PARTIAL DIFFERENTIAL EQUATIONS FOR SCIENCE AND ENGINEERING

TSE.M202-01

Erdenebat Battseren
ID: 20B60029

Final Report

Instructor: Prof. Varquez Alvin Christopher Galang
Transdisciplinary Science and Engineering
Tokyo Institute of Technology
August 7th, 2021

Contents

Listings

List of Figures

1 Problem 1: Diffusion Equation

2-D heat plate with dimensions 100 m by 100 m given by the equation,

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

1.1 Steady-State Condition: Poisson equation

To investigate the steady-state condition given by Dirichlet boundary conditions, let us use *Successive Over-relaxation* method or SOR for Poisson equation.

Algorithm for SOR 2-D Poisson equation or the steady state of the heat conduction can be expressed as follows. We consider the existence of external force of gravity.

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

Assuming $\Delta x = \Delta y$, construct finite difference method:

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

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

SOR Procedure:

1. Set k number of iterations
2. Calculate the residual R^k for each iteration loop as follows:

$$R^k_{i,j} = \frac{1}{4}\left(T^k_{i+1,j} + T^{k+1}_{i-1,j} + T^k_{i,j+1} + T^{k+1}_{i,j-1} + \Delta x^2 g\frac{\rho}{T} \right) - T^k_{i,j}$$

3. Make adjustments for $T_{i,j}$ for $k+1$-th iteration:

$$T^{k+1}_{i,j} = T^k_{i,j} + \omega R^k_{i,j}$$

Implementation: We chose the initial condition as follows:

- External force: g = 9.81 m/s^2
- $\frac{\rho}{T} = 0.01$
- Temperatures at the top, bottom, left and right walls are: $300K$, $300K$, $200K$, $200K$.
- There is a heat source at the middle of the matrix with a temperature $T = 100K$

Algorithm for SOR Procedure:
We calculate individually for i and j since T^{k+1} refers to the next time step. Also, we can measure the computation time for each calculation.

```
time_start = time.perf_counter()  # Time computation
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*RhoT*dx**2.)
            T[j,i] = (1-w)*T[j,i]+w*R  # Adjustment for each iteration
time_elapsed = (time.perf_counter() - time_start)
memMb=resource.getrusage(resource.RUSAGE_SELF).ru_maxrss/1024.0/1024.0
print ("%5.1f secs %5.1f MByte" % (time_elapsed,memMb))
```

Listing 1: Python code (part of): SOR Algorithm

Following is the trial-and-error values for the relaxation coefficient - ω

ω (relaxation coefficient)	T - Elapsed Time (s)	Resource (MByte)
1.30	21.9	0.20
1.40	21.4	0.20
1.49	21.9	0.20
1.50	20.0	0.20
1.51	21.2	0.20
1.60	20.8	0.20
1.72	20.4	0.20
1.80	20.6	0.20
1.90	20.9	0.20

Table 1: Trial and Error method for the value of ω

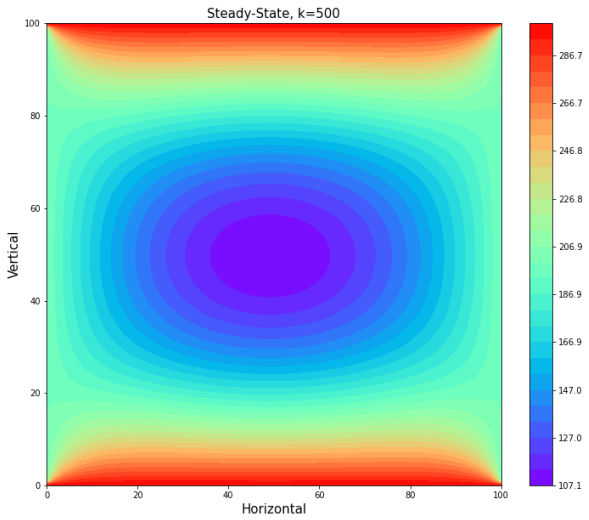

Figure 1: Steady-State of the 2-D heat conduction

Discussion: We defined the Dirichlet boundary conditions at 4 walls and the fixed heat source at the middle of the matrix. From Figure 1, we can see the steady-state condition of the heat conduction after 500 iterations. The SOR method adjusts the value of T for each iteration until the difference between the next time-step is small enough to be negligible. Which indicates the steady state condition. We could not see the effect of gravity since our value for $\frac{\rho}{T}$ was set sufficiently small.

Furthermore, we observed that the computation time is dependent on the value of omega-ω or the relaxation coefficient. From Table 1, we can see that the best optimal value for omega is $\omega = 1.50$ with a computation time of $t = 20.0s$ for $k = 500$ iterations.

1.2 Investigation of various boundary conditions

To investigate the various boundary conditions, let us use the discretization method - FTCP. Discretization can be implemented as follows: Assuming $\Delta x = \Delta y$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \alpha \left\{ \frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2} \right\}$$

$$T_{i,j}^{n+1} - T_{i,j}^{n} = \frac{\alpha \Delta t}{\Delta x^2} \left\{ T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n} \right\}$$

$$T_{i,j}^{n+1} = T_{i,j}^{n} + d \left\{ T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n} \right\}$$

Where $d = \alpha \Delta t / \Delta x^2$ is the Diffusion Number. The system stability is dependent on the value of d which will be discussed later. We set different boundary conditions from the previous modeling.

1.2.1 Dirichlet Boundary Condition

Program:

```
import numpy as np
import matplotlib.pyplot as plt

# Initial and given conditions
d = 0.1
alpha = 0.7
dx = 0.1
len_x = 10.0
len_y = 10.0
dt = d*(dx**2)/alpha
end_time = 2.0

x = np.arange(0.0, len_x+dx, dx) # number of grids, different points of x
y = np.arange(0.0, len_y+dx, dx)
X, Y = np.meshgrid(x,y)

# Initial Condition:
T = np.zeros_like(X)
```

```
#T[:,-1] = 10.0
T[0,:] = 25.0
T[-1,:] = 25.0
T[45:50,45:50] = 60.0

plt.figure(figsize=(20,20))
icounter = -1
for it in np.arange(0.0,end_time+dt, dt):
    #if it ==0.0:
    icounter = icounter + 1
    if np.mod(icounter,15)==0:
        ax = plt.axes(projection='3d')
        p = ax.scatter(X,Y,T, c=T, cmap='plasma', vmin = 0.0, vmax = 60.0)
        ax.set_xlabel('X',fontsize = 20)
        ax.set_ylabel('Y',fontsize = 20)
        ax.set_zlim([0.0,60.0])
        plt.colorbar(p)
        plt.title('Time=%.6f'%(it),fontsize = 20)
        plt.savefig('%06.6d.jpg'%(icounter))
        plt.cla()
        plt.clf()
    T = T + d*(np.roll(T,-1,axis=1)+np.roll(T,1,axis=1)+np.roll(T,-1, axis
   =0)+np.roll(T,1, axis=0) - 4.0*T)
#    T[:,0] = 10.0
#    T[:,-1] = 10.0
    T[0,:] = 25.0
    T[-1,:] = 25.0
    T[45:50,45:50] = 60.0
```

Listing 2: Dirichlet Boundary Condition

Discussion: Dirichlet Boundary condition indicates the values that a solution needs to take along the boundary of the domain. Therefore, when programming, we need to define the boundary values for every time step. From Line-40 from Listing-2, you can see how the FTCP method is implemented, and from lines 41-45, Dirichlet boundary conditions are specified in every time step. From Figure-2, you can see the fixed boundary conditions are acting as if they are absorbing the heat of the system.

From the animation (included in the ZIP file), we can observe that the diffusion velocity slows down as the system proceeds over time. We assume that the system will eventually reach a steady state if we give a sufficient amount of time.

1.2.2 Neumann Boundary Condition

To demonstrate a Neumann Boundary condition, we need to specify a gradient. The physical meaning of this gradient can be interpreted as a heat flux. We need to define $dT/dx = G$ at a certain region of the system. However, we cannot directly program it using the FTCP method. Thus, we must use some kind of approximation that brings the concept of a Ghost point. We use central difference discretization and some imaginary points in space.

$$\frac{dT}{dx} = G = \frac{T_2 - T_0}{2\Delta x}$$

FTCP and Neumann Boundary Condition:

```
# adjusted T
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[:,0] = 10.0
T[:,-1] = 10.0
# Neumann Boundary condition
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,:] = 20.0
T[10:20,45:50] = 60.0
```

Listing 3: Neumann Boundary Condition

Discussion: Unlike Dirichlet Boundary Condition, we adjusted the value of T. From Line-2 from Listing-3, we can see that T is adjusted since it is not cyclic anymore. The reason why function - np.roll is not used on the axis=0 is to prevent the cycling of the system inside itself. The heat flux is specifically defined at the region therefore the cyclic condition is no longer satisfied.

From Figure-3, the Neumann boundary condition is defined on the x-axis and it is acting as a wall inside a system. The characteristic of the system can be observed in more detail from the animation.

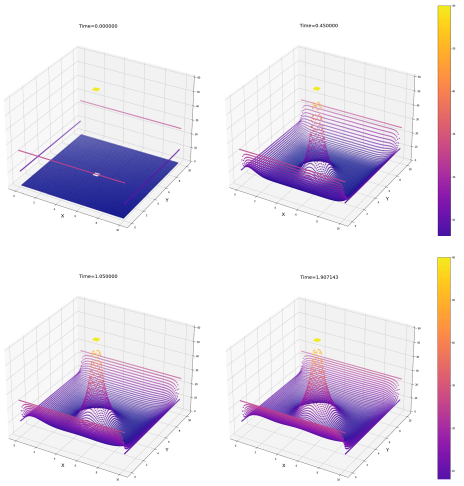

Figure 2: Dirichlet Boundary Condition

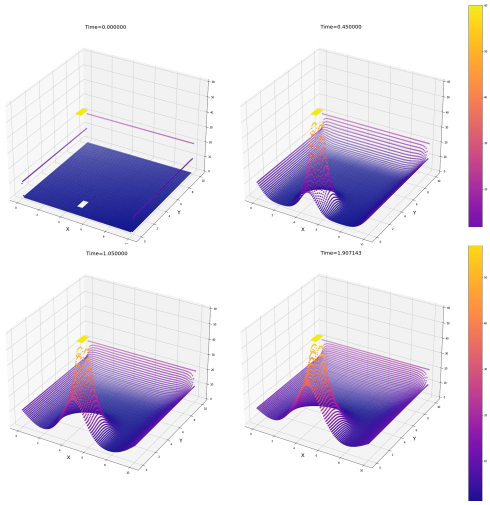

Figure 3: Neumann Boundary Condition

1.2.3 Mixed Boundary Conditions

Two different mixed boundaries were investigated using different conditions. First, cyclic and Dirichlet boundary conditions. We only defined the boundary conditions at the x-axis. From Figure-4, we can see that x-axis values are fixed. However, at the y-axis, the system overflows to the other ends of the system, satisfying cyclic conditions.

Second, Neumann and Dirichlet boundary conditions are set at the same time. We defined a Dirichlet condition on the y-axis and Neumann condition on the x-axis. From Figure-5, we can observe two walls are acting on the x-axis while and values at the y-axis are fixed.

Program:

```
# Adjusted T
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[:,0] = 10.0
T[:,-1] = 10.0
# Two walls acting on the x-axis
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,:] = T[0,:] +d*(T[0,:]-4.0*T[-1,:]+T[0,:]-2.0*dx*G+np.roll(T
    [-1,:],-1,axis=0)+np.roll(T[-1,:],1,axis=0))
# Two Heat sources
T[10:20,45:50] = 60.0
T[80:90,45:50] = 60.0
```

Listing 4: Neumann Boundary Condition

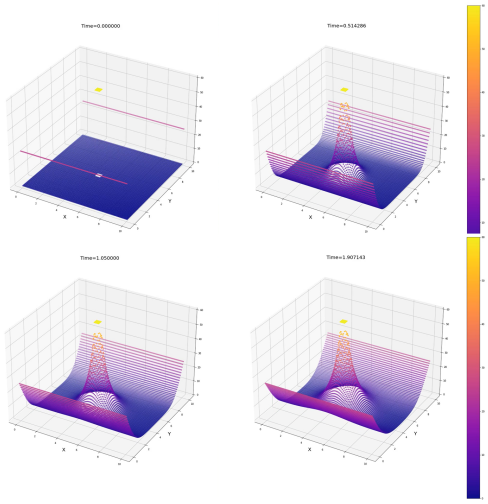

Figure 4: Dirichlet and Cyclic Boundary Condition

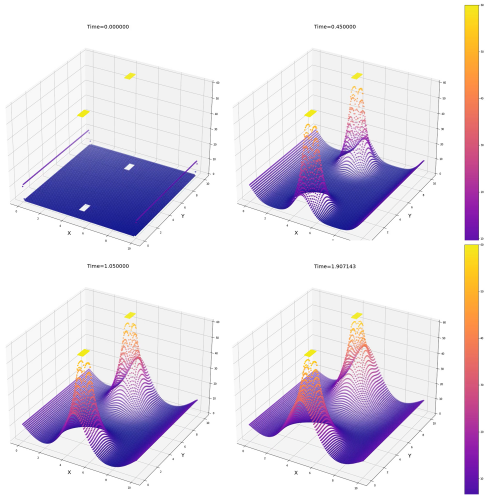

Figure 5: Dirichlet and Neumann Boundary Condition

1.3 System Stability

As mentioned briefly at the beginning, the system stability is hugely dependent on the value of d, the diffusion number. We will consider Von Neumann Stability Analysis and derive a range of threshold values for d to be stable. Various d values were tested to investigate the influence of $d = \alpha\Delta t/\Delta x^2$ ranging from $0.0 \leq d \leq 1.0$.

1.3.1 Von Neumann Stability Analysis

Regardless of a higher order of accuracy of numerical modeling, it will always be an approximation of the system. There will always be an existence of error. Let us express this term mathematically for 2-D diffusion equation.
From FTCS method, introduce an error term:

$$T_i^n = \bar{T}_i^n + \varepsilon_i^n$$

$$\frac{\varepsilon_{i,j}^{n+1} - \varepsilon_{i,j}^n}{\Delta t} = \alpha \left(\frac{\varepsilon_{i+1,j}^n - 2\varepsilon_{i,j}^n + \varepsilon_{i-1,j}^n}{\Delta x^2} + \frac{\varepsilon_{i,j+1}^n - 2\varepsilon_{i,j}^n + \varepsilon_{i,j-1}^n}{\Delta y^2} \right)$$

Amplify the i-th and j-th harmonics and follow the similar procedure introduced in class. Assuming that $\Delta x = \Delta y$, the stability condition is:

$$\frac{\Delta t}{\Delta x^2} + \frac{\Delta t}{\Delta y^2} \leq \frac{1}{2}$$

Thus, we get the threshold value for d:

$$0.0 \leq d = \frac{\alpha \Delta t^2}{\Delta x} \leq 0.25$$

Let us check if this condition is satisfied by investigating our model. We already checked the case $d = 0.1$. Let us try the values $d = 0.25$ and $d = 0.26$.

Discussion As viewed from Figure-6, the system is indeed stable when $d = 0.25$ at its maximum stable threshold value. However, when we slightly increased the value, at $d = 0.26$, interesting characteristics were shown.

From Figure-7, we can observe the system is stable at the beginning, but at the time stamp $t = 0.40$s, the errors start to amplify. The system diverges and becomes extremely unstable as it proceeds over time. Eventually, we can see the completely diverged system at $t = 1.26$s. In addition, we can observe the system diverges faster if we increase the value of the diffusion number. We can conclude that the amplification of error increases according to the value of the diffusion coefficient.

For the values between $0.5 \leq d \leq 1.0$, the system is *unconditionally* unstable. Therefore, we can conclude that the growth or decay of the error determines whether if the numerical algorithm is stable.

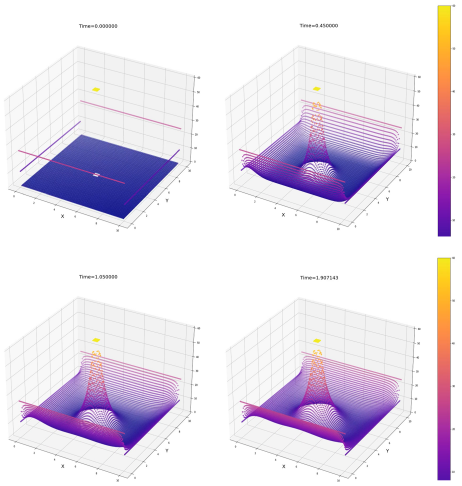

Figure 6: System When $d = 0.25$

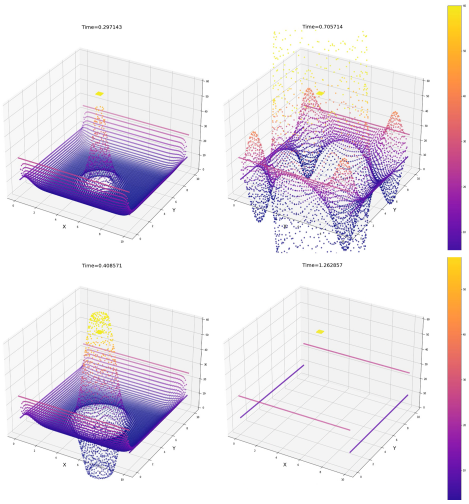

Figure 7: System When $d = 0.26$

2 Problem 2: Burger's Equation

Burger's equation is the combination of hyperbolic, parabolic, and elliptic equations.

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

2.1 Discretized form of the equation

We will use **forward-in-time and backward-in-space** for the 1^{st} derivative, and **centered difference** for the 2^{nd} derivative.

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

Assuming $\Delta x = \Delta y$, we can define the forward in time system as follows:

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

In this expression, a or u is a vector field, and v is the kinematic viscosity. The physical meaning of Burger's equation can be interpreted as a simplified version of the turbulence equation.

2.2 Linear and Non-linear systems

If we consider our system as the representation of a simple fluid dynamics, Burger's equation describes the fluid in spatial location along with the container as time progresses.

Investigation of Linear system

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

When a is constant, we consider the case of a scalar field. Hence, the system will mostly behave according to the value of *kinematic viscosity* or v. We can model Burger's equation as follows:

```
from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt

# Initial and given conditions
v = 0.1 # Kinematic Viscosity
a = 0.5 # Vector field, Scalar case
dx = 0.1
len_x = 10.0
len_y = 10.0
dt = 0.01
end_time = 20.0

x = np.arange(0.0, len_x+dx, dx)   # number of grids, different points of
    x
y = np.arange(0.0, len_y+dx, dx)
U = np.zeros_like(X)  # initially the T is zero
U[:,0] = 20.0
U[:,-1] = 20.0
U[0,:] = 30.0
U[-1,:] = 30.0
U[50:90,40:75] = 150.0

plt.figure(figsize=(20,20))
icounter = -1
for it in np.arange(0.0,end_time+dt, dt):
    icounter = icounter + 1
    if np.mod(icounter,20)==0:
        ax = plt.axes(projection='3d')
        p = ax.scatter(X,Y,U,c=U,cmap='plasma', vmin = 0.0, vmax = 200.0)
        ax.set_xlabel('X',fontsize = 20)
        ax.set_ylabel('Y',fontsize = 20)
        ax.set_zlim([0.0,200.0])
        plt.colorbar(p)
        plt.title('Time=%.6f'%(it),fontsize = 20)
        plt.savefig('%06.6d.jpg'%(icounter))
        plt.cla()
        plt.clf()
    U = U - ((a*dt)/dx)*(2*U-np.roll(U,1,axis=1)-np.roll(U,1,axis=0))+((v*
    dt)/(dx**2))*(np.roll(U,-1,axis=1)+np.roll(U,1, axis=1)+np.roll(U,-1,
    axis=0)+np.roll(U,1, axis=0) - 4.0*U)
# Animation is created same way
```

Listing 5: Modeling of the Linear Burger's equation

Discussion: One can confirm that the system satisfies cyclic boundary conditions since lines 17 to 21 are not specified inside of a loop function. From Figure-8, we can see the characteristics of the system when $v = 0.1$. Since the system satisfies the cyclic boundary condition, we can confirm the fluid inside overflows from one end to another end.

We can consider that Burger's equation describes fluid transportation by means of advection $\vec{v} \cdot \nabla^2$ and diffusion. It can be understood that when $v = 0$, the system becomes a heat equation.

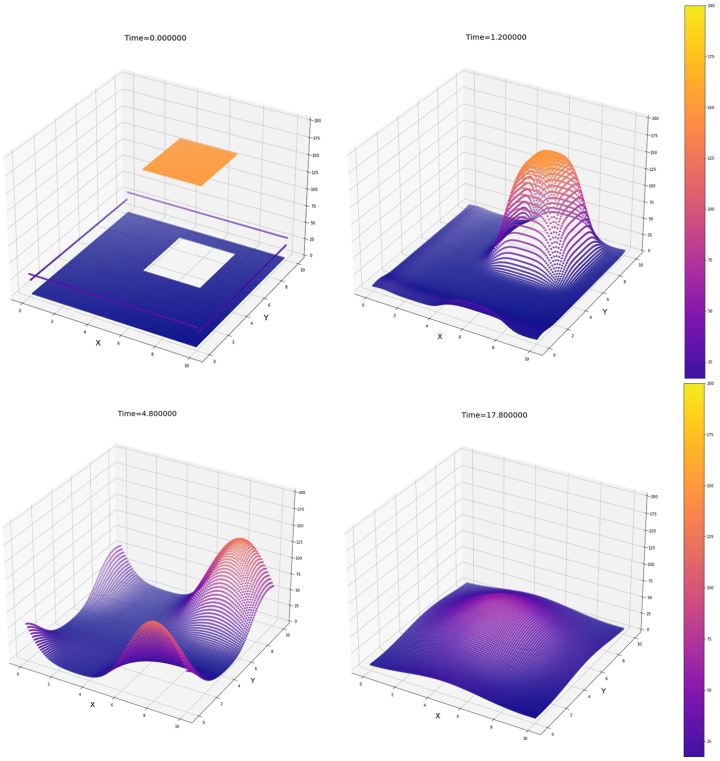

Figure 8: Linear case of Burger's Equation

Investigation of Non-Linear system

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

Discussion: Similar numerical modeling can be implemented on the non-linear case. Unlike the case when $a = const$, we now have 4 different parameters which represent the dynamic contingent of the system. Additional u term indicates the velocity of fluid at spatial and temporal coordinates. The simultaneous presence of non-linear convective term: $u\,(\partial u/\partial x)$ and diffusive term: $v\left(\partial^2 u/\partial x^2\right)$ make the system difficult to interpret.

From Figure-9, one can observe that the system shows the characteristics of dissipation. Starting from the given initial condition, cyclic boundary condition allows the system to naturally dissipate over time and reach its steady-state.

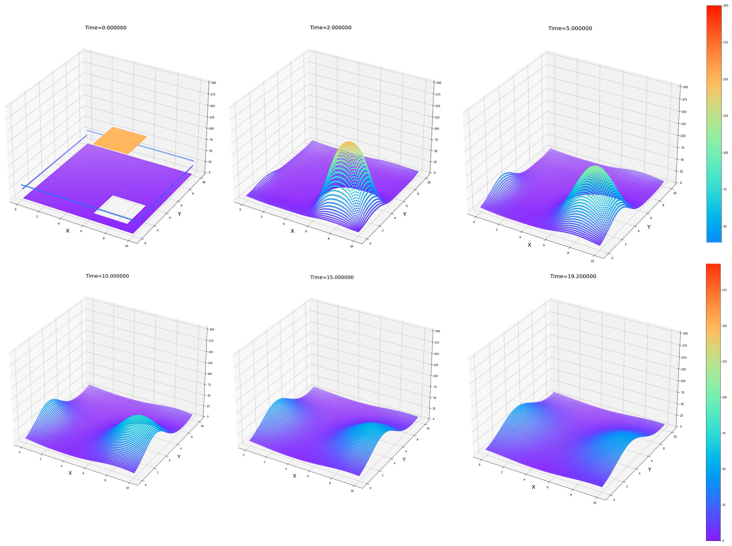

Figure 9: Non-Linear case of Burger's Equation

2.3 Behaviour of u at Mixed boundary conditions

We set a cyclic boundary at one axis and a close-wall boundary at another axis where there is no net flux. Modeling is similar to the way we constructed the Neumann boundary condition for the heat equation. We once again introduce the notion of Ghost point. Because we used **forward-in-time and backward-in-space** for the 1^{st} derivative, and **centered difference** for the 2^{nd} derivative, it is necessary to specify Ghost points everywhere applicable.

Mathematically, it can be expressed as follows.

$$G = \frac{u^n_{i,j} - u^n_{i-1,j}}{2\Delta x}$$

$$u^{n+1}_{0,j} = u^n_{0,j} - \frac{a\Delta t}{\Delta x}\left(2u^n_{0,j} - u^n_{-1,j} - u^n_{0,j-1}\right) + \frac{v\Delta t}{\Delta x^2}\left(u^n_{1,j} + u^n_{-1,j} + u^n_{0,j+1} + u^n_{1,j-1} - 4u^n_{0,j}\right)$$

$$u^{n+1}_{0,j} = u^n_{0,j} - \frac{a\Delta t}{\Delta x}\left[2u^n_{0,j} - \left(u^n_{0,j} - 2\Delta x G\right) - u^n_{0,j-1}\right] + \frac{v\Delta t}{\Delta x^2}\left[u^n_{1,j} + \left(u^n_{0,j} - 2\Delta x G\right) + u^n_{0,j+1} + u^n_{1,j-1} - 4u^n_{0,j}\right]$$

Program:

```
# Parameter values are set same as the previous implementation

v = 0.1
#v = 0.01
#v = 0.25
G = 0  # Zero net flux at x-axis
U = np.zeros_like(X)  # initially the T is zero

# Neumann boundary condition
```

```
U[0,:] = U[0,:] - ((a*dt)/dx)*(2*U[0,:]-(U[1,:]-2*dx*G)-np.roll(U[0,:],1,
    axis=0))+((v*dt)/dx**2)*(U[1,:]+(U[1,:]-2*dx*G)+np.roll(U[0,:],-1, axis
    =0)+np.roll(U[0,:],1, axis=0) - 4.0*U[0,:])
U[10:50,50:70] = 200

plt.figure(figsize=(20,20))
icounter = -1
for it in np.arange(0.0,end_time+dt, dt):
    icounter = icounter + 1
    if np.mod(icounter,50)==0:
        ax = plt.axes(projection='3d')
        p = ax.scatter(X,Y,U,c=U,cmap='plasma', vmin = 0.0, vmax = 200.0)
        ax.set_xlabel('X',fontsize = 20)
        ax.set_ylabel('Y',fontsize = 20)
        ax.set_zlim([0.0,200.0])
        plt.colorbar(p)
        plt.title('Time=%.6f'%(it),fontsize = 20)
        plt.savefig('%06.6d.jpg'%(icounter))
        plt.cla()
        plt.clf()
    # Adjusted U to prevent cyclic overflow
    U[1:-1,:] = U[1:-1,:] - ((a*dt)/dx)*(2*U[1:-1,:]-np.roll(U[1:-1,:],1,
    axis=1)-np.roll(U[1:-1,:],1,axis=0))+((v*dt)/dx**2)*(np.roll(U
    [2:,:],-1,axis=1)+U[0:-2,:]+np.roll(U[1:-1,:],-1, axis=0)+np.roll(U
    [1:-1,:],1, axis=0) - 4.0*U[1:-1,:])
    # Neumann Boundary for each iteration
    U[0,:] = U[0,:] - ((a*dt)/dx)*(2*U[0,:]-(U[1,:]-2*dx*G)-np.roll(U
    [0,:],1,axis=0))+((v*dt)/dx**2)*(U[1,:]+(U[1,:]-2*dx*G)+np.roll(U
    [0,:],-1, axis=0)+np.roll(U[0,:],1, axis=0) - 4.0*U[0,:])
# Animation is created same way
```

Listing 6: Mixed boundary condition for Burger's equation

Discussion:

We test the model for different kinematic viscosity values to analyze the stability and the characteristic of the system. Same Neumann and cyclic boundary conditions are introduced for every value of v. Generally, we can see that the cyclic areas overflow to one another, while Neumann boundary acts like a wall.

From Figure-10, we can see the comparison between the decay rate of error over time for values $v = 0.01$, $v = 0.2$, and $v = 0.25$. We can observe that the decay rate of error amplifies as we increase the value for v. The higher the value of v, the faster it reaches the steady-state (within the stable range of values). When the kinematic viscosity reaches the value $v = 0.25$, the system becomes extremely unstable and completely diverges in only 4 seconds. Any values higher than $v = 0.25$ will cause a system instability regardless of other factors.

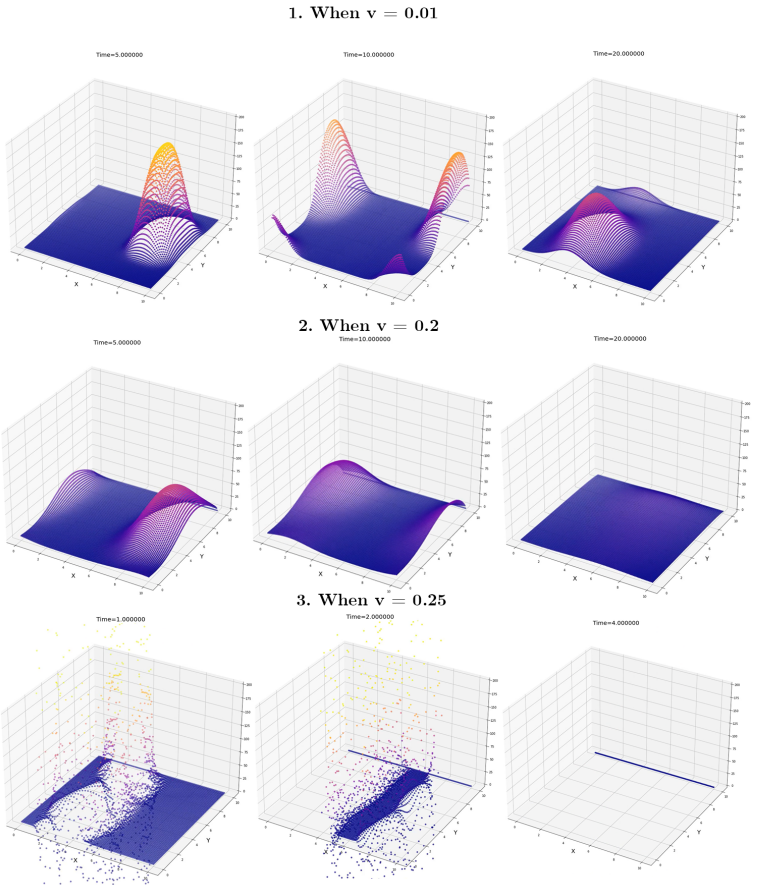

Figure 10: Comparison between Mixed boundary conditions for Burger's equation

3 Problem 3: 1-D Advection Equation

We have given the following 1-dimentional advection equation:

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

At $t = 0$,

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

When we construct advection modeling, it is crucial to choose an applicable approximation method. For instance, the FTCS method (Figure - 11) is not useful if we consider the Von Neumann Stability. The system is unconditionally unstable for any value of *Courant number*-C. In terms of physical phenomenon, the advection generally relies on the flow of the upstream. That is why the forward differencing method is not suitable for advection modeling. For the following sections, we will implement three different approximation methods that use the concept of *interpolation* and analyze their behavior.

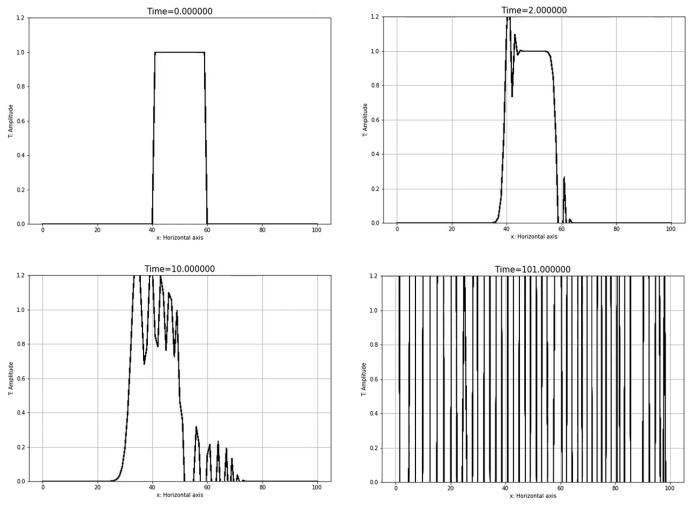

Figure 11: Time series - FTCS method - Amplification of error over time

3.1 Construction of modeling

3.1.1 Upwind Scheme modeling

Upwind scheme is a discretization method that calculates the derivatives in the direction of the flow. Let us construct the first-order upwind scheme:

$$F_{i\rightarrow iup} = f_i^{n+1} = f_i^n + \frac{u\Delta t}{\Delta x}\left(f_{iup}^n - f_i^n\right)$$

Where the upwind direction must be defined as:

$$iup = i - sign\left(u\right) = \begin{cases} i-1, & u > 0 \\ i+1, & u < 0 \end{cases}$$

Program:

```
import numpy as np
import matplotlib.pyplot as plt
u = 1.0 # wave speed
dx = 1.0
c = 0.1 # Courant number /defines the stability of the system/
dt = c*dx/u
len_x = 100.0
len_y = 100.0
end_time = 200.0

sign = int(np.sign(u))  # Sign of the wave speed

x = np.arange(0.0, len_x+dx, dx)  # number of grids, different points of x
t = np.arange(0.0, len_y+dx, dx)
X, T = np.meshgrid(x,t)
f = np.zeros_like(X)  # initially the f is zero

```

```
#Intitial Condition
f[41:60]=1.0 # Zero elsewhere
plt.figure(figsize=(10,7))
icounter = -1
for it in np.arange(0.0,end_time+dt, dt):
    icounter = icounter + 1
    if np.mod(icounter,20)==0:
        plt.plot(T,f,'k-',label='Upwind Scheme')
        plt.xlabel('x: Horizontal axis')
        plt.ylabel('T: Amplitude')
        plt.title('Time=%.6f'%(it),fontsize = 15)
        plt.ylim(0.,1.2)
        plt.savefig('%06.6d.jpg'%(icounter))
        plt.cla()
        plt.clf()
        plt.close
        plt.grid()
    f = f + c*(np.roll(f,sign,axis=0)-f)
# Animation is created same way
```

Listing 7: Program for Upwind Scheme

Stability and Accuracy

Courant number determines the stability of the system.

$$C \equiv \left| \frac{u\Delta t}{\Delta x} \right|$$

$$f_i^{n+1} - f_i^n = (1-C)\, f_i^n + C f_{i+1}^n - f_i^n$$

$$f_i^{n+1} - f_{i+1}^n = (1-C)\, f_i^n + C f_{i+1}^n - f_{i+1}^n$$

$$\left(f_i^{n+1} - f_i^n\right)\left(f_i^{n+1} - f_{i+1}^n\right) = -C\,(1-C)\left(f_i^n - f_{i+1}^n\right)^2$$

For system to satisfy this condition the range of C must be:

$$0 \leq C \leq 1$$

Discussion:

We can confirm from Figure-12 that the system is indeed stable for the values between $0 \leq C \leq 1$. However, when we test the value $C = 1.01$ and any other value higher than 1.0, the system becomes unconditionally unstable. In the third row of Figure-12, we can observe an error which is referred to as Gibb's error. We see the same horn-like pattern or jump discontinuities when we expand the Fourier series of a periodic function.

In terms of accuracy, the higher the value of *Courant number* we choose, the more accurate the system behaved. At the value $C = 1.00$, the system is in its most accurate and stable condition. We can conclude that when $C = 1.00$, both analytical and numerical amplification magnitude are the same and equal to 1 and there is no dispertion error. What is so interesting is that as C decreases, the numerical amplitude also decreases over time. Another observation is for the values between $0.5 < C < 1.0$ the system moves slower than the analytical solution. One can thoroughly examine from the provided animations.

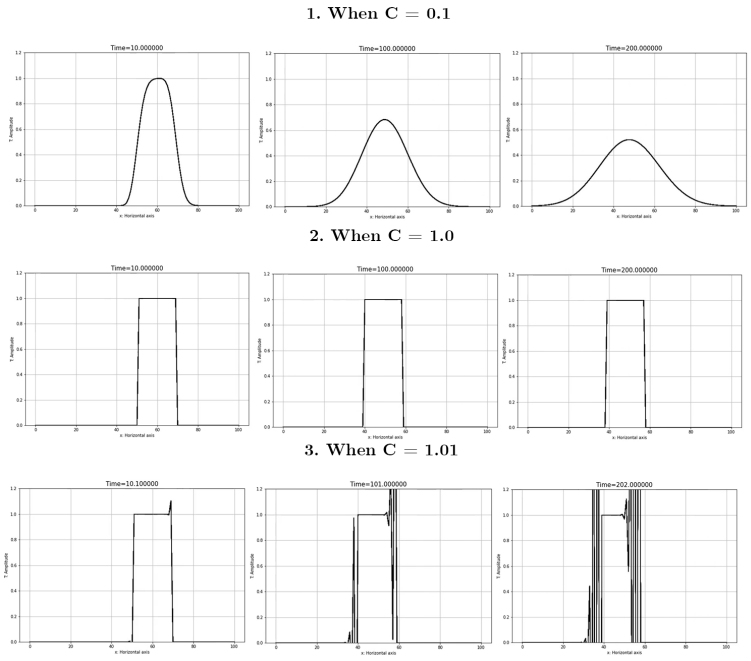

Figure 12: Upwind Scheme: Comparison between different Courant numbers

3.1.2 Leith's Method

Leith's or Lax-Wendroff Method is one of the conservative classes of approximation that was developed to suppress the dissipation error by increasing the order of interpolation. In this report we used the second-order approximation.
Consider a second order polynomial - $F_i(x) = a_i X_i^2 + b_i X_i + c_i$. Then:

$$c_i = f_i^n$$

$$b_i = \frac{1}{2\Delta x}\left(f_{i+1}^n - f_{i-1}^n\right)$$

$$a_i = \frac{1}{2\Delta x^2}\left(f_{i+1}^n - f_i^n + f_{i-1}^n\right)$$

Therefore, for the next time-step:

$$f_i^{n+1} = a_i\left(u\Delta t^2\right) - b_i\left(u\Delta t + c_i\right)$$

By making small adjustments to the previous code, we get the following results:

```
f = np.zeros_like(X)  # initially f is zero
f[41:60]=1.0
sign = int(np.sign(u))
# Set loop function for each time step /similar procedure/
    A = (1/(2*(dx**2)))*(np.roll(f,-1,axis=0)-2*f+np.roll(f,1,axis=0))
    B = (1/(2*dx))*(np.roll(f,-1,axis=0)-np.roll(f,1,axis=0))
    C = f
    f = A*((u*dt)**2)-B*(u*dt)+C
```

Listing 8: Leith's Method - Advection

Stability and Accuracy:

If we consider the solution periodic and express in matrix form, we get the following:

$$f^{n+1} = A^{n+1} f^n$$

This way, we can analyze the stability in a quantitative manner.

$$A = a_0 + a_{-1} e^{-ik\Delta x} + a_1 e^{ik\Delta x}$$

$$|A|^2 = 1 - 4C^2 \left(1 - 4C^2\right) sin\left(k\Delta x\right)$$

The system is stable if $|A| \leq 1$. Which leads to:

$$C = \left| \frac{u\Delta t}{\Delta x} \right| \leq 1$$

Discussion: Same as the previous upwind scheme, the stable region for the courant number is between 0 to 1. From Figure-13, We can see a decrease in dissipation error compared to the upwind scheme. In addition, similar to the upwind scheme, the higher the value of *Courant number* we choose, the more accurate the system behaved. When we tried the value $C = 1.01$, the amplification of error was faster than the upwind scheme. This can be explained by the higher order of approximation.

Comparison between different C - values can be observed as well. We can conclude that when $C = 1.00$, both analytical and numerical amplification magnitude are the same and equal to 1 and there is no dispertion error. Also, as C decreases, the numerical amplitude also decreases over time.

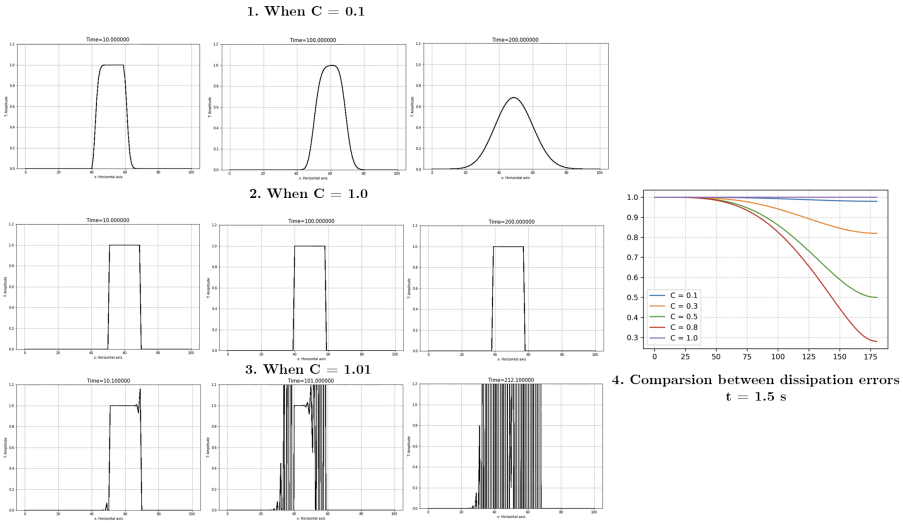

Figure 13: Upwind Scheme: Comparison between different Courant numbers and Dissipation errors

3.1.3 CIP Method

Lagrange Interpolation method requires more nodes and more conditions as the order of polynomial increases. In this section, we will look at the Hermite Interpolation which uses cubic-spline.

Constrained Interpolation Method We need another function aside from f to construct a model. Define a gradient:

$$F_i(x) = a_i X_i^3 + b_i {X_i}^2 + c_i X_i + d_i$$

$$g_i^n = \left(\frac{\partial f}{\partial x}^n{}_i\right)$$

To simplify the calculation of g, let us derive an interpolation function for g_i^n.

$$\frac{\partial}{\partial x}\left[\frac{\partial f}{\partial t} + u\left(\frac{\partial f}{\partial x}\right)\right]$$

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

We can see that the gradient of the system can also be advected. Therefore, the 1-D Advection by CIP method can be modeled as:

$$a_i = \frac{-2\left(f_{iup}^n - f_i^n\right)}{\Delta x_{i\to iup}^3} + \frac{\left(g_{iup}^n + g_i^n\right)}{\Delta x_{i\to iup}^2}$$

$$b_i = \frac{-3\left(f_i^n - f_{iup}^n\right)}{\Delta x_{i\to iup}^2} + \frac{\left(2g_i^n + g_{iup}^n\right)}{\Delta x_{i\to iup}}$$

$$iup = i - sign(u) = \begin{cases} i-1, & u_i^n > 0 \\ i+1, & u_i^n < 0 \end{cases}$$

Let $\xi_i^n = -u_i^n \Delta t$. Then,

$$f_i^{n+1} = a_i\xi_i^3 + b_i\xi_i^2 + g_i^n\xi_i + f_i^n$$

$$g_i^{n+1} = \left(3a_i\xi_i^2 + 2b_i\xi_i + g_i^n\right)\left[1 - \left(\frac{\partial u}{\partial x}\right)_i^n \Delta t\right]$$

We can set the program as the following:

```
f = np.zeros_like(X)  # initially f is zero
f[41:60]=1.0
sign = int(np.sign(u))
# Set loop function for each time step /similar procedure/
    A = -2*(np.roll(f,sign,axis=0)-f)/(dx*3) + (g + np.roll(g,sign,axis=0)
   )/(dx**2)
    B = -3*(np.roll(f-f,sign,axis=0))/(dx*2) + (2g + np.roll(g,sign,axis
   =0))/(dx)
    Xi = -u*dt
    f = f + A*(Xi**3) + B*(Xi**2) + g*Xi
    g = g + 3*A*(Xi**2) + 2*B*Xi
```

Listing 9: CIP Method - Advection

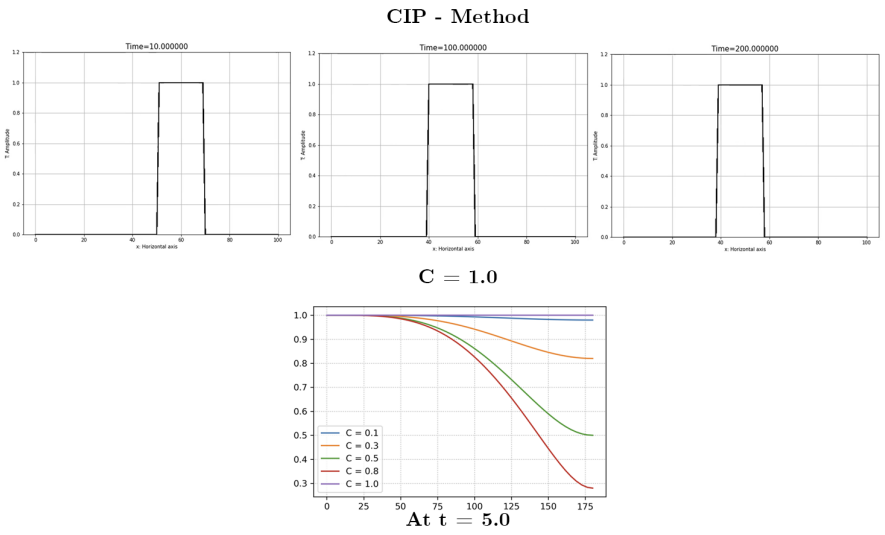

Figure 14: 1D Advection: CIP - Method

3.1.4 Analytical Solution

Let us try to solve this problem using D'Alembert's solution.
Perform variable conversion:

$$(t, x) \rightarrow (\tau, \xi) = \begin{cases} \tau = x - ct, & \text{c: } const \\ \xi = x + ct, \end{cases}$$

From Chain rule:

$$\frac{\partial f}{\partial t} = \frac{\partial \tau}{\partial t}\frac{\partial f}{\partial \xi} + \frac{\partial \tau}{\partial t}\frac{\partial f}{\partial \xi} = -c\frac{\partial f}{\partial \tau} + c\frac{\partial f}{\partial \xi} = c\left(\frac{\partial f}{\partial \xi} - \frac{\partial f}{\partial \tau}\right)$$

$$\frac{\partial f}{\partial x} = \frac{\partial \tau}{\partial X}\frac{\partial f}{\partial \xi} + \frac{\partial \tau}{\partial x}\frac{\partial f}{\partial \xi} = \frac{\partial f}{\partial \tau}\frac{\partial f}{\partial \xi} = \left(\frac{\partial f}{\partial \xi} + \frac{\partial f}{\partial \tau}\right)$$

Now, if we substitute into the advection equation and assuming $c = u$, we get:

$$c\left(\frac{\partial f}{\partial \xi} - \frac{\partial f}{\partial \tau}\right) + \left(\frac{\partial f}{\partial \xi} + \frac{\partial f}{\partial \tau}\right)$$

$$\left(\frac{\partial f}{\partial \xi}\right) = 0$$

Integrate with respect to ξ,

$$f\left(t, x\right) = C(\tau)$$

Assume f at the initial time:

$$f\left(0, x\right) = C(\tau) = F(x) = F(\tau)$$

Therefore, with the following initial value, the analytical solution obtained from D'Alembert's solution is:

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

$$f(t,x) = f(0, x - ut)$$

Discussion: From the previous numerical solutions, we can grasp the idea that this is simply a shifting function of its initial value. We get the following animation by modeling the 1-D advection equation.

From Figure-15, we can confirm it is being shifted by ut in every time step. Also, we can confirm from the previous numerical modelings that systems with $C = 1.0$ courant numbers are in agreement with the analytical solution.

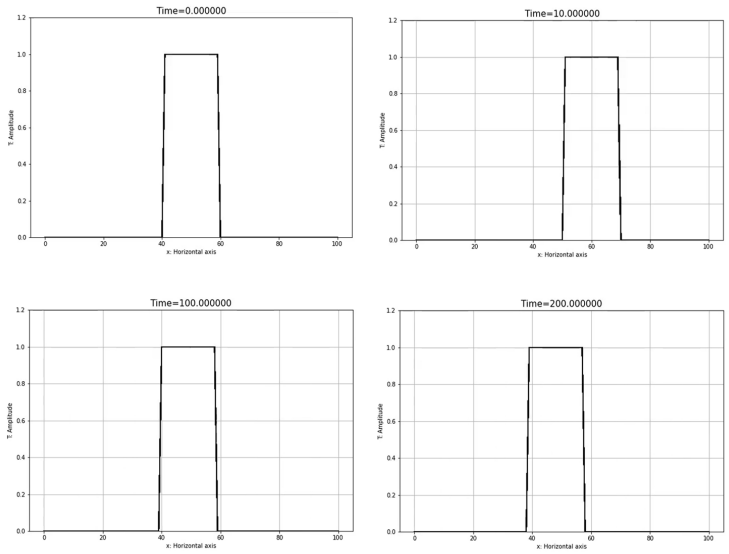

Figure 15: 1D Advection: Time series of exact solution

4 Conclusion

In this report, we performed various numerical modeling techniques and computations on three different physical phenomenons. For the first problem, diffusion equation, we analyzed the steady-state using *Successive over-relaxation* method or SOR. We observed the SOR method adjusts the value of T for each iteration until the difference between the next time-step is small enough to be negligible. We also observed the computation time is dependent on the value of omega-ω. Then, we analyzed various boundary conditions using discretization methods. Dirichlet Boundary condition indicates the values that a solution needs to take along the boundary of the domain. Whereas, the Neumann boundary condition defines the flux of the system. Important step when modeling is the concept of *Ghost point*.

For stability, we considered Von Neumann Stability to quantitatively understand the system.

We observed the decay growth of error or the amplification of error increases according to the value of the parameters. This is why chose certain parameters with great care such as *diffusion number* and *courant number*.

For Problems 2 and 3, we closely observed why there are many approximation techniques are used in numerical modeling. For instance, from Figure-11, we can see that the FTCS method is not useful when modeling advection equation. Also, in the case of Burger's equation, we used backward-in-space for the first derivative. This is because Burger's equation is the combination of hyperbolic, parabolic, and elliptic equations. Hence, we must carefully consider which approximation technique is the most optimal for the system.

Finally, for the advection modeling, we observed the existence of dissipation errors in every numerical approximation. If we increase the courant number higher than its stable region, we observed the Gibbs error which amplifies over time. Dissipation error can be suppressed by a higher order of approximation. In addition, the value of the courant number hugely affects the stability and the accuracy of computation. Generally, the higher the value of the Courant number we choose, the more accurately the system behaved.

5 References

Computational Fluid Dynamics by T.J. Chung

Lecture materials, Prof. Varquez Alvin Christopher Galang, PDE: TSE.M202-01