東京工業大学
Tokyo Institute of Technology

PARTIAL DIFFERENTIAL EQUATIONS FOR SCIENCE AND ENGINEERING

Final Report

AUGUST 7, 2021

Altan-Ochir Anuul

20B60012

Contents

Table of Figures

Problem 1 Diffusion Equation

Construct a diffusion model for a 2-D heat plate with dimensions 100 m. by 100 m given the equation,

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0$$

Implement the following:

1. Investigate steady-state condition given boundary conditions (Dirichlet) using "successive over-relaxation" method. Use poisson equation by assigning external forcing (e.g. *g*). (10 points)
2. Investigate various boundary conditions (non-steady state) by discussing its effects using animation. Feel free to set the grid sizes, time-steps, internal parameters. Ensure stability by setting an appropriate *d*. Discuss clearly the set-up and your interpretations of the results.
   a. Dirichlet Boundary condition (5 points)
   b. Neumann Boundary condition (5 points)
   c. Mixed boundaries (5 points)
3. Investigate the influence of $d = \frac{\alpha \Delta t}{\Delta x^2}$ by testing various values for *d* (0. to 1.0). What are the threshold values for *d* to simulate a stable model behavior? Why is *d* causing this effect? (10 points) Discuss clearly the set-up and your interpretations of the results.

Program

The following is the program used to investigate the Dirichlet using successive over-relaxation method.

```
import numpy as np
import matplotlib.pyplot as plt
g = 9.81
dx = dy = 1.
alpha = 0.7
len_x = 100.
len_y = 100.
grid_x = int(len_x/dx)
grid_y = int(len_y/dy)
x = np.linspace(0.,grid_x*dx,grid_x)
y = np.linspace(0.,grid_y*dy,grid_y)
T = np.zeros((grid_y,grid_x))
T[:,0] = 0.
T[:,-1] = 0.
T[0,:] = 0.
T[-1,:] = 0.
T[0,25:75] = 50.
w = 1.6
for k in range(0,nk):
    for j in range(1,grid_y-1):
        for i in range(1,grid_x-1):
            R = 0.25*(T[j,i+1]+T[j,i-1]+T[j+1,i]+T[j-1,i]-g/alpha*dx**2.)
            T[j,i] = (1-w)*T[j,i]+w*R
plt.contourf(x,y,T,cmap='rainbow',levels=np.linspace(np.min(T),np.max(T),30))

plt.colorbar()
plt.show()
T = np.zeros((grid_y,grid_x))
T[:,0] = 0.
T[:,-1] = 0.
T[0,:] = 0.
T[-1,:] = 0.
```

```
T[0,25:75] = 50.
w = 1.9 #Corresponding to omega in the SOR algorithm.
nk = 1000 #Corresponds to the number of iterations.
icounter = 0
for k in range(0,1000):
    for j in range(1,grid_y-1):
        for i in range(1,grid_x-1):
            if (k == 0)&(j < 50)&(i < 50): #Only see the effect for one iteration and loop of i and j less than 50.
                plt.clf()
                plt.cla()
                plt.contourf(x,y,T,cmap='rainbow',levels=np.linspace(-3.,51.,30))
                plt.colorbar()
                plt.title('k=%04.4i i=%03.3i j=%03.3i'%(k,i,j))
                plt.savefig('%06.6i.png'%(icounter))
                icounter += 1
            R = 0.25*(T[j,i+1]+T[j,i-1]+T[j+1,i]+T[j-1,i]+g/alpha*RhoT*dx**2.)-T[j,i]
            T[j,i] = T[j,i]+w*R
```

3D animation of the Diffusion (Mixed Condition). I uploaded the Mixed Boundary condition here since it is easier to change the boundaries to Dirichlet and Neuman.

```
!pip install ffmpeg
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML
from base64 import b64encode
def play(filename):
    html = ''
    video = open(filename,'rb').read()
    src = 'data:video/mp4;base64,' + b64encode(video).decode()
    html += '<video width=1000 controls autoplay loop><source src="%s" type="video/mp4"></video>' % src
    return HTML(html)
!rm *.jpg
d = 0.255
alpha = 0.7
dx = 0.1
len_x = 10.0 #
len_y = 10.0 #
dt = d*(dx**2)/alpha
end_time = 3.0
x = np.arange(0.0, len_x+dx, dx)
y = np.arange(0.0, len_y+dx, dx)
X, Y = np.meshgrid(x,y)
T = np.zeros_like(X)
G = 0.0  # no flux
T[25:-70,45:-50] = 10.0 #
T[-70:25,45:-50] = 10.0 #
T[0,:] = T[0,:] + d*(T[1,:]-4.0*T[0,:]+T[1,:]-2.0*dx*G+np.roll(T[0,:],-1,axis=0)+np.roll(T[0,:],1,axis=0))
T[-1,:] = 1.0
T[7,2:5] = 1.0
plt.figure(figsize=(10,10)) #
icounter = -1
```

```
for it in np.arange(0.0, end_time+dt, dt):
    icounter = icounter + 1
    if np.mod(icounter,10)==0:
        ax = plt.axes(projection='3d')
        p = ax.scatter(X,Y,T,c=T,cmap='rainbow',vmin=0.0,vmax=5.0)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlim([0.0,10.0])
        plt.colorbar(p)
        plt.title('Time=%.6f'%(it))
        plt.savefig('%06.6d.jpg'%(icounter))
        plt.cla()
        plt.clf()
    T[1:-1,:] = T[1:-1,:] + d*(np.roll(T[1:-1,:],-1,axis=1)+np.roll(T[1:-1,:],1,axis=1)+T[2:,:]+T[0:-2,:]-4.0*T[1:-1,:])
    T[25:-70,45:-50] = 10.0
    T[-70:25,45:-50] = 10.0
    T[0,:] = T[0,:] + d*(T[1,:]-4.0*T[0,:]+T[1,:]-2.0*dx*G+np.roll(T[0,:],-1,axis=0)+np.roll(T[0,:],1,axis=0))
    T[-1,:] = 1.0
    T[7,5] = 1.0
!rm *.mp4
!ffmpeg -r 3 -pattern_type glob -i '/kaggle/working/*.jpg' -vf "scale=trunc(iw/2)*2:trunc(ih/2)*2" -vcodec libx264 -pix_fmt yuv420p test.mp4
play('/kaggle/working/test.mp4')
```

Objective & Algorithm for SOR:

Here in this section, we will solve the first part of problem 1 Diffusion equation.

The objective is to investigate the 2D heat plate with dimensions 100m by 100m given by the following equation with external forcing g:

Here the following is the derivation of the discretized form of the given diffusion equation with external force of g:

$$\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = g$$

$$\frac{\partial T}{\partial t} = g + \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0$$

Since $\frac{\partial T}{\partial t} = 0,$ we obtain

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = -\frac{g}{\alpha}$$

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} = -\frac{g}{\alpha}$$

$$T_{i+1,j} - 2T_{i,j} + T_{i-1,j} + T_{i,j+1} - 2T_{i,j} + T_{i,j-1} = -\Delta x^2 \frac{g}{\alpha}$$

Now, we can acquire the discretized form of the equation:

$$T_{i,j} = \frac{1}{4}\left(T_{i+1,j} + T_{i-1,j} + T_{i,j+1}T_{i,j-1} + \Delta x^2 \frac{g}{\alpha}\right)$$

SOR Procedure:

1. Set k numbers of iteretions

2. Calculate the residual R^k for each iteration loop as follows:

$$R_{i,j}^k = \frac{1}{4}\left(T_{i+1,j}^k + T_{i-1,j}^{k+1} + T_{i,j+1}^k + T_{i,j-1}^{k+1} + \Delta x^2 \frac{g}{\alpha}\right) - T_{i,j}^k$$

Then conduct the adjustment for $T_{i,j}$ for k+1-th iteration

$$T_{i,j}^{k+1} = T_{i,j}^k + \omega R_{i,j}^k$$

Discussion

1.1)

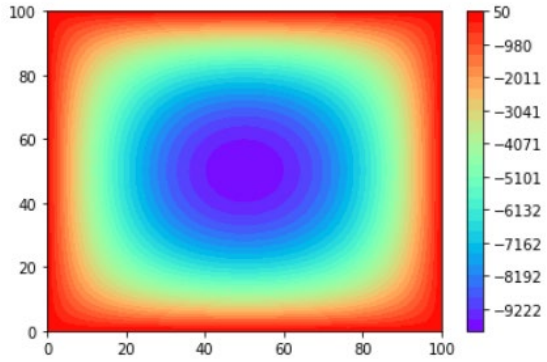

Figure 1.1 SOR when alpha is 0.7

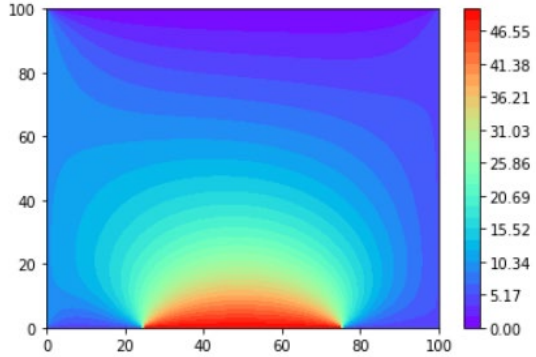

Figure 1.2 SOR when alpha is 100000

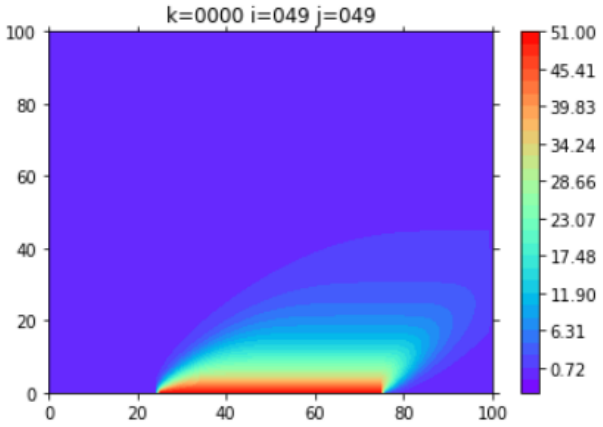

Figure 1.3 SOR when alpha is 0.7

In this section, I have taken the external force as a gravitational g=9.81 solely. In the class, we previously did the SOR on similar phenomena. In that case, we had the Rhot value which is very low that makes the influence of g to be nearly nothing. But, In this class, I wanted to test, what if there is only a g. The result I got from g is in Figure 1.1 when taking the alpha 0.7, and when we take the alpha value as a 100000, the plot looks more similar to the equation that we worked on in the class. The reason lies in the influence of g. For instance, the reason we take alpha so high is to see the distribution when there is almost no external force (negligent) while also seeing the model when an external force is large enough to consider.

1.2) On the program that I have provided, I included the 3D animation in case of mixed boundaries so that we could change the boundary conditions to Neumann or Dirichlet faster.
For the following Boundary condition, I changed the initial conditions several times to see the relations. For instance, as the value of d increases, the breakdown were more commonly happening

which leads to chaos and unstable condition. But, on the other hand, if we lower the d value too much, the value of dt will gets decreases which lead to an increased amount of iteration number and executing time.

Dirichlet Boundary Condition:

On the Dirichlet boundary condition, we are assigning the specific value at the boundary (the wall) that has a grid point with a distance of dx throughout the dimension.

Assigned Values:

d=0.1, alpha=0.7, dx=0.1, boundary values= between 1-2

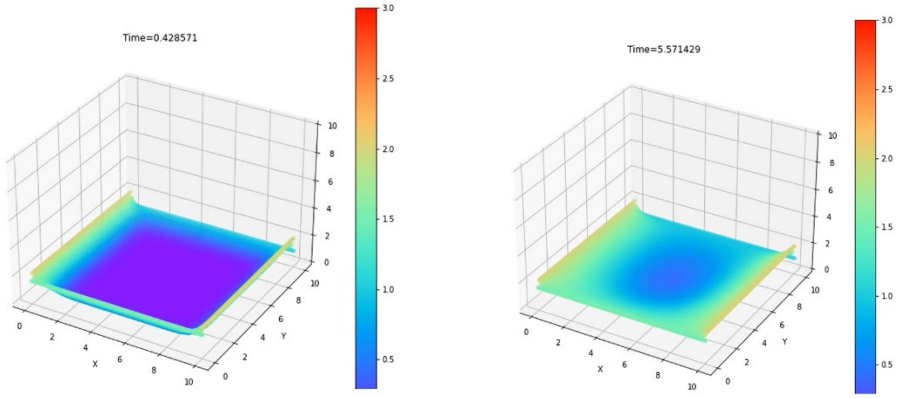

Figure 2.4 & 1.5 Testing Dirichlet Boundary Condition at d=0.1

From the Figure 1, we can observe that initially we have the Dirichlet Boundary Condition (Specific Values assigned at walls). As time passes the heat distributed from Boundaries is approaching to equilibrium condition. From this process we can explain many physical phenomena in real life. For instance, bringing the cool soda to room temperature or the room temperature increases, etc.

Neuman Boundary Condition:

On the Neuman condition, we are specifying the gradient which means we are approximating the dT/dx by a certain value G under a no flux condition.

d=0.1, alpha=0.7, dx=0.1,

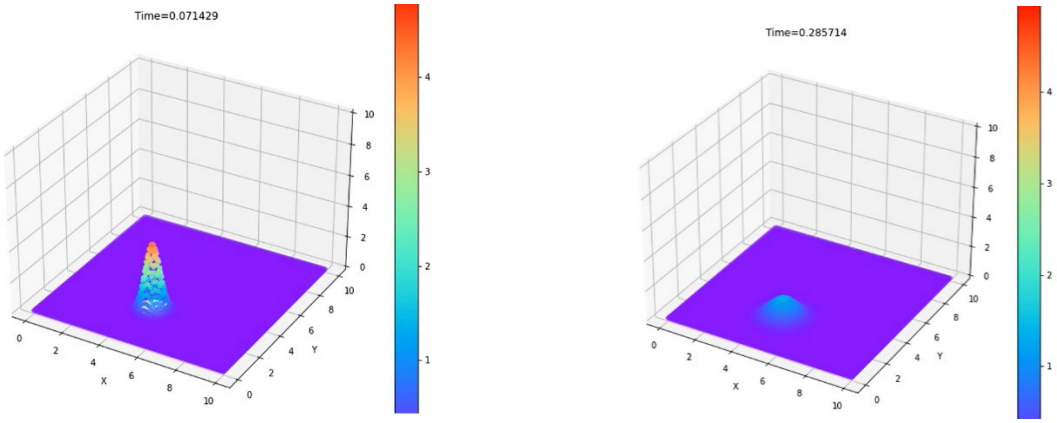

Figure 3.6 & 1.7 Testing Neuman Boundary Condition at d=0.1

When we are applying the Neuman Boundary Condition, the heat flux from a surface would serve as a boundary condition. Initially, we have a source point and it heats the nearby points as we can see on figure 6, but after a long enough time the heat spreads and spreads until it gets very low since we don't have Dirichlet Boundary Condition

Mixed Boundary Condition:

To investigate Mixed Boundary Condition, we apply both Dirichlet and Neuman Boundary Condition to see the physical phenomena.

d=0.1, alpha=0.7, dx=0.1, boundary values= 1

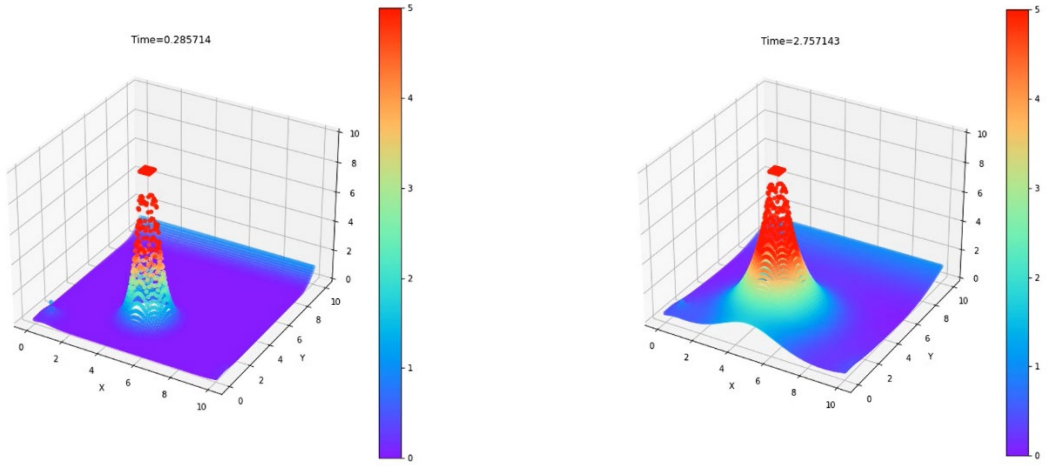

Figure 4.8 & 1.9 Testing Mixed boundary condition at d=0.1

In this case, we can observe both Dirichlet and Neuman conditions. Since we have the initial heat, it distributes and heats up the surroundings while the Dirichlet boundary keeps and bounds regarding the change time.

1.3) As we investigate more deeply into the 3d graph by changing the d value. I have found that diffusion was breaking when it is between 0.25 and 0.26. So, I set 0.255 to see what would happen to the animation. As we can see from the Figure 1 and 2, we can notice that for the first 0.5 spans of time, the PDE was diffusing normally, but after a while, it goes chaotic which means the breaking point is somewhere near. So that, I tried the simulation again when d is 0.25 and then the diffusion was going smoothly. Therefore, we can conclude that the **threshold value of d is 0.25**

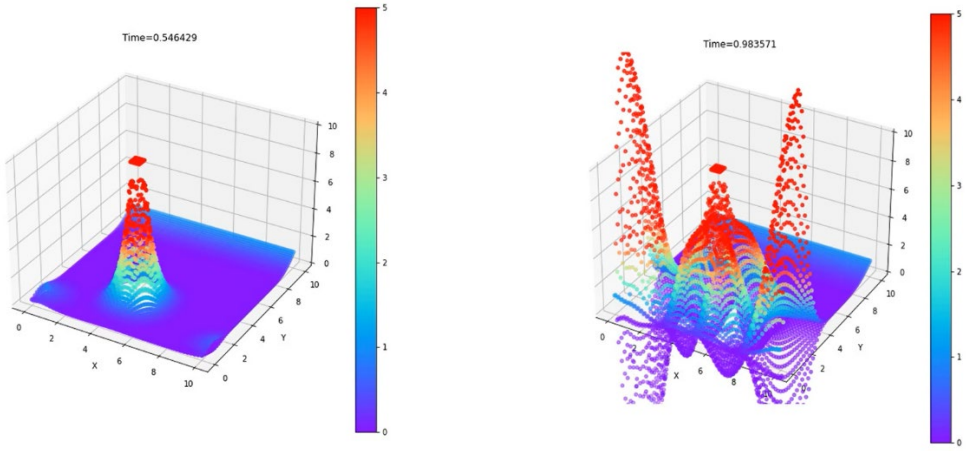

Figure 5.10 & 1.11 Setting d value as 0.255 to see the break point

The reason why it is 0.25:

The main reason lies on Von Neumann Stability Analysis. From that, we can find the derivation of 1D heat equation`s $\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2}\right) = 0$ stability condition which is $d = \frac{\alpha * \Delta t}{(\Delta x)^2} \leq \frac{1}{2}$. Since we are worked on the 2D $\frac{\partial T}{\partial t} - \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) = 0$, we can further obtain the following $\alpha\left(\frac{\Delta t}{(\Delta x)^2} + \frac{\Delta t}{(\Delta y)^2}\right) = \frac{1}{2}$ which leads to $d = \frac{\alpha * \Delta t}{(\Delta x)^2} \leq \frac{1}{4}$ when $(\Delta x)^2 = (\Delta y)^2$

Problem 2 Burger's Equation

Using forward-in-time and backward-in-space for the 1st derivative, and centered difference for the 2nd derivative, construct a numerical model for the Burger's equation. Decide your own initial conditions and internal parameter value, *v*.

$$\frac{\partial u}{\partial t} + a\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) = v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Discuss the following (add figures if necessary):

1. Derive and write (type or by hand) the discretized form of the equation above as an algebraic equation (10 points).
2. Investigate by modelling the differences when *a* (linear) is a constant and when *a* is *u* (non-linear). Use a cyclic boundary (10 points). Discuss clearly the set-up and your interpretations of the results.
3. Investigate the behavior of *u* with time as you set a cyclic boundary at one axis and a close-wall boundary at another axis (i.e. no net flux) (10 points). Discuss clearly the set-up and your interpretations of the results.

Program

Here down, I included the script used to investigate the model`s characteristic differences by changing the value of variable a to u and numerical value which I took as an one.

```
import numpy as np
import matplotlib.pyplot as plt
len_x = 100.0
len_y = 100.0
v = 1
n=0
dx = dy = 1.0
dt = 0.01
end_time = 100
x = np.arange(0.0,len_x+dx,dx)
y = np.arange(0.0,len_y+dy,dy)
X,Y = np.meshgrid(x,y)
u = np.zeros((y.shape[0],x.shape[0]))
u[5:50,10:60] = 20.0
for it in np.arange(0.0,end_time + dt,dt):
    a=u
    #a=1
    u = u + (dt/dx) * (a * (np.roll(u,1,1)+np.roll(u,1,0)-
2*u) + v/dx * (np.roll(u,-1,1)+np.roll(u,+1,1)+np.roll(u,-
1,0)+np.roll(u,+1,0)-4*u))
    if n%30==0:
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, u, cmap='rainbow', linewidth=0)
        ax.view_init(20,80)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('u')
        plt.savefig('test_%05i.png'%(n/30))
        plt.cla()
        plt.clf()
    n+=1
```

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, u, cmap='rainbow', linewidth=0)
ax.view_init(20,80)
ax.set_ylabel('y')
ax.set_zlabel('u')
```

```
import numpy as np
import matplotlib.pyplot as plt
# initial values
dx = dy = 1.0
len_x = 100.0
len_y = 100.0
end_time = 100
dt = 0.01
a = 1
n = 0
v = 1
Top = 0
Bottom = 0
Right = 0
Left = 0
# meshgrid
x = np.arange(0.0,len_x+dx,dx)
y = np.arange(0.0,len_y+dy,dy)
X,Y = np.meshgrid(x,y)
u = np.zeros((y.shape[0],x.shape[0]))
u[5:50,10:60] = 20.0
# cyclic iteration
for it in np.arange(0.0,end_time + dt,dt):
    uc = np.zeros((y.shape[0]+2,x.shape[0]+2))
    uc[1:-1,1:-1] = u
    uc[1:-1,0] = u[:,1] + dy*Top
    uc[1:-1,-1] = u[:,-1] + dy*Bottom
    uc[0,1:-1] = u[-1,:] + dx*Right
    uc[-1,1:-1] = u[1,:] + dx*Left
```

```
    uc = uc+(dt/dx)*(a*(np.roll(uc,1,1)+np.roll(uc,1,0)-np.roll(uc,-1,0)-np.roll(uc,-1,1))+v/dx*(np.roll(uc,-1,1)+np.roll(uc,+1,1)+np.roll(uc,-1,0)+np.roll(uc,+1,0)-4*uc))
    u = uc[1:-1,1:-1]

    if n%30==0:
        fig = plt.figure()
        ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, u, cmap='rainbow', linewidth=0)
        ax.view_init(20,80)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('u')
        plt.savefig('test_%05i.png'%(n/30))
        plt.cla()
        plt.clf()
    n+=1
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, u, cmap='rainbow', linewidth=0)
ax.view_init(20,80)
ax.set_ylabel('y')
ax.set_zlabel('u')
```

Discussion

2.1)

$$\frac{\partial u}{\partial t} + a\left(\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}\right) = v\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

Apply the basic FDM methods to derive the following:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + a\left(\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\partial x} + \frac{u_{i,j}^{n} - u_{i,j-1}^{n}}{\partial y}\right) = v\left(\frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} - 2u_{i,j}^{n}}{\partial x^2} + \frac{u_{i,j+1}^{n} + u_{i,j-1}^{n} - 2u_{i,j}^{n}}{\partial y^2}\right)$$

Assume dx = dy, and then derive the discretized form of the equation.

$$\boldsymbol{u_{i,j}^{n+1} = u_{i,j}^{n} - \frac{a\Delta t}{\Delta x}\left(2u_{i,j}^{n} - u_{i-1,j}^{n} - u_{i,j-1}^{n}\right) + \frac{v\Delta t}{\Delta x^2}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n} - 4u_{i,j}^{n}\right)}$$

2.2)

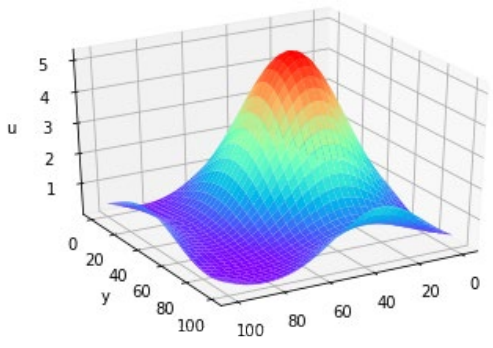

Figure 2.1 Setting the value of a as 1

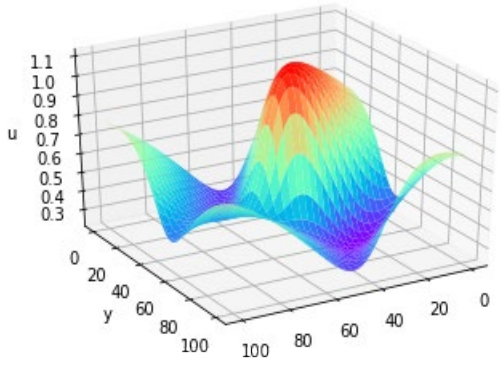

Figure 2.2 Setting the value of a as u

Here we have the 2 different figures taking the value of variable a as both 1 (numerical value) and u. In case we suppose the number, we get the linear equation while u for the non-linear equation. The common thing that we can notice from the graph is that, when we take the linear equation, the model with cyclic boundary looks smoother than the non-linear equation. So, we can prove that nonlinear systems are complicated and if we look more deeply the reason lies in the high dependency of the system variables on each other.

Another important aspect we can observe is that the nonlinear equation can produce bifurcation in the model. As we can see in Figure 2.2, between the x-y coordinate we can see the bifurcation produced differently than the linear case.

2.3)

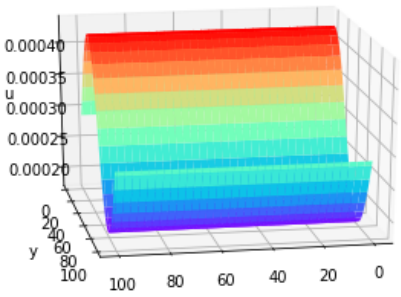

Figure 2.3

In this graph, I am certain that I made mistake somewhere because I was facing a lot of errors, and did not work as I expected. Even the derived plot has an extraordinary image. But what we can see is that we can observe on the u-y plane we got a sinusoidal curve-like shape. As for the x plane, it has the same value for the certain u and y values. I am expecting that we have received this result because we have taken the value as an integer which makes the equation linear so there is no bifurcation that we can perceive.

Problem 3 1-D Advection Equation

Given the following 1-dimensional equation

$$\frac{\partial f}{\partial t} + u\left(\frac{\partial f}{\partial x}\right) = 0$$

At t=0,

$$f(0,x) = \begin{cases} 1 & 40 \le x \le 60 \\ 0 & \text{otherwise} \end{cases}$$

and with a cyclic boundary, discuss using the following parameters: $u = 1.0, \Delta x = 1.0, 0 \le x \le 100$. To answer the questions, decide on appropriate C values. Construct a model using (1) Upwind scheme (10 pts), (2) Leith's Method (10 pts), (3) CIP Method (10 pts), and (4) analytical solution (10 pts).

1. Construct f plots along x for t=10, 200, 400, 600. Compare the results of each method and discuss the errors accompanied by each method. Discuss clearly the set-up and your interpretations of the results.

Program

Plotting the 1-D Advection by using Upwind Scheme

The previous Script Highlighter is not working today ☹

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # initial values
4. C = 0.1
5. u = 1.0
6. dt = 0.1
7. dx = 1.0
8. # time span and x
9. x = np.arange(0, 100+dx, dx)
10. t = np.arange(0, 600+dt, dt)
11. # assigning f(0,x)
12. f = np.zeros(x.shape[0])
13. f[40:60] = 1
14. fx=np.copy(f)
15. # Plotting in case of 10,200,400 and 600
16. for iter in t:
17.     fx = fx+C*(np.roll(fx,int(np.sign(u)),0)-fx)
18.     if iter==10:
19.         plt.plot(x,fx, label= f't = {iter}')
20.         plt.legend()
21.     if iter==200:
22.         plt.plot(x,fx, label= f't = {iter}')
23.         plt.legend()
24.     if iter==400:
25.         plt.plot(x,fx, label= f't = {iter}')
26.         plt.legend()
27.     if iter==600:
28.         plt.plot(x,fx, label= f't = {iter}')
29.         plt.legend()
```

Plotting the 1-D Advection by using Leith`s Method

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. C = 0.1
4. u = 1.0
5. dt = 0.1
6. dx = 1.0
7. x = np.arange(0, 100+dx, dx)
8. t = np.arange(0, 1000+dt, dt)
9. func = np.zeros(x.shape[0])
10. func[40:60] = 1
11.
12. fx=np.copy(func)
13. for iter in t:
14.     c = fx
15.     b = 1/(2*dx)*(np.roll(fx,-1,0)-np.roll(fx,1,0))
16.     a = 1/(2*dx**2)*(np.roll(fx,-1,0)-
    2*fx+np.roll(fx,1,0))
17.     fx = a*(C*dx)**2-b*(C*dx)+c
18.
19.     if iter==10:
20.         plt.plot(x,fx, label= f't = {iter}')
21.         plt.legend()
22.     if iter==200:
23.         plt.plot(x,fx, label= f't = {iter}')
24.         plt.legend()
25.     if iter==400:
26.         plt.plot(x,fx, label= f't = {iter}')
27.         plt.legend()
28.     if iter==600:
29.         plt.plot(x,fx, label= f't = {iter}')
30.         plt.legend()
```

Plotting the 1-D Advection by using CIP Method

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Initial Values
4. dx = 1.0
5. C = 0.1
6. u = 1.0
7. dt = 0.1
8. # Time span and X
9. x = np.arange(0, 100+dx, dx)
10. t = np.arange(0, 1000+dt, dt)
11. # Assigning f(0,x)
12. f = np.zeros(x.shape[0])
13. f[40:60] = 1
14.
15. fx=np.copy(f)
16.
17. g = (np.roll(fx,-1)-np.roll(fx,1))/(2*dx)
18.
19. for iter in t:
20.     a = -2*(np.roll(fx,int(np.sign(u)),0)-fx)/((-
    dx*np.sign(u))**3)+(g+np.roll(g,int(np.sign(u))))/((-
    dx*np.sign(u))**2)
21.     b = -3*(fx-np.roll(fx,int(np.sign(u)),0))/((-
    dx*np.sign(u))**2)-(2*g+np.roll(g,int(np.sign(u))))/(-
    dx*np.sign(u))
22.     fx = a*(-C*dx)**3+b*(-C*dx)**2+g*(-C*dx)+fx
23.     g = 3*a*(-u*C*dx)**2+2*b*(-C*dx)+g
24.     if iter==10:
25.         plt.plot(x,fx, label= f'When t is {iter}')
26.         plt.legend()
27.     if iter==200:
28.         plt.plot(x,fx, label= f'When t is {iter}')
29.         plt.legend()
30.     if iter==400:
```

```
        plt.plot(x,fx, label= f'When t is {iter}')
        plt.legend()
    if iter==600:
        plt.plot(x,fx, label= f'When t is {iter}')
        plt.legend()
```

Plotting the 1-D Advection by using Analytical Solution

```
import numpy as np
import matplotlib.pyplot as plt
# initial values
C =0.1
dx = 1.0
u = 1.0
dt = 0.1
# time span and x
x = np.arange(0, 100+dx, dx)
t = np.arange(0, 1000+dt, dt)
# assigning f(0,x)
f = np.zeros(x.shape[0])
f[40:60] = 1
fx1=np.copy(func)
for iter in t:
    fx = np.roll(fx1,round(u*iter/dx),0)
    if iter==10:
        plt.plot(x,fx, label= f't = {iter}')
        plt.legend()
    if iter==200:
        plt.plot(x,fx, label= f't = {iter}')
        plt.legend()
    if iter==400:
        plt.plot(x,fx, label= f't = {iter}')
        plt.legend()
    if iter==600:
        plt.plot(x,fx, label= f't = {iter}')
        plt.legend()
```

Discussion

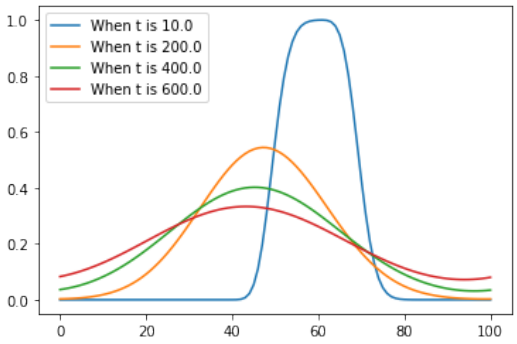

Figure (3.1) 1-D advection by Upwind Scheme

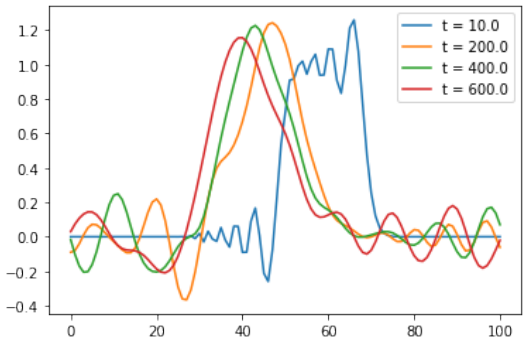

Figure (3.2) 1-D advection by Leith

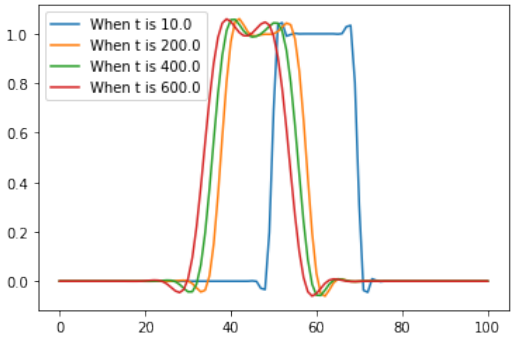

Figure (3.3) 1-D advection by CIP Method

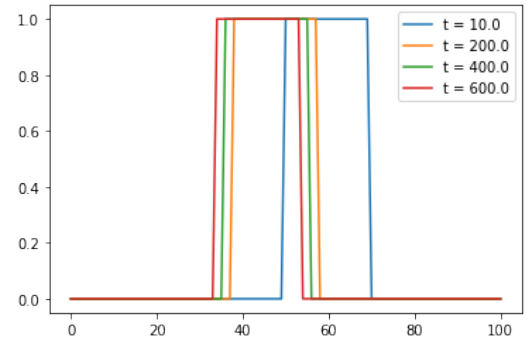

Figure (3.4) 1-D advection by Leith

For each of the plots, the initial values are initially given as:

dx = 1.0, u = 1.0

$$f(0,x) = \begin{cases} 1 & 40 \le x \le 60 \\ 0 & \text{otherwise} \end{cases}$$

Also, to see less error, we set the C as 0.1. and set the dt to be 0.1

For the upwind Scheme, I tried to change the C values from 0.1 to 1.1. The value of C was working properly until it reaches 1. After 1, the program shows the error or showing so chaotic plot.

In the case of Leith, it shows the same behavior as the Upwind scheme regarding the C values. When I change it to be above 1 it shows the error.

In the case of the CIP Method, similar to the rest of the equations, we have experienced chaos when C is above or equal to 1. Then interestingly we can see that it is closer to the analytic solution which shows the error rate is much lower than any other method which proves high order approximation is more precise.

About the analytic solution, we can see the clear result which is a block with a height of 1 with an x range of 60-40=20. As time passes, it moves leftward, and eventually, you can notice that the speed is the same since the distance travel between 200, 400, and 600 are the same.