

A dark blue vertical bar runs down the left side of the page. A blue arrow-shaped banner points to the right from this bar, containing the date. In the bottom-left corner, several thin, curved lines in dark blue and light gray sweep upwards and to the right.

6/12/2024

# Project 4 Report

MATH 453

Cameron Robinson

## Table of Contents

Table of Figures .....	1
Project 4 .....	2
Part 1 .....	2
Part 2 .....	3
Part 3 .....	5
Part 4 .....	6
Part 6 .....	7
Appendix .....	11

## Table of Figures

Figure 1: PDE for Part 1 and 2 .....	2
Figure 2: Polar Coordinate Conversions .....	2
Figure 3: 2D Wave Finite Difference Cartesian Update Equation.....	2
Figure 4: Solution Snapshots for Part 1.....	3
Figure 5: 2D Wave Finite Difference Polar Update Equation .....	4
Figure 6: Solution Snapshots for Part 2.....	4
Figure 7: 1D Acoustics System for Part 3 and 4.....	5
Figure 8: Update Equation and Stencil for 1D Wave Finite Differences.....	5
Figure 9: Solution Snapshots for Part 3.....	6
Figure 10: First Order Finite Volume Update Equation .....	6
Figure 11: Solution Snapshots for Part 4.....	7
Figure 12: Parabolic PDE for Part 6 .....	7
Figure 13: Solution to Part 6 PDE .....	7
Figure 14: FTCS Update Equation for Part 6.....	7
Figure 15: Numerical Solution Snapshots for Part 6 .....	8
Figure 16: Exact Solution Snapshots for Part 6 .....	9
Figure 17: Error Graphs .....	10
Figure 19: Method Code for Part 1 .....	11
Figure 20: Graphing Code for Part 1.....	11
Figure 21: Code for Part 2 .....	12
Figure 22: Part 3 Code.....	13
Figure 23: Code for Part 4 .....	14
Figure 24: Method Code for Part 6 .....	15
Figure 25: Graphing Code for Part 6.....	16

## Project 4

### Part 1

The objective of part one was to use finite differences to solve a cartesian 2D wave PDE with a circular fixed boundary. The PDE is shown in Figure 1; the area is a disk with a 3-unit radius and initial wave speed of two ( $c = 2$ ). The BCs and IVs are given in polar coordinates, but can be converted to cartesian using the equations in Figure 2.

*2D Wave Equation for Part 1 and Part 2:*

$$u_{tt} = c^2(u_{xx} + u_{yy})$$

$$u(x, y, 0) = e^{-4r^2}$$

$$u(r = 3, 0) = 0$$

$$u_t(x, y, 0) = 0$$

$$|r| < 3$$

$$0 \leq t \leq 6$$

*Figure 1: PDE for Part 1 and 2*

*Polar Coordinate Conversions*

$$r^2 = x^2 + y^2$$

$$\tan(\theta) = \frac{y}{x}$$

$$x = r \cos(\theta)$$

$$y = r \sin(\theta)$$

*Figure 2: Polar Coordinate Conversions*

Figure 3 shows the update equation for the finite difference method implemented to solve the PDE. There is only one unknown value so each point can be solved for on its own. The method has  $O((\Delta x)^2) + O((\Delta y)^2) + O((\Delta t)^2)$  error. The code to implement this method is shown in Figure 18 in the appendix.

$$u(x, y, t + \Delta t) = 2u(x, y, t) - u(x, y, t - \Delta t) + c^2(\Delta t)^2 \left( \frac{u(x + \Delta x, y, t) - 2u(x, y, t) + u(x - \Delta x, y, t)}{(\Delta x)^2} + \frac{u(x, y + \Delta y, t) - 2u(x, y, t) + u(x, y - \Delta y, t)}{(\Delta y)^2} \right)$$

*Figure 3: 2D Wave Finite Difference Cartesian Update Equation*

Figure 4 shows snapshots of the numerical solution, which is a wave that propagates in a ring from the center. Since the method was implemented with cartesian coordinates, the boundary of the solution is jagged, though this could be improved with smaller step-sizes. The code to create the animation is shown in Figure 19.

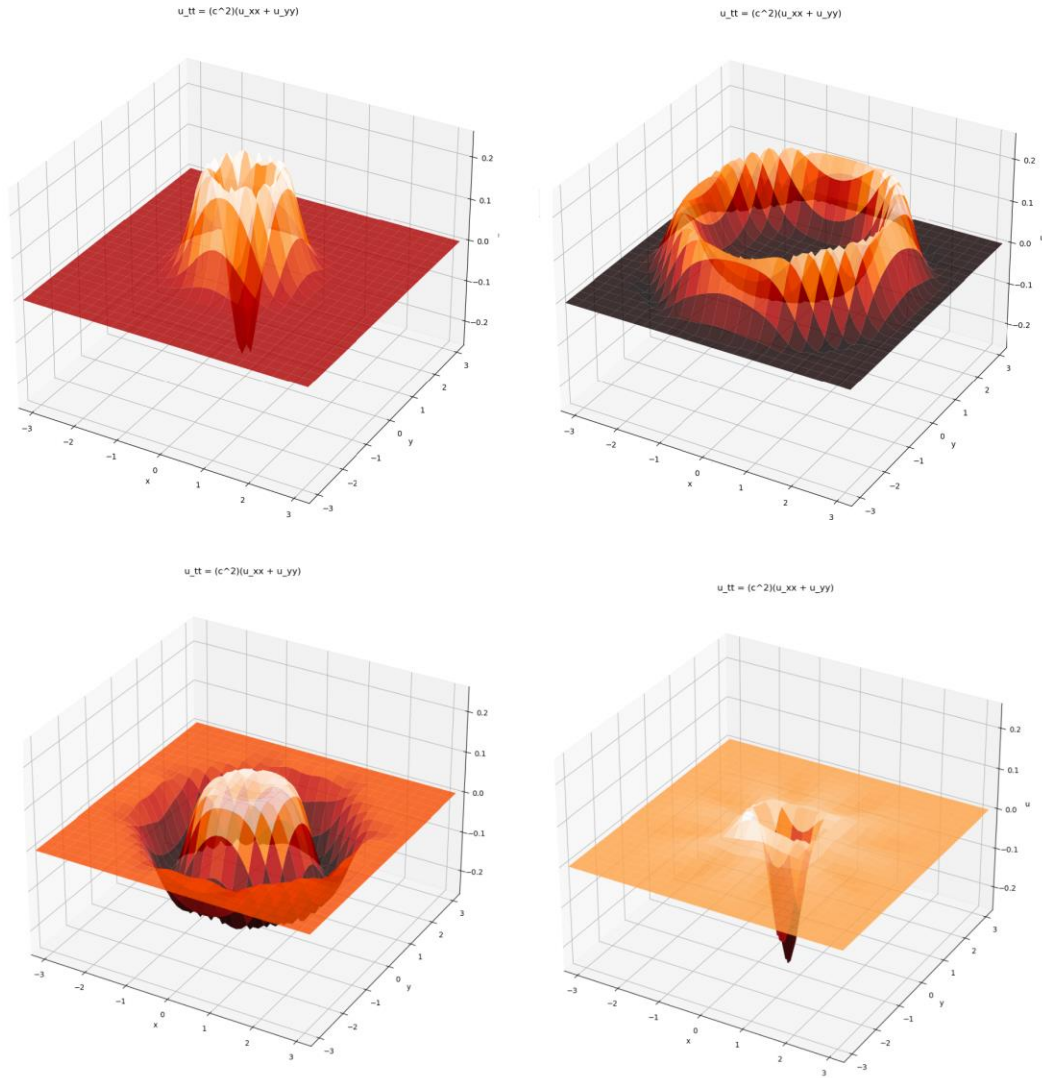


Figure 4: Solution Snapshots for Part 1

## Part 2

The objective of part two was to solve the same PDE from part one, but it was first converted to polar coordinates and then solved with finite differences. The update equation for this method is shown in Figure 5. Since the PDE is radially symmetric, the theta components were removed from the equation since the solution did not depend on it. Since this equation is undefined at  $r = 0$ , the method only calculates with values of  $r$  near zero (e.g., 0.000000001).

$$u(r, \theta, t + \Delta t) = 2u(r, \theta, t) - u(r, \theta, t - \Delta t) + c^2(\Delta t)^2 \left( \frac{u(r+\Delta r, \theta, t) - 2u(r, \theta, t) + u(r-\Delta r, \theta, t)}{(\Delta r)^2} + \frac{u(r+\Delta r, \theta, t) - u(r-\Delta r, \theta, t)}{2r(\Delta r)} \right)$$

$r$  cannot be 0

*Note: The PDE is radially symmetric for this problem, so the  $\theta$  components were removed.*

Figure 5: 2D Wave Finite Difference Polar Update Equation

Figure 6 shows snapshots of the numerical solution. The solution is the same as in part one, but the boundary is much smoother even though the number of steps is the same. Although the solution is not solved at the center, that does not seem to affect it much at all. The code implementation for this part is shown in Figure 20.

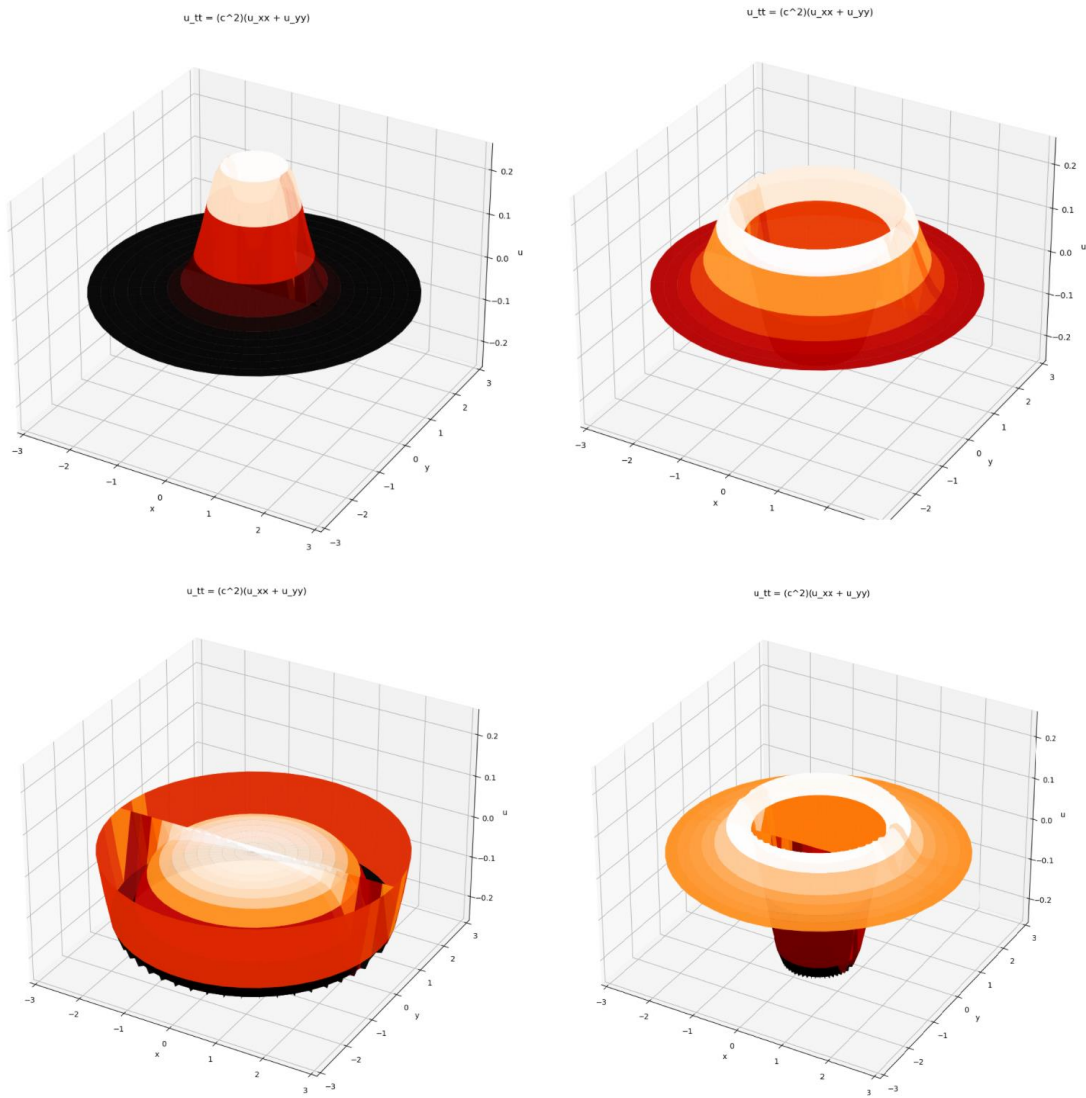


Figure 6: Solution Snapshots for Part 2

### Part 3

The objective of part three was to solve the 1D acoustics system illustrated in Figure 7. To solve the system, the pressure and velocity components were eliminated individually to create two separate PDEs with the same form as the normal 1D wave equation. These equations were then solved using finite differences. The update equation and five-point stencil for this method is shown in Figure 8. Since only one initial condition was given for the pressure and velocity solutions, the original system was used to solve for  $p_t$  and  $v_t$  at  $t=0$  for all  $x$  ( $p_t = -kv_x$ ,  $v_t = (1/\rho)p_x$ ).

*1D Acoustics System for Part 3 and 4*

$$p_t + kv_x = 0$$

$$v_t + \frac{p_x}{\rho} = 0$$

$$x = [0,20], t = [0,6]$$

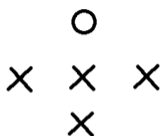
$$p(x, 0) = 2 \text{ for } x = [8,12], \text{ otherwise } 0$$

$$k = 4, \rho = 1, c = 2$$

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

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

*Figure 7: 1D Acoustics System for Part 3 and 4*



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

*Figure 8: Update Equation and Stencil for 1D Wave Finite Differences*

Figure 9 shows snapshots of the numerical solution for the acoustics system. The red line is the solution for pressure, and the black line is the solution for velocity. Since the PDE was solved with outflow boundary conditions, the solution just propagates off the screen instead of reflecting like the fixed-boundary solutions in parts one and two. Likely due to the sharp corners in both solutions, there is a lot of numerical dispersion in this solution, especially in the pressure solution. This is one of the major limitations of finite-difference methods. The code to implement this method and create the animation is shown in Figure 21.

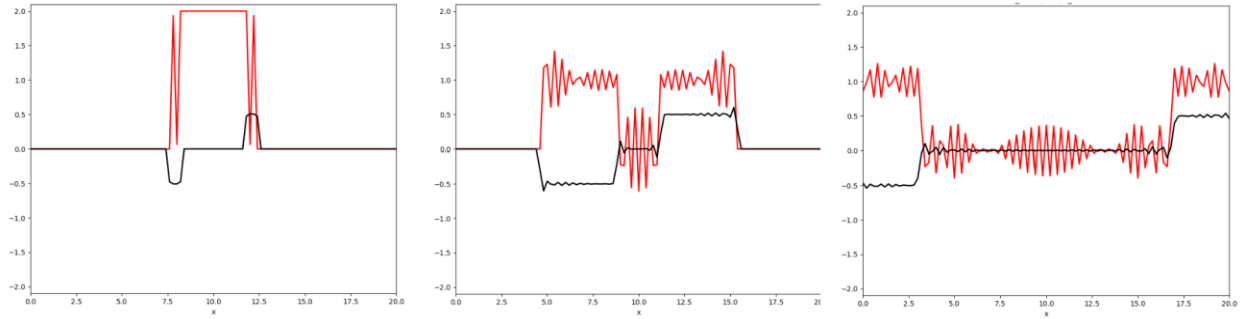


Figure 9: Solution Snapshots for Part 3

## Part 4

The objective of part four was to solve the same 1D acoustics system in part three using a first-order finite volume method. The update equations for this method are shown in Figure 11, where  $p$  is the pressure,  $v$  is the velocity,  $c$  is the wave speed ( $\sqrt{k/\rho}$ ), and  $z$  is the impedance ( $\rho \cdot c$ ). Although they are constant in this problem,  $c$ ,  $\rho$ ,  $k$ , and  $z$  could be functions of  $x$ . The finite volume method works by solving Riemann problems at cell interfaces (the  $x$ -values, at least in my implementation); this required using the midpoint values of the cells ( $p$ ,  $z$ ,  $c$ , etc.) to the right and left of the interface to solve for the solution at the next time-step.

### First – Order Finite Volume

$$\alpha = \frac{(p(t, x + \Delta x) - p(t, x) + z(x + \Delta x) (v(t, x + \Delta x) - v(t, x)))}{z(x) + z(x + \Delta x)}$$

$$\beta = \frac{(p(t, x) - p(t, x - \Delta x) + z(x) (v(t, x) - v(t, x - \Delta x)))}{z(x) + z(x - \Delta x)}$$

$$p(t + \Delta t, x) = p(x, t) - \frac{\Delta t}{\Delta x} (c(x) \beta z(x + \Delta x) + c(x) \alpha z(x))$$

$$v(t + \Delta t, x) = v(x, t) - \frac{\Delta t}{\Delta x} (c(x) \beta - c(x) \alpha)$$

Figure 10: First Order Finite Volume Update Equation

Figure 12 shows snapshots of the numerical solution calculated using the finite volume method. As before, the red and black lines are the solutions for pressure and velocity, respectively. The solutions have the same shape and values as the one calculated with finite differences, only there is little to no numerical dispersion, which is one of the benefits of the finite volume method.

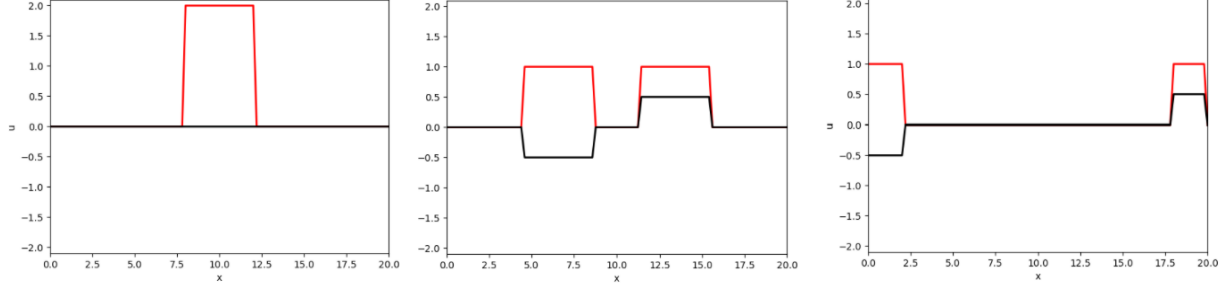


Figure 11: Solution Snapshots for Part 4

## Part 6

The objective of part six was to solve the 2D parabolic PDE shown in Figure 13 with some chosen method and then compare the numerical solution to the exact solution given in Figure 14. The method chosen for this part was FTCS. Similar to the CFL condition for the finite difference and finite volume methods from the previous parts, this method also has a restriction on the size of the time-steps ( $\Delta t \leq$

$$\frac{1}{2\alpha(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2})}, \alpha = 1).$$

*Parabolic PDE for Part 6*

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

$$x = [0, 1], y = [0, 1], t = [0, 0.1]$$

$$u(x, y, 0) = 100 \text{ if } y \leq x, \text{ otherwise } 0$$

Figure 12: Parabolic PDE for Part 6

$$u(x, y, t) = 100 \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} C_{n,m} \sin(n\pi x) \sin(m\pi y) \exp(-\pi^2(n^2 + m^2)t),$$

$$\text{Where } C_{n,m} = \frac{4((-1)^n((-1)^m - 1)n^2 + ((-1)^n - 1)m^2)}{[nm(n^2 - m^2)\pi^2]} \text{ if } n \neq m$$

$$C_{n,m} = \frac{2((-1)^n - 1)^2}{[n^2\pi^2]} \text{ if } n = m$$

Figure 13: Solution to Part 6 PDE

Figure 14 shows the update equation for the FTCS method. This method has  $O((\Delta t)) + O((\Delta x^2)) + O((\Delta y^2))$  error. The code implementation for this method is shown in Figure 23 in the appendix.

*FTCS 2D Parabolic PDE Update Equation*

$$u(x, y, t + \Delta t) = u(x, y, t) + \Delta t \left( \frac{u(x+\Delta x, y, t) - 2u(x, y, t) + u(x-\Delta x, y, t)}{(\Delta x)^2} + \frac{u(x, y+\Delta y, t) - 2u(x, y, t) + u(x, y-\Delta y, t)}{(\Delta y)^2} \right)$$

Figure 14: FTCS Update Equation for Part 6



Figure 15 shows snapshots of the numerical solution at sequential time-steps; the solution approaches zero as time increases. The result is very similar to the exact solution graphs shown in Figure 16. The one major difference between the two is their initial conditions. For some reason, the exact solution's initial condition has a bunch of ripples all over the surface. This may be due to calculating the exact solution with too few terms from the infinite series. At later times, the solutions both approach zero.

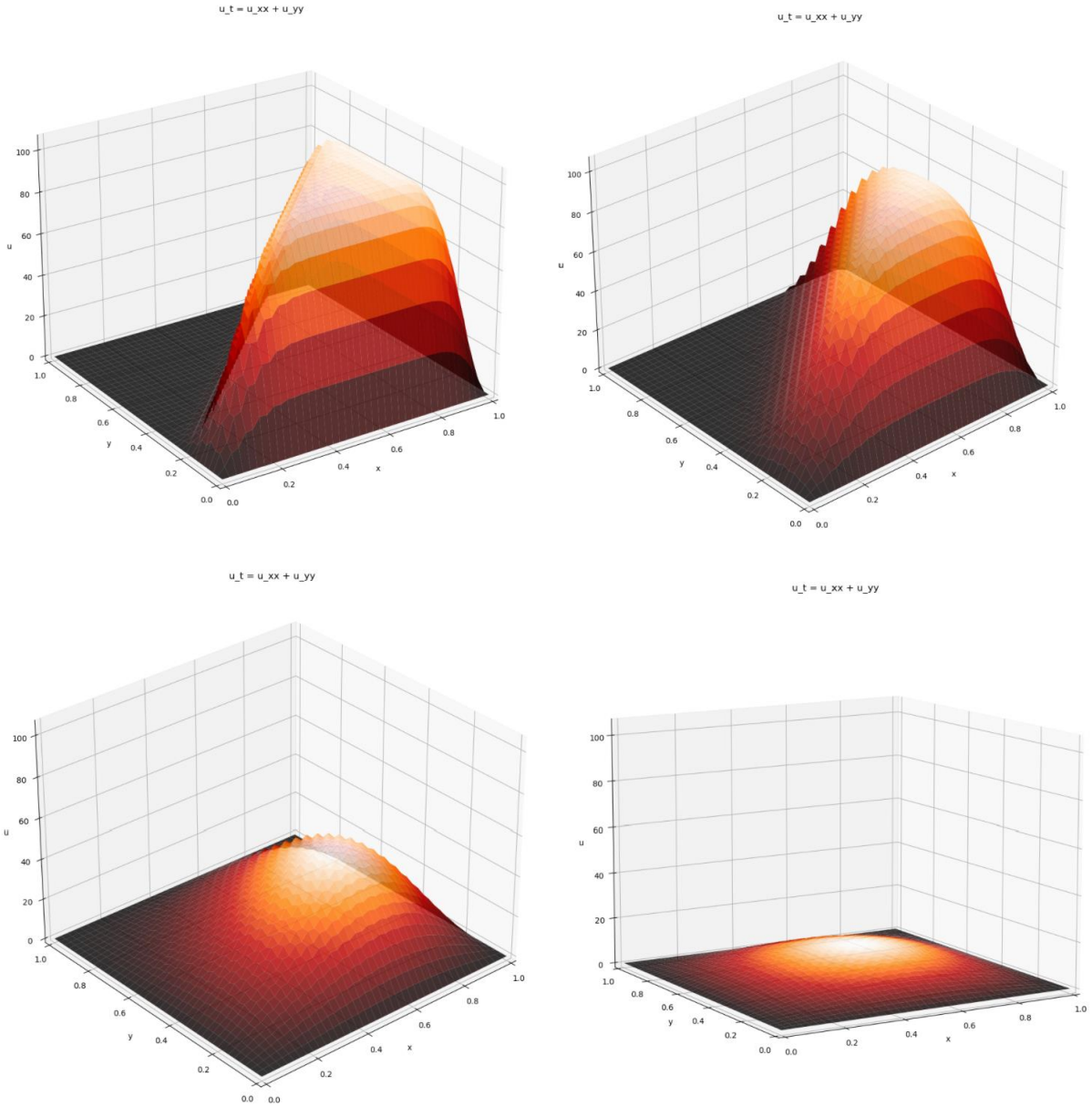


Figure 15: Numerical Solution Snapshots for Part 6

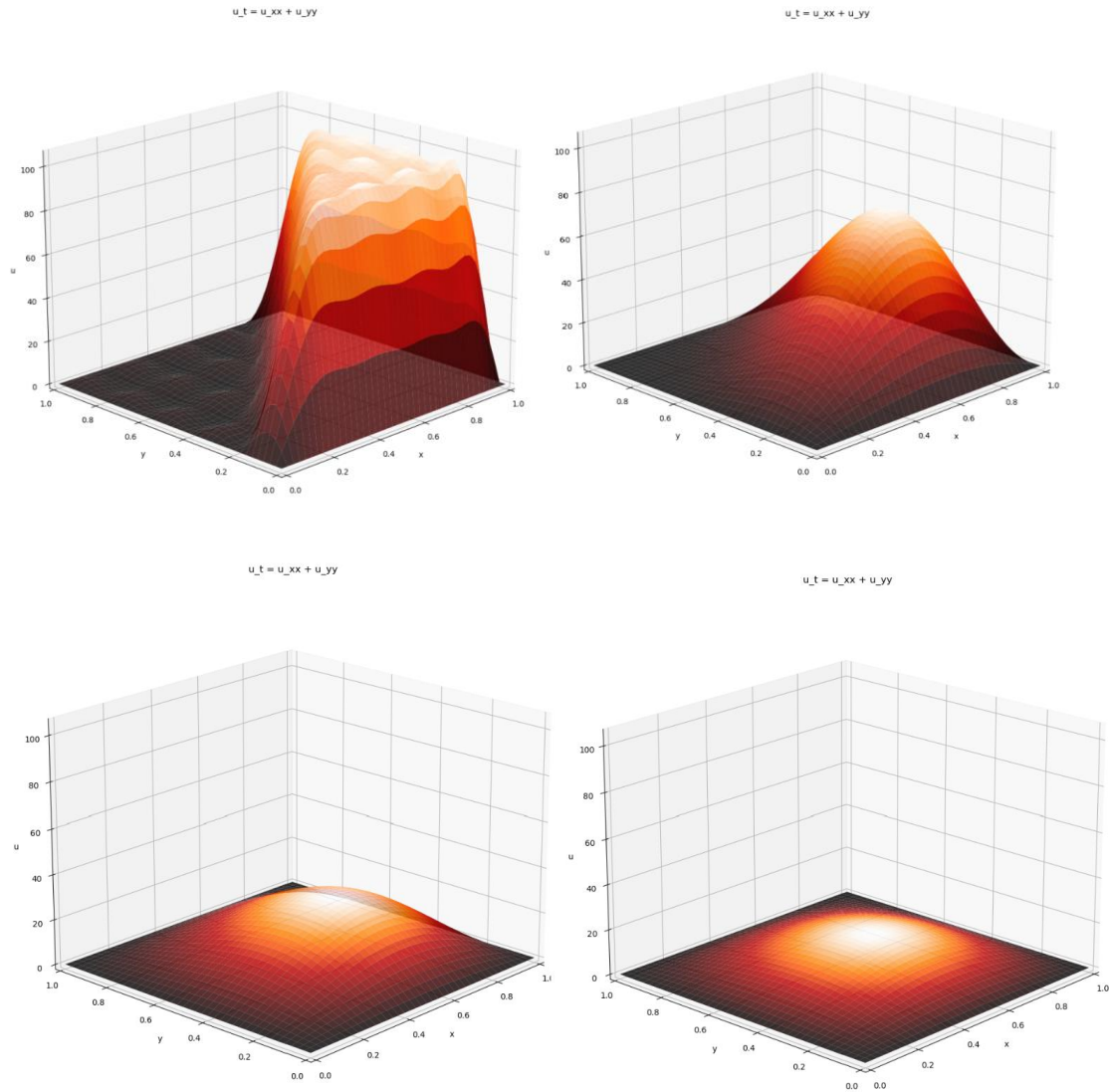


Figure 16: Exact Solution Snapshots for Part 6

Figure 17 contains snapshots of the error between the numerical and exact solutions above. As expected, the error is initially very high since the solutions' initial values are different. Similarly, as time increases, the error decreases, since both solutions converge towards zero. The code to create the animations for the solutions and error is shown in Figure 24.

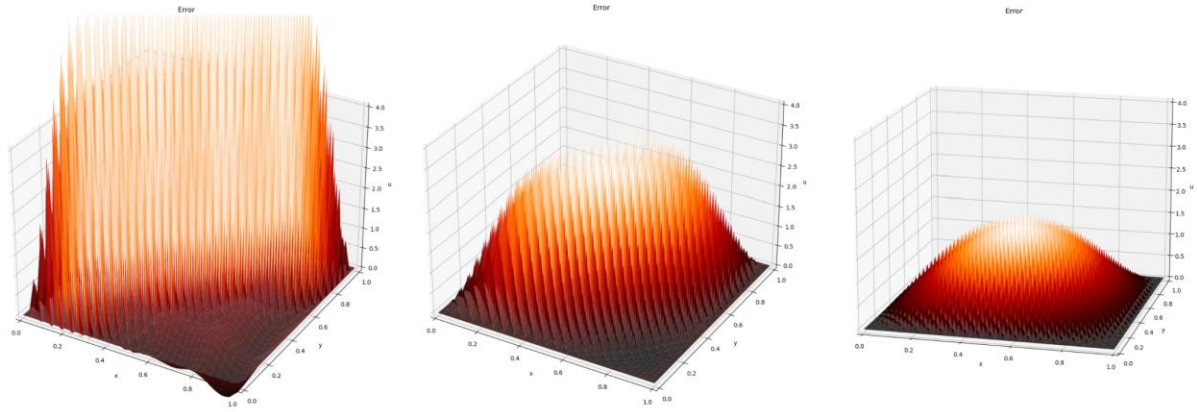


Figure 17: Error Graphs

## Appendix

```
def f(x, y):
    return np.select([(x**2 + y**2 < 3)], [e**(-4*(x**2 + y**2))], 0)

def F(x, y):
    return 0

def p(x, t):
    return 0

def r(x, t):
    return 0

def g(y, t):
    return 0

def h(y, t):
    return 0

def FiniteDiff2D_Wave(x, y, t, f, F, p, r, g, h, c, steps):
    delX = (x[1] - x[0])/ steps
    delY = (y[1] - y[0])/ steps
    #Solve for time-step restraint (CFL condition)
    stepsT = int(np.ceil(2*c*(t[1] - t[0])/((delX))))
    delT = (t[1] - t[0])/ stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((steps + 1, steps + 1, stepsT + 1))
    x_points = np.linspace(x[0], x[-1], steps + 1)
    y_points = np.linspace(y[0], y[-1], steps + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)

    X, Y = np.meshgrid(x_points, y_points)
    u[:, :, 0] = f(X, Y)
    u[:, :, 1] = u[:, :, 0] + F(X, Y)*delT

    for j in range(1, stepsT):
        for k in range(1, steps):
            for i in range(1, steps):
                if((delX*i + x[0])**2 + (delY*k + y[0])**2 >= 9):
                    u[i, k, j] = 0
                    u[i, k, j+1] = 2*u[i, k, j] - u[i, k, j-1] + ((c**2)*(delT**2))*((u[i+1, k, j] - 2*u[i, k, j] + u[i-1, k, j])/((delX**2) + (u[i, k+1, j] - 2*u[i, k, j] + u[i, k-1, j])/((delY**2))

    return x_points, y_points, t_points, u
```

Figure 18: Method Code for Part 1

```
L = [-3.2, 3.2]
W = [-3.2, 3.2]
time = [0, 6]
c = 2

x, y, t, u = FiniteDiff2D_Wave(L, W, time, f, F, p, r, g, h, c, 50)
X, Y = np.meshgrid(x, y)

#Animation
fig1 = plt.figure(1, figsize=(12,14))
ax = plt.axes(projection = '3d')
line = [ax.plot_surface(X, Y, u[:, :, 0], color='0.75', rstride=1, cstride=1)]
ax.set_xlim3d([L[0], L[-1]])
ax.set_ylim3d([W[0], W[-1]])
ax.set_zlim3d([-0.25, 0.25])
ax.set_title('u_tt = (c^2)(u_xx + u_yy)')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')

def draw_wave(num, u, line):
    line[0].remove()
    line[0] = ax.plot_surface(X, Y, u[:, :, num], cmap='gist_heat', alpha=0.8)

anim = animation.FuncAnimation(fig1, draw_wave, len(u[0, 0, :]), fargs=(u, line))
# afile = r"Proj3P5Numerical50.gif"
# writergif = animation.PillowWriter(fps=4)
plt.show()
```

Figure 19: Graphing Code for Part 1

```

def f(r):
    return np.select([abs(r) <= 3], [e**(-4*r**2)], 0)

def F(x, y):
    return 0

def p(x, t):
    return 0

def q(x, t):
    return 0

def g(y, t):
    return 0

def h(y, t):
    return 0

def FiniteDiff2D_Wave_Polar(r, o, t, f, F, p, q, g, h, c, steps):
    delR = (r[1] - r[0]) / steps
    delO = (o[1] - o[0]) / steps
    #Solve for time-step restraint (CFL condition)
    stepsT = int(np.ceil(2*c*(t[1] - t[0])/(delR)))
    delT = (t[1] - t[0]) / stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((steps + 1, steps + 1, stepsT + 1))
    r_points = np.linspace(r[0], r[-1], steps + 1)
    o_points = np.linspace(o[0], o[-1], steps + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)

    R, O = np.meshgrid(r_points, o_points)
    u[:, :, 0] = f(R)
    u[:, :, 1] = u[:, :, 0] + F(R, O)*delT
    #Fill in boundary values
    u[0, :, :] = g(o_points, t_points)
    u[-1, :, :] = h(o_points, t_points)
    u[:, 0, :] = p(r_points, t_points)
    u[:, -1, :] = q(r_points, t_points)

    for j in range(1, stepsT):
        for i in range(1, steps):
            for k in range(1, steps):
                rp = r_points[k]
                if(abs(rp) <= 0.0000000000000001): #Do not calculate at r = 0
                    rp = 0.0000000000000001
                #Ignores theta component since radially symmetric
                u[i, k, j+1] = 2*u[i, k, j] - u[i, k, j-1] + ((c**2)*(delT**2))*((u[i, k+1, j] - u[i, k-1, j])/(2*rp*delR) + (u[i, k+1, j] - 2*u[i, k, j] + u[i, k-1, j])/(delR**2))
    return r_points, o_points, t_points, u

radius = [-3, 3]
theta = [0, np.pi * 2]
time = [0, 6]
c = 2

r, o, t, u = FiniteDiff2D_Wave_Polar(radius, theta, time, f, F, p, q, g, h, c, 50)
R, O = np.meshgrid(r, o)
x = R*np.cos(O)
y = R*np.sin(O)

#Animation
fig1 = plt.figure(1, figsize=(12,14))
ax = plt.axes(projection = '3d')
line = [ax.plot_surface(x, y, u[:, :, 0], color='0.75', rstride=1, cstride=1)]
ax.set_xlim3d([-3,3])
ax.set_ylim3d([-3,3])
ax.set_zlim3d([-0.25,0.25])
ax.set_title('u_tt = (c^2)(u_xx + u_yy)')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')

def draw_wave(num, u, line):
    line[0].remove()
    line[0] = ax.plot_surface(x, y, u[:, :, num], cmap='gist_heat', alpha=0.8)

anim = animation.FuncAnimation(fig1, draw_wave, len(u[0,0,:]), fargs=(u, line))
plt.show()

```

Figure 20: Code for Part 2

```

def initial_v(x):
    return 0
def initial_p(x):
    return np.select([(x >= 8)*(x <= 12)], [2], 0)
def F(x, delX):
    dx = np.zeros((len(x)))
    dx[0] = ((-3/2)*x[0] + 2*x[1] - 0.5*x[2]) / delX
    for i in range(1, len(x)-1):
        dx[i] = (x[i+1] - x[i-1])/(2*delX)
    dx[-1] = ((3/2)*x[-1] - 2*x[-2] + 0.5*x[-3]) / delX
    return dx

def FiniteDiffPDE_Wave(x, t, ip, iv, rho, k, c, stepsX):
    delX = (x[1] - x[0]) / stepsX
    #Solve for time-step restraint (CFL condition)
    stepsT = int(np.ceil(1.01*c*(t[1] - t[0])/(delX)))
    delT = (t[1] - t[0]) / stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((stepsT + 1, stepsX + 3, 2))
    x_points = np.linspace(x[0], x[-1], stepsX + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)
    #Fill initial values for first two time steps using initial position and velocity
    u[0, 1:-1, 0] = ip(x_points)
    u[0, 1:-1, 1] = iv(x_points)
    u[1, 1:-1, 0] = u[0, 1:-1, 0] - k*F(u[0, 1:-1, 1], delX)*delT
    u[1, 1:-1, 1] = u[0, 1:-1, 1] - (1/rho)*F(u[0, 1:-1, 0], delX)*delT

    for i in range(1, len(u[:,0,0]) - 1):
        u[i+1,0,0] = u[i,1,0]
        u[i+1,-1,0] = u[i,-2,0]
        u[i+1,0,1] = u[i,1,1]
        u[i+1,-1,1] = u[i,-2,1]
        for j in range(1, len(u[0,:,0]) - 1):
            u[i + 1, j, 0] = 2*u[i, j, 0] - u[i - 1, j, 0] + (((c**2)*(delT**2))/(delX**2)) * (u[i, j + 1, 0] - 2*u[i, j, 0] + u[i, j - 1, 0])
            u[i + 1, j, 1] = 2*u[i, j, 1] - u[i - 1, j, 1] + (((c**2)*(delT**2))/(delX**2)) * (u[i, j + 1, 1] - 2*u[i, j, 1] + u[i, j - 1, 1])
    return x_points, t_points, u[:,1:-1,: ]

L = [0,20]
time = [0, 6]
k = 4
rho = 1
c = (k/rho)**(1/2)
fig1 = plt.figure(1, figsize=(12,14))
ax = plt.axes(projection = '3d')
x, t, u = FiniteDiffPDE_Wave(L, time, initial_p, initial_v, rho, k, c, 100)
X, T = np.meshgrid(x, t)

#Animation
fig1 = plt.figure(1)
ax = plt.axes(xlim=(L[0],L[1]),ylim=(-2.1,2.1))
ax.set_xlabel('x')
ax.set_ylabel('P')
ax.set_title('P_tt = (c^2)P_xx')

line = [[],[]]
line[0], = ax.plot([], [], linewidth=2.0,color='red', label='Pressure')
line[1], = ax.plot([], [], linewidth=2.0,color='black', label='Velocity')

def draw_wave(T):
    line[0].set_data(x,u[T,:,0])
    line[1].set_data(x,u[T,:,1])

anim = animation.FuncAnimation(fig1, draw_wave, frames=len(u[:,0,0]))

```

Figure 21: Part 3 Code

```

def initial_v(x):
    return 0
def initial_p(x):
    return np.select([(x >= 8)*(x <= 12)], [2], 0)

def rho(x):
    return 1
def k(x):
    return 4
def speed(x):
    return np.sqrt(rho(x)*k(x))

def Wave1D_FiniteVolumes(x, t, k, rho, c, ip, iv, stepsX, c_max):
    delX = (x[1] - x[0])/ stepsX
    #Solve for time-step restraint (CFL condition)
    stepsT = int(np.ceil(c_max*(t[1] - t[0])/(delX)))
    delT = (t[1] - t[0])/ stepsT

    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((stepsT + 1, stepsX + 3, 2))#+2 in x for ghost cells
    x_points = np.linspace(x[0], x[-1], stepsX + 1)
    xm_points = np.linspace(x[0] - delX/2, x[-1] + delX/2, stepsX + 2)#Midpoints
    t_points = np.linspace(t[0], t[-1], stepsT + 1)

    #ICs
    u[0, 1:-1, 0] = ip(x_points)
    u[0, 1:-1, 1] = iv(x_points)

    for n in range(len(t_points)-1):
        #Set Ghost Cells
        u[n, 0, 0] = u[n, 1, 0]
        u[n, -1, 0] = u[n, -2, 0]
        u[n, 0, 1] = u[n, 1, 1]
        u[n, -1, 1] = u[n, -2, 1]
        for i in range(len(x_points)):
            pd = u[n, i+1, 0] - u[n, i, 0]
            vd = u[n, i+1, 1] - u[n, i, 1]
            #May be wrong, but does not matter for this problem since they are constants
            ci = c(xm_points[i])
            zi = rho(xm_points[i]) * ci
            ci1 = c(xm_points[i+1])
            zi1 = rho(xm_points[i+1]) * ci1
            #print(str(ci) + ", " + str(ci1) + ", " + str(zi) + ", " + str(zi1))
            a = (-pd + vd*zi1)/(zi + zi1)
            #My issue was taking a and B from the same point, when B should have been from i-1/2, not i+1/2 like a
            B = ((u[n, i, 0] - u[n, i-1, 0]) + (u[n, i, 1] - u[n, i-1, 1])*zi)/(zi + zi1)
            u[n+1, i, 0] = u[n, i, 0] - (delT/delX)*(ci*B*zi1 + ci*a*zi)
            u[n+1, i, 1] = u[n, i, 1] - (delT/delX)*(ci*B - ci*a)
    return x_points, t_points, u[:,1:-1,:]
```

```

steps = 100
L = [0, 20]
time = [0, 6]
c_max = 2

x, t, u = Wave1D_FiniteVolumes(L, time, k, rho, speed, initial_p, initial_v, steps, c_max)

#Animation
fig3 = plt.figure(3)
ax = plt.axes(xlim=(L[0],L[1]),ylim=(-2.1,2.1))
ax.set_xlabel('x')
ax.set_ylabel('u')
ax.set_title('U_tt = (c^2)U_xx')

line = [[],[]]
line[0], = ax.plot([], [], linewidth=2.0,color='red', label='Numerical Solution')
line[1], = ax.plot([], [], linewidth=2.0,color='black', label='Initial Condition')

def draw_wave(T):
    line[0].set_data(x,u[T, :, 0])
    line[1].set_data(x,u[T, :, 1])

anim = animation.FuncAnimation(fig3, draw_wave, frames=len(u[:,0,0]))

```

Figure 22: Code for Part 4

```

def f(x, y):
    return np.select([y <= x], [100], 0)

def p(x, t):
    return 0

def r(x, t):
    return 0

def g(y, t):
    return 0

def h(y, t):
    return 0

def FTCS_2DParabolicPDE(x, y, t, f, p, r, g, h, a, steps):
    delX = (x[1] - x[0])/ steps
    delY = (y[1] - y[0])/ steps
    #Solve for time-step restraint (CFL condition)
    stepsT = int(np.ceil(2*a*(t[1] - t[0])*(1/(delX**2) + 1/(delY**2))))
    delT = (t[1] - t[0])/ stepsT
    #Set up solution array. Columns are delta-x, rows are delta-t
    u = np.zeros((steps + 1, steps + 1, stepsT + 1))
    x_points = np.linspace(x[0], x[-1], steps + 1)
    y_points = np.linspace(y[0], y[-1], steps + 1)
    t_points = np.linspace(t[0], t[-1], stepsT + 1)

    X,Y = np.meshgrid(x_points, y_points)
    u[:, :, 0] = f(X, Y)
    #Fill in boundary values
    u[0, :, :] = g(y_points, t_points)
    u[-1, :, :] = h(y_points, t_points)
    u[:, 0, :] = p(x_points, t_points)
    u[:, -1, :] = r(x_points, t_points)

    for j in range(0, stepsT-1):
        for k in range(1, steps):
            for i in range(1, steps):
                u[i, k, j+1] = u[i,k,j] + (delT)*((u[i+1,k,j] - 2*u[i,k,j] + u[i-1,k,j])/(delX**2) + (u[i,k+1,j] - 2*u[i,k,j] + u[i,k-1,j])/(delY**2))
    return x_points, y_points, t_points, u

```

Figure 23: Method Code for Part 6



```

L = [0, 1]
W = [0, 1]
time = [0, 0.1]
a = 1

X, y, t, u = FTCS_2DParabolicPDE(L, W, time, f, p, r, g, h, a, 40)
X, Y = np.meshgrid(x, y)

#Animation
fig1 = plt.figure(1, figsize=(12,14))
ax = plt.axes(projection = '3d')
line = [ax.plot_surface(X, Y, u[:, :, 0], color='0.75', rstride=1, cstride=1)]
ax.set_xlim3d([L[0], L[-1]])
ax.set_ylim3d([W[0], W[-1]])
ax.set_zlim3d([0, 105])
ax.set_title('u_t = u_xx + u_yy')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')
def draw_wave(num, u, line):
    line[0].remove()
    line[0] = ax.plot_surface(X, Y, u[:, :, num], cmap='gist_heat', alpha=0.8)

# anim = animation.FuncAnimation(fig1, draw_wave, len(u[0,0,:]), fargs=(u, line))
# plt.show()

nn=10
stepsX = 40
stepsY = 40
delX = (L[1] - L[0])/stepsX
delY = (W[1] - W[0])/stepsY
stepsT = int(np.ceil(2*a*(time[1] - time[0])*(1/(delX**2) + 1/(delY**2))))
X = np.linspace(L[0], L[1], stepsX + 1)
Y = np.linspace(W[0], W[1], stepsY + 1)
T = np.linspace(time[0], time[1], stepsT + 1)

x,y,t = np.meshgrid(X,Y,T)

ue=np.zeros((stepsY + 1, stepsX + 1, stepsT + 1))
ve=np.zeros((stepsY + 1, stepsX + 1, stepsT + 1))

for n in range(1,nn):
    for m in range(1,nn):
        if(n != m):
            C = 4*((-1)**n)*((-1)**m) - 1)*n**2 + (((-1)**n) - 1)*m**2)/(n*m*(n**2 - m**2)*(np.pi)**2)
        else:
            C = (2*((-1)**n) - 1)**2)/((n**2)*(np.pi)**2)
        ve = C*np.sin(n*np.pi*x)*np.sin(m*np.pi*y)*e**(-(np.pi)**2*(n**2 + m**2)*t) + ve
    ue = 100 * ve
X1, Y1 = np.meshgrid(X,Y)
fig2 = plt.figure(2, figsize=(12,14))
ax = plt.axes(projection = '3d')
line2 = [ax.plot_surface(X1, Y1, ue[:, :, 0], color='0.75', rstride=1, cstride=1)]
ax.set_xlim3d([L[0], L[-1]])
ax.set_ylim3d([W[0], W[-1]])
ax.set_zlim3d([0, 105])
ax.set_title('u_t = u_xx + u_yy')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')

def draw_wave2(num, ue, line):
    line2[0].remove()
    line2[0] = ax.plot_surface(X1, Y1, ue[:, :, num], cmap='gist_heat', alpha=0.8)

# anim = animation.FuncAnimation(fig2, draw_wave2, len(ue[0,0,:]), fargs=(ue, line2))
# plt.show()

#Error
#Error is initially very large, but it seems that the IC of the exact is not the same.
err = u - ue
fig3 = plt.figure(3, figsize=(12,14))
ax = plt.axes(projection = '3d')
line3 = [ax.plot_surface(X1, Y1, err[:, :, 0], color='0.75', rstride=1, cstride=1)]
ax.set_xlim3d([L[0], L[-1]])
ax.set_ylim3d([W[0], W[-1]])
ax.set_zlim3d([0, 4])
ax.set_title('Error')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')

def draw_wave3(num, err, line3):
    line3[0].remove()
    line3[0] = ax.plot_surface(X1, Y1, err[:, :, num], cmap='gist_heat', alpha=0.8)

anim = animation.FuncAnimation(fig3, draw_wave3, len(err[0,0,:]), fargs=(err, line3))
plt.show()

```

Figure 24: Graphing Code for Part 6