*u[i - 1, j - 1]
        u[i, :] = np.linalg.solve(A, u[i, :]) #Solve Ax = b for unknown rows in u (temp along rod at a certain time)

    return x_points, t_points, u
```

Figure 32: Crank-Nicolson Method Code

```
L = [0,1]
time = [0, 0.25]

fig1 = plt.figure(1, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 5, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig2 = plt.figure(2, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 10, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

fig3 = plt.figure(3, figsize=(12,7))
ax = plt.axes(projection = '3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 20, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u, cmap='gist_heat', alpha=0.8)
ax.set_title('Heat Eqn Solution x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('U(x,t)')
plt.show()

#Exact
fig4 = plt.figure(4, figsize=(12,7))
ax = plt.axes(projection = '3d')
x = np.linspace(L[0], L[1], 200)
t = np.linspace(time[0], time[1], 200)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T), cmap='gist_heat', alpha=0.8)
ax.set_title('u(x,t) = sin(pi*x)*e^((-pi^2)*t) + np.sin(2*pi*x)*e^(-4*(pi^2)*t)')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u(x,t)')
plt.show()
```

Figure 33: Part 5 Graphing Code

```
#Error
fig5 = plt.figure(5)
ax = fig5.add_subplot(1, 3, 1, projection='3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 5, 5)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 5')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')

ax = fig5.add_subplot(1, 3, 2, projection='3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 10, 10)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 10')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')

ax = fig5.add_subplot(1, 3, 3, projection='3d')
x, t, u = CrankNicolson(L, time, f, g, h, 1, 20, 20)
X, T = np.meshgrid(x, t)
ax.plot_surface(X, T, u - (np.sin(np.pi * X)*np.e**(-((np.pi)**2)*T) + np.sin(2 * np.pi * X)*np.e**(-4*((np.pi)**2)*T)), cmap='gist_heat', alpha=0.8)
ax.set_title('Error x-steps = 20')
ax.set_xlabel('x')
ax.set_ylabel('t')
ax.set_zlabel('u_est(x,t) - u_exact(x,t)')
plt.show()
```

Figure 34: Part 5 Error Graphing Code

References

[1] M. Bader, “Introduction to Scientific Computing.”

[2] “Parabolic partial differential equation,” Wikipedia, https://en.wikipedia.org/wiki/Parabolic_partial_differential_equation (accessed Apr. 15, 2024).

[3] Salih, “Classification of Partial Differential Equations and Canonical Forms.” Indian Institute of Space Science and Technology, Thiruvananthapuram, 2014

[4] “Partial differential equation,” Wikipedia, https://en.wikipedia.org/wiki/Partial_differential_equation (accessed Apr. 15, 2024).

[5] “Boundary value problem,” Wikipedia, https://en.wikipedia.org/wiki/Boundary_value_problem (accessed Apr. 16, 2024).

[6] “Dirichlet boundary condition,” Wikipedia, https://en.wikipedia.org/wiki/Dirichlet_boundary_condition (accessed Apr. 18, 2024).

[7] “Neumann boundary condition,” Wikipedia, https://en.wikipedia.org/wiki/Neumann_boundary_condition (accessed Apr. 18, 2024).

[8] J. Caulfield, “How to cite a Wikipedia article: APA, MLA & Chicago,” Scribbr, https://www.scribbr.com/citing-sources/how-to-cite-wikipedia/ (accessed Apr. 18, 2024).