PARTIAL DIFFERENTIAL EQUATIONS FOR SCIENCE AND ENGINEERING

Final Report

NOVEMBER 27, 2017

Thumwanit Napat

16B00133

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,

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

Discuss the following (add figures if necessary):

1. Influence of the initial condition. Test various initial conditions or distributions.
2. Investigate various boundary conditions:
   a. Dirichlet Boundary condition
   b. Neumann Boundary condition
3. Investigate what is the influence of $d = \frac{\alpha \Delta t}{\Delta x^2}$ by testing various values for *d*. What are the threshold values for *d*?
4. Show three time steps (start, middle, and almost steady-state). Steady-state means the variations with time are almost negligible.

A. Effect of Initial Condition (By Neumann) [Q.1]

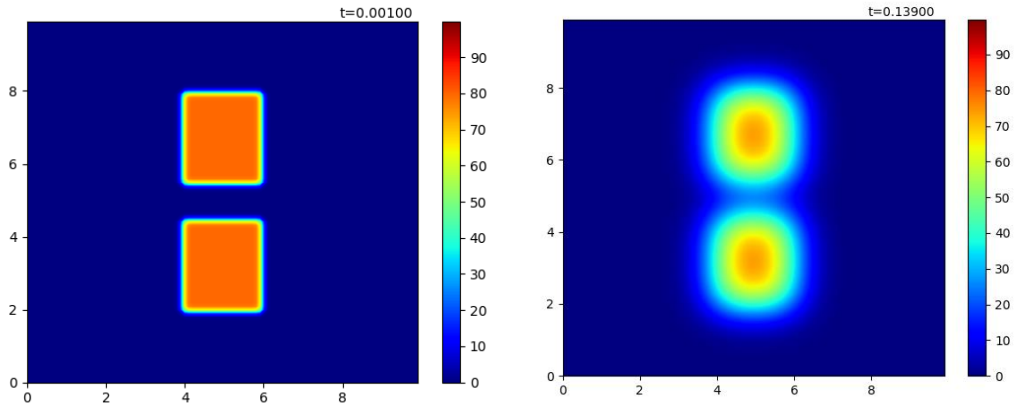
Initial Condition

```
dx = 0.1
dy = dx
alpha = 1.
grid_x = 100
grid_y = 100
nt = 100
d = 0.1
dt = d * (dx**2)/alpha

def init():
    T = np.zeros((grid_x,grid_y))
    T[70:grid_x, 70:grid_y] = 80
    return T
T =init()
```

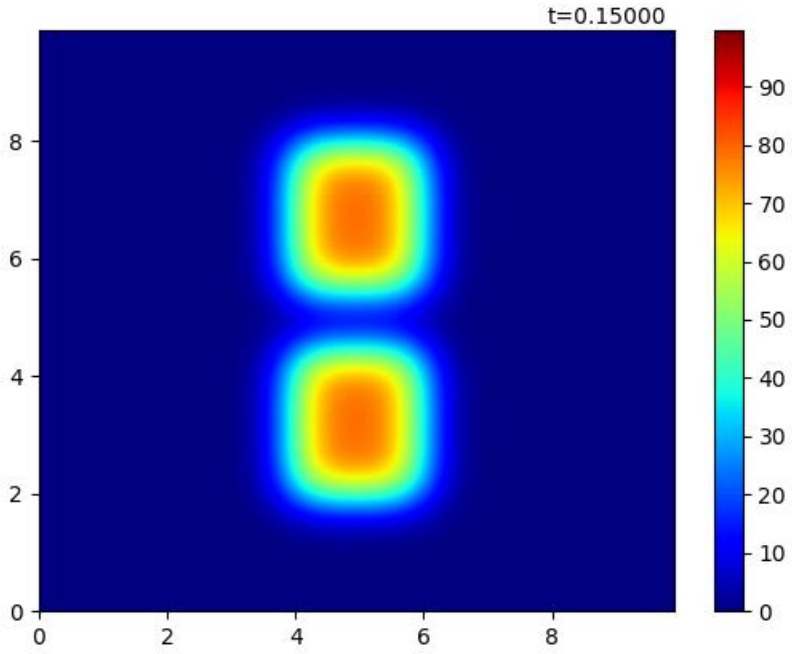
- couple blobs

```
def init():
    T = np.zeros((grid_x,grid_y))
    T[20:45, 40:60] = 80.
    T[55:80, 40:60] = 80.
    return T
```



If we set new value for our initial condition, alpha.

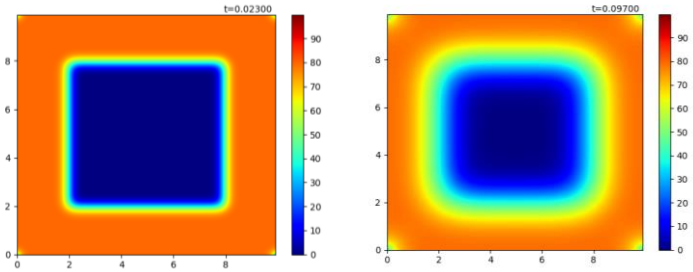
```
1.      alpha = 0.5
```



The dt per each timestep will increase and use longer actual time to disperse because with same d value the change per timestep is the same. (can be compared with the above figure that spent only 0.139 while this figure spent 0.150)

- Heat wall

```
def init():
    T = np.zeros((grid_x,grid_y))
    T[0:20, :] = 80.
    T[grid_x-20:grid_x, :] = 80.
    T[:,0:20] = 80.
    T[:,grid_y-20:grid_y] = 80.
    return T
```



B. Observation of Conditions [Q.2+4]

Desired Initial

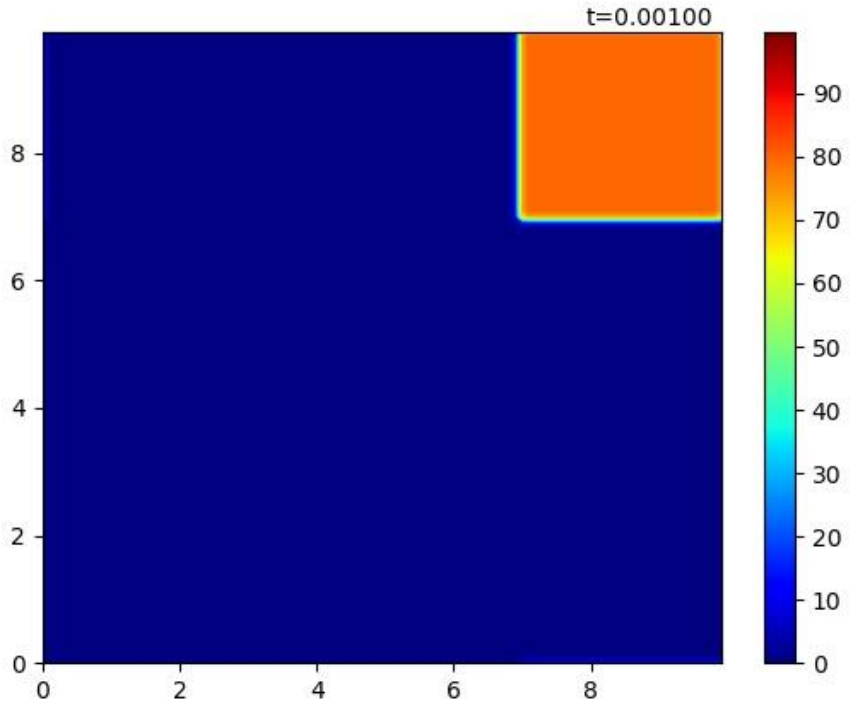


Figure 1 initial condition for observation

Here is the visualization for the initial values and init() function above. We initiate the blob of heat at the corner to observe each method.

Plot function

```
1.  def plot_activate(X,Y,n):
2.      global T
3.      plt.cla()
4.      plt.clf()
5.      plt.xlim(0.,np.max(x))
6.      plt.ylim(0.,np.max(x))
7.      cl = plt.contourf(X,Y,T,levels,cmap=cmap)
8.      plt.colorbar(cl)
9.      plt.text(np.max(x)*0.8,np.max(y)+dy,"t=%01.5f"%(dt*n))
```

Cyclic

```
def cyc(n, plot=True):  #Input plot argument to decide whether to plot or not (since plot spent much time)
    global T, icount
    Tn = T.copy()
    T = Tn+d*(np.roll(Tn,1,axis=0)+np.roll(Tn,-1,axis=0)+np.roll(Tn,1,axis=1)+np.roll(Tn,-1,axis=1)-4*Tn)
    if plot:
        plot_activate(X,Y,n)
    icount += 1
```

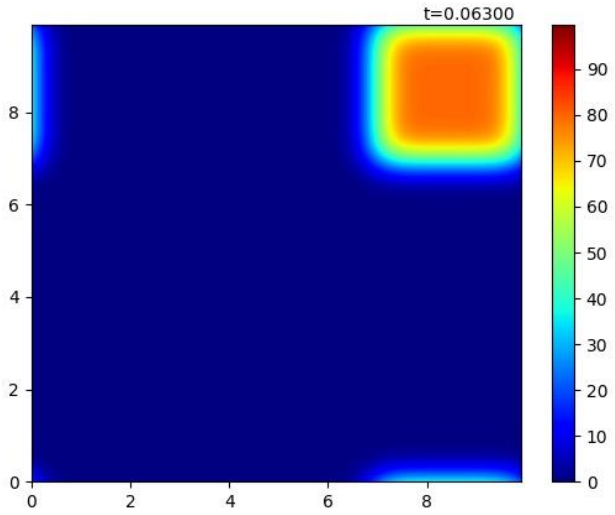


Figure 2 middle state of cyclic condition

Unfortunately, our np.roll() function rolls over the matrix and the temperature somehow leaks to the other side. However, the dispersion can be observed clearly.

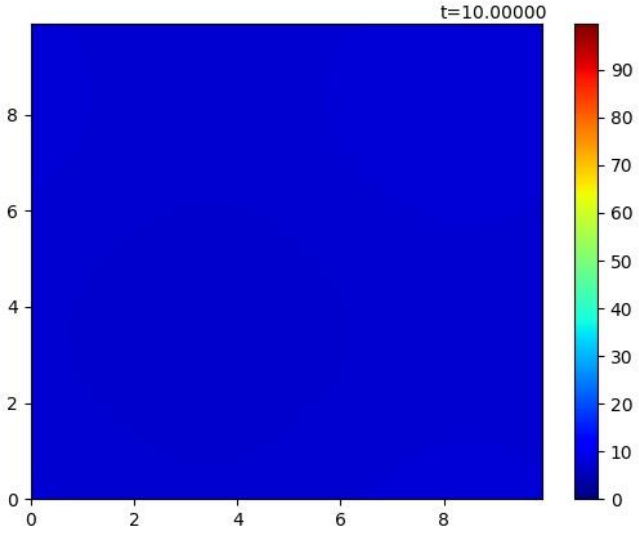


Figure 3 steady state of cyclic condition

The above figure is the steady state where the heat uniformly distributed through the plane. We will compare it later with Dirichlet and Neumann.

Dirichlet

```
8.  def dir(n,plot=True):
9.      global T, icount
10.     T[0,:], T[:,0], T[grid_x-1,:], T[:,grid_y-1] = 0,0,0,0
11.     Tn = T.copy()
12.     T = Tn+d*(np.roll(Tn,1,axis=0)+np.roll(Tn,-
    1,axis=0)+np.roll(Tn,1,axis=1)+np.roll(Tn,-1,axis=1)-4*Tn)
13.     if plot:
14.         plot_activate(X,Y,n)
15.     icount += 1
```

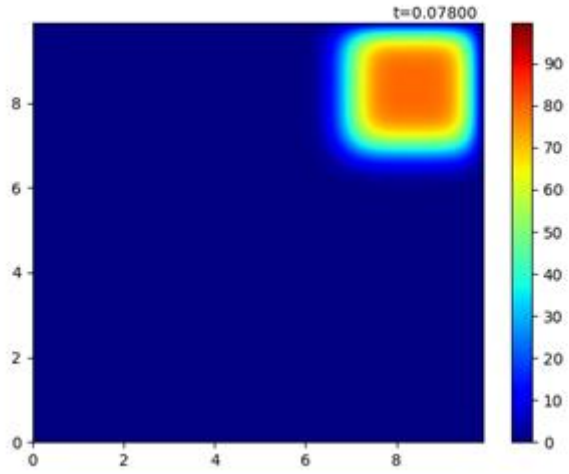


Figure 4 middle state of Dirichlet condition

Dirichlet dispersed in the same way as Cyclic did but not leak to the other side. Due to the wall condition, the wall itself acted like “black hole” that heat suddenly lost over there.

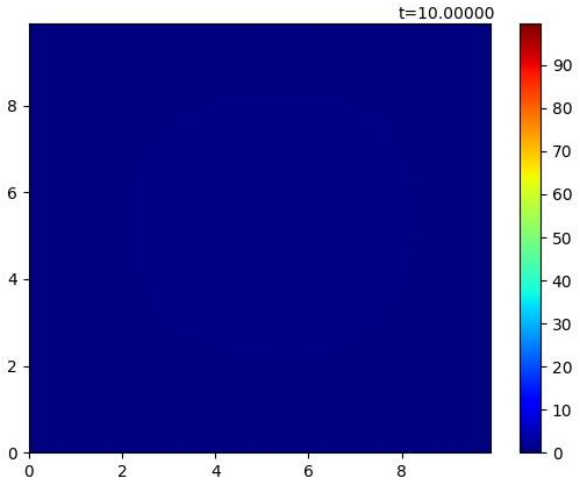


Figure 5 final state of Dirichlet condition

If we compare steady state with the cyclic condition, it is somehow darker, to be said, lower heat. This is because the wall condition did not conserve the heat.

Neumann (G=0)

```
def neu(n, g=0,plot=True):
    global T, icount
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    left[0,:],right[grid_x-1,:],up[:,grid_y-1],down[:,0] = 0,0,0,0  #set zero for the term that cyclically protruded the wall
    T = Tn+d*(up+down+right+left-4*Tn)
    T[0,1:grid_y-1] += d * ((-2 * dx * g + Tn[1,1:grid_y-1]) > 0) * (-2 * dx * g + Tn[1,1:grid_y-1]) #prevent negative value due to G which may crash the plot
    T[grid_x-1,1:grid_y-1] += d * ((-2 * dx * g+Tn[grid_x-2,1:grid_y-1]) > 0) * (-2 * dx * g+Tn[grid_x-2,1:grid_y-1])
    T[1:grid_x-1,0] += d * ((-2 * dx * g+Tn[1:grid_x-1,1]) > 0) * (-2 * dx * g+Tn[1:grid_x-1,1])
    T[1:grid_x-1,grid_y-1] += d * ((-2 * dx * g+Tn[1:grid_x-1,grid_y-2]) > 0) * (-2 * dx * g+Tn[1:grid_x-1,grid_y-2])
    if plot:
        plot_activate(X,Y,n)
    icount += 1
```

```
T[0,1:grid_y-1] += d * ((-2 * dx * g + Tn[1,1:grid_y-1]) > 0) * (-
                    2 * dx * g + Tn[1,1:grid_y-1])
```

$$T_{0,y} = d(T_{-1,y} + T_{1,y} + T_{0,y-1} + T_{0,y+1}) \overset{Ghost}{\Rightarrow} d(T_{-1,y})$$
$$T_{-1,y} = T_{1,y} - 2Gdx$$

As the equation above, we need to add the ghost term as following. However, in the case that $T_{1,y} = 0, G > 0$, it will cause $T_{-1,y} < 0$ which may make some area become negative and the plot will break down. In order to prevent that problem, we kept only positive values in the ghost wall; otherwise, set them to zero.

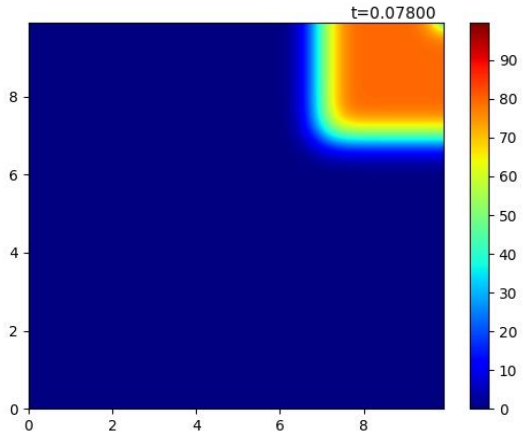


Figure 6 middle state of Neumann condition

Neumann's condition has a meaning of imposing heat flux through the wall so we can see that there is a heat around the wall.

If we compare with Dirichlet, the wall and the one beside the wall have almost the same amount of heat. Notice that np.roll() is implemented; however, we set the values of those rows or columns that went across the plane to zero. When g=0 the effect from the ghost point is going to be as following.

```
T[0,1:grid_y-1] += d * Tn[1,1:grid_y-1]
```

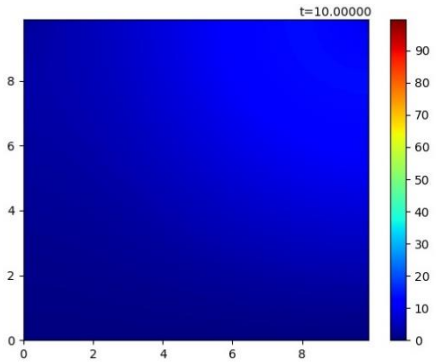


Figure 7 almost final state of Neumann condition

It is pretty sure that steady state should be uniformly distributed. However, with same time (10 seconds or 10000 loops) as cyclic and dirichlet condition, more heat is conserve than dirichlet but not well distributed as cyclic because heat cannot go through the wall.

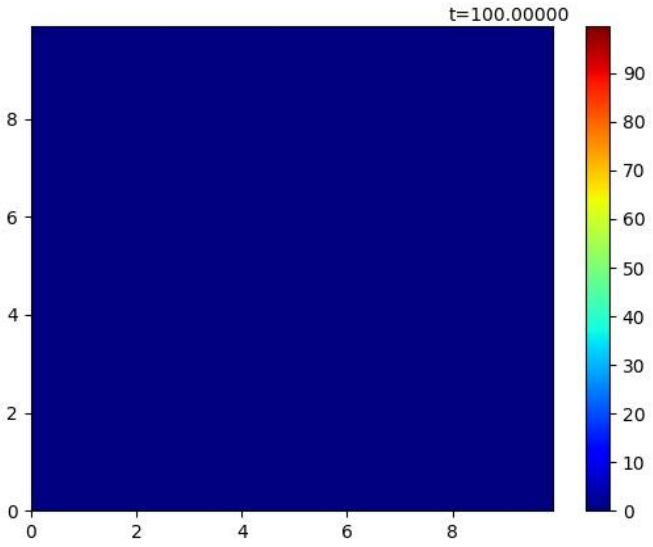


Figure 8 final state of Neumann condition

In order to observe the final state of Neumann condition, we loop it 10 times more (100000) then we reach the steady heat plane which looks as same as Dirichlet because heat lost through time.

Neumann (G=20)

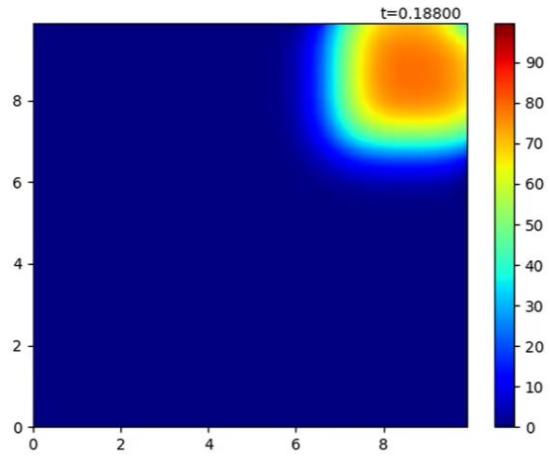


Figure 9 middle state of Neumann condition (G≠ 0)

With G >0 the length of the dx affected the heat of the ghost point. We can see that there is some gradient of heat near the wall.

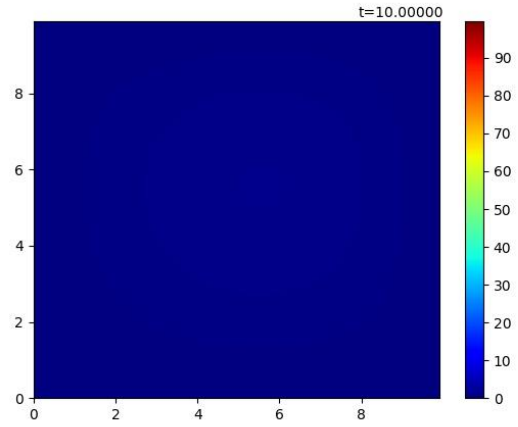


Figure 10 final state of Neumann condition (G≠ 0)

Within the system, heat loss by dx so the final state of this condition looks similar to Dirichlet condition.

Final State Sum up

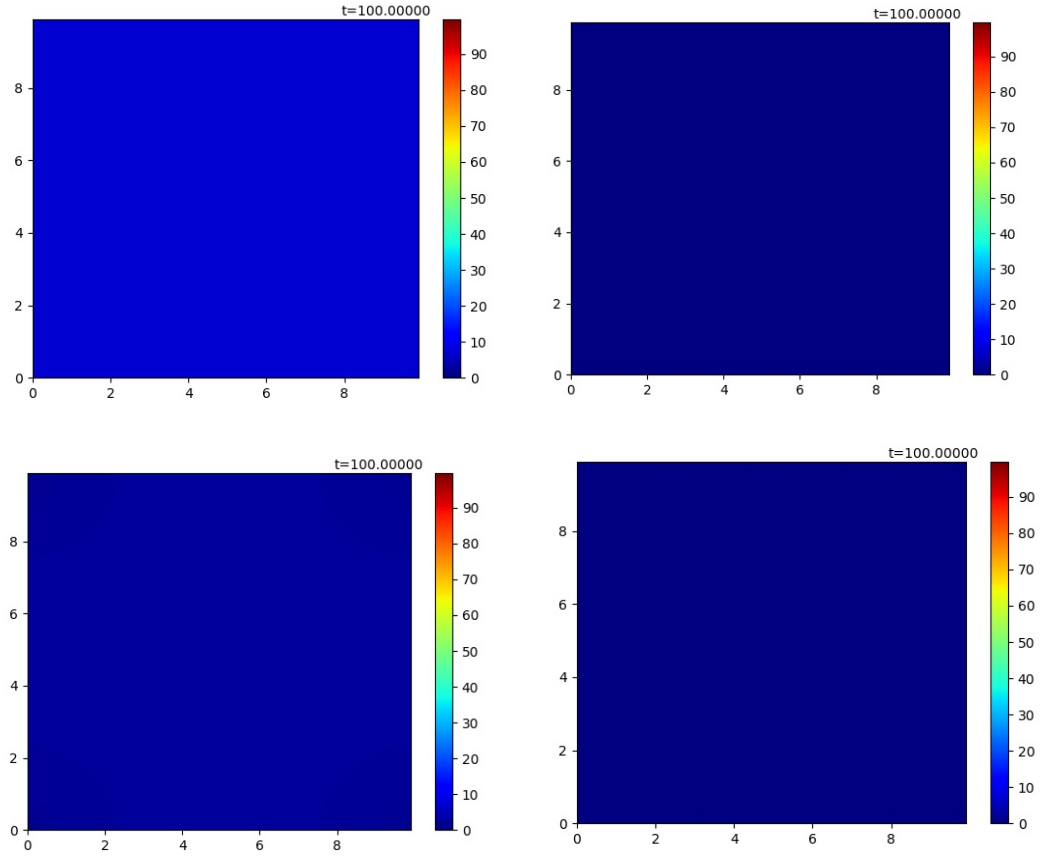


Figure 11 Final State of Every Condition (Top: cyclic, dirichlet, bottom: neumann (G=0),neumann (G=20))

The figure above shows all of the steady state for 100000 loops of timesteps which assure that every condition can reached their steady state (negligible). As described above, Dirichlet and Neumann(G=20) yielded similar result while Neumann and Cyclic are brighter or left more heat in order. To confirm the different between all of the result, we sum up the heat of the plane at the last timestep. Interestingly, cyclic condition did not lose its heat ($80 \times 30 \times 30 = 72000$).

```
1.  n = 100000
2.  for i in range(n):
3.      cyc(1, False)  #(Pass False argument)Don't plot! Update value only
4.  plot_activate(X,Y,n)  #Now let's plot
5.  plt.savefig('cyc/cyc-end.jpg')
6.  print('cyc-end-heat-sum:%03.5f'%(np.sum(T)))
```

cyc -sum:72000.0000, dir -sum:0.00004, neuG0 -sum:24841.49328, neuG20 -sum:0.00013

C. Effect of d's value [Q.3]

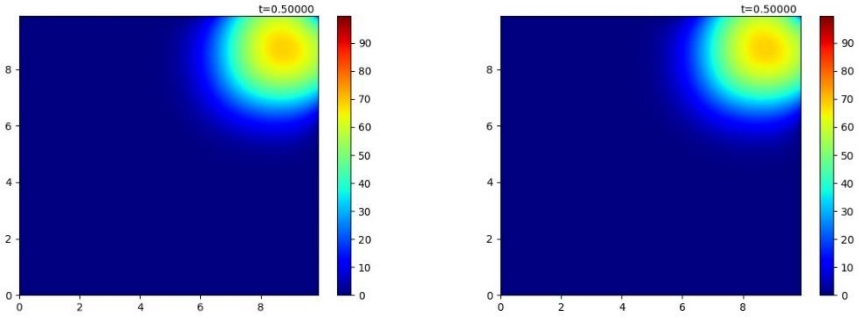


Figure 12 d=0.25/0.1 200,500 loops

From the figure, we used Neumann's condition with G=20 both for d=0.25 and d=0.1. In order to obtain the state of a same actual time we need to iterate inverse proportionally to the value d. For the case d=0.25 we used 200 loops, d=0.1 500 loops.

$$T = N_1 dt_1 = N_1 \frac{d_1 \times dx^2}{\alpha}$$

$$T = N_2 dt_2 = N_2 \frac{d_2 \times dx^2}{\alpha}$$

$$\therefore N_1 / N_2 = d_2 / d_1$$

We can observe clearly that the change of d value did not affect the diffusion because the figures are completely same. However, what did affect the diffusion is alpha value (mentioned above in the section A. Effect of Initial Condition)

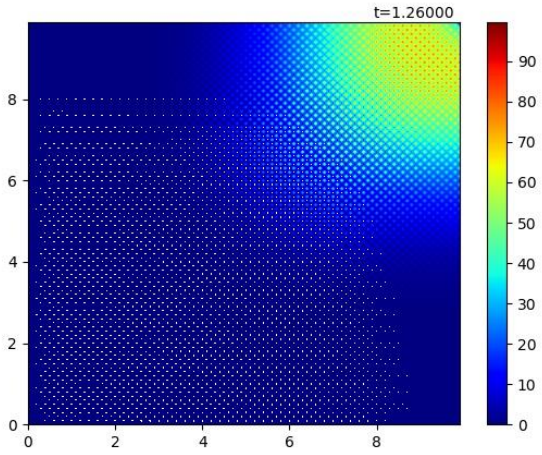


Figure 13 d=0.252 500 loops

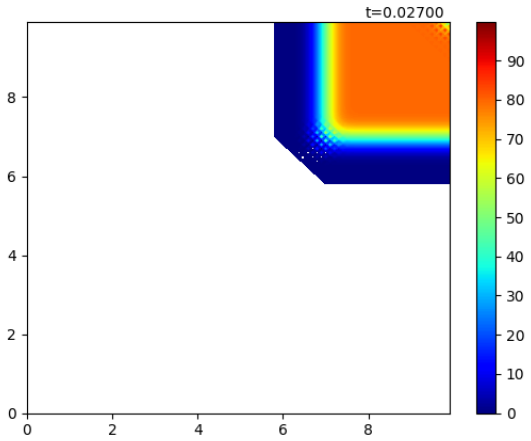


Figure 14 d = 0.3

According to Von Neumann Stability Analysis, for 2-dimensional data, it should be that d □0.25. It was verified by setting d=0.25 and passed it through 10000 loops which still sustained its stability; in the other hand, a small change to 0.252 with only 500 loops can destruct the plot by overflow of error. We tried d=0.3. Within 0.027 second the whole animation collapses because the error surges and exceed the heat range which displayed by white color.

D. Python Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  import os
4.  from matplotlib import animation
5.  dir = os.path.dirname(os.path.realpath(__file__))
6.  os.chdir(dir)
7.
8.  dx = 0.1
9.  dy = dx
10. alpha = 1.
11. grid_x = 100
12. grid_y = 100
13. nt = 100
14. d = 0.1
15. dt = d * (dx**2)/alpha
16.
17. def plot_activate(X,Y,n):
18.     global T
19.     plt.cla()
20.     plt.clf()
21.     plt.xlim(0.,np.max(x))
22.     plt.ylim(0.,np.max(x))
23.     cl = plt.contourf(X,Y,T,levels,cmap=cmap)
24.     plt.colorbar(cl)
25.     plt.text(np.max(x)*0.8,np.max(y)+dy,"t=%01.5f"%(dt*n))
26.
27. def init():
28.     T = np.zeros((grid_x,grid_y))
29.     T[0:20, :] = 80.
30.     T[grid_x-20:grid_x, :] = 80.
31.     T[:,0:20] = 80.
32.     T[:,grid_y-20:grid_y] = 80.
33.     return T
34.
```

```
x = np.linspace(0,dx * (grid_x - 1), grid_x)
y = np.linspace(0,dy * (grid_y - 1), grid_y)
X,Y = np.meshgrid(x,y)

cmap = plt.cm.get_cmap("jet")
cmap.set_over('grey')
g = 20
levels = np.arange(0.,100.,0.2)
count = 1
icount = 0
T = init()
def cyc(n, plot=True):
    global T, icount
    Tn = T.copy()
    T = Tn+d*(np.roll(Tn,1,axis=0)+np.roll(Tn,-1,axis=0)+np.roll(Tn,1,axis=1)+np.roll(Tn,-1,axis=1)-4*Tn)
    if plot:
        plot_activate(X,Y,n)
    icount += 1

def dir(n,plot=True):
    global T, icount
    T[0,:], T[:,0], T[grid_x-1,:], T[:,grid_y-1] = 0,0,0,0
    Tn = T.copy()
    T = Tn+d*(np.roll(Tn,1,axis=0)+np.roll(Tn,-1,axis=0)+np.roll(Tn,1,axis=1)+np.roll(Tn,-1,axis=1)-4*Tn)
    if plot:
        plot_activate(X,Y,n)
    icount += 1

#FIX WALL
def neu(n, g=0,plot=True):
    global T, icount
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
```

```
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    left[0,:],right[grid_x-1,:],up[:,grid_y-1],down[:,0] = 0,0,0,0
    T = Tn+d*(up+down+right+left-4*Tn)
    T[0,1:grid_y-1] += d * ((-2 * dx * g + Tn[1,1:grid_y-1]) > 0) * (-
2 * dx * g + Tn[1,1:grid_y-1])
    T[grid_x-1,1:grid_y-1] += d * ((-2 * dx * g+Tn[grid_x-2,1:grid_y-
1]) > 0) * (-2 * dx * g+Tn[grid_x-2,1:grid_y-1])
    T[1:grid_x-1,0] += d * ((-2 * dx * g+Tn[1:grid_x-1,1]) > 0) * (-
2 * dx * g+Tn[1:grid_x-1,1])
    T[1:grid_x-1,grid_y-1] += d * ((-2 * dx * g+Tn[1:grid_x-1,grid_y-
2]) > 0) * (-2 * dx * g+Tn[1:grid_x-1,grid_y-2])
    if plot:
        plot_activate(X,Y,n)
    icount += 1
fig = plt.figure()
a = animation.FuncAnimation(fig, neu, fargs=(g,),frames=200,interval=10)
plt.show()
```

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 on your own initial conditions and values for 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 the discretization as an algebraic equation
2. Investigate by modelling the differences when a (linear) is a constant and when a is u (non-linear) for a cyclic condition.
3. What happens when you set a Dirichlet boundary?
4. List potential applications for Burger's equation.

A. Discretization Algebraic Equation [Q.1]

By these equation

$$\frac{\partial U}{\partial t} = \frac{U_{x,y}^{n+1} - U_{x,y}^{n}}{\Delta t}$$

$$\frac{\partial U}{\partial x} = \frac{U_{x,y}^{n} - U_{x-1,y}^{n}}{\Delta x}$$

$$\frac{\partial^2 U}{\partial x^2} = \frac{U_{x+1,y}^{n} + U_{x-1,y}^{n} - 2U_{x,y}^{n}}{\Delta x^2}$$

We can derive the algebraic equation

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

If we let $d_1 = \frac{a\Delta t}{\Delta x}$ and $d_2 = \frac{v\Delta t}{\Delta x^2}$, we can rewrite our equation as

$$U_{x,y}^{n+1} = U_{x,y}^{n} - d_1\left(2U_{x,y}^{n} - U_{x-1,y}^{n} - U_{x,y-1}^{n}\right) + d_2\left(U_{x+1,y}^{n} + U_{x-1,y}^{n} + U_{x,y+1}^{n} + U_{x,y-1}^{n} - 4U_{x,y}^{n}\right)$$

B. Observation of Modelling and Condition [Q.2+3]

Initial Value

```
dx = 0.1
dy = dx
a = 1.
v = 2.
grid_x = 100
grid_y = 100
nt = 100
d1 = 0.1
d2 = 0.05
dt = d2 * (dx**2)/v

def init():
    T = np.zeros((grid_x,grid_y))
    T[70:90, 70:90] = 80.
    return T
```

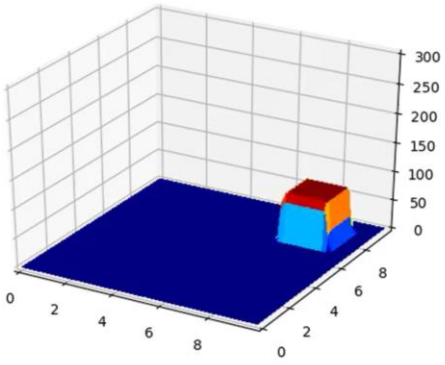


Figure 15 Burger's Equation Initial Condition

Plot Function

```
1.  fig = plt.figure()
2.  ax = fig.gca(projection='3d')
3.  cl = plt.contourf(X,Y,T,levels,cmap=cmap)
4.  plt.colorbar(cl)
5.
6.  def plot_activate(X,Y,n):
7.      global T
8.      plt.cla()
9.      plt.clf()
10.     plt.xlim(0.,np.max(x))
11.     plt.ylim(0.,np.max(x))
12.     plt.contourf(X,Y,T,levels,cmap=cmap)
13.     plt.text(np.max(x)*0.8,np.max(y)+dy,"t=%01.5f"%(dt*n))
14.
15. def plot3d_activate(X,Y,n):
16.     ax.cla()
17.     ax.set_xlim(0.,np.max(x))
18.     ax.set_ylim(0.,np.max(y))
19.     ax.set_zlim(0.,300)
20.     cl = ax.plot_surface(X,Y,T,linewidth=0,vmin=np.min(levels),vmax=np.max(lev
    els), cmap=cmap,antialiased=False)
```

Cyclic (Constant)

```
def cyc_con(n, plot_type=1):
    global T,d1,d2
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    T = Tn-d1*(2 * Tn - left - down) + d2 * (left+right+up+down-4*Tn)
    if plot_type == 1:
        plot_activate(X,Y,n)
    if plot_type == 2:
        plot3d_activate(X,Y,n)
```

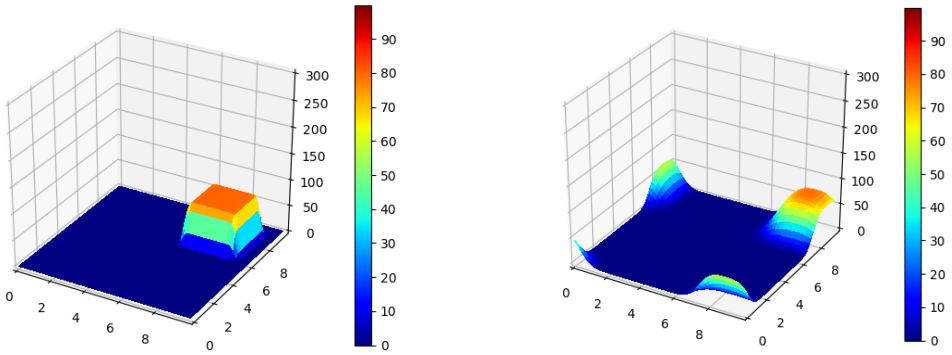


Figure 16 Cyclic Condition for Burger's Equation

Similar but different to the Diffusion equation, we included advection which cause the blob of heat to move along the $\hat{i}+\hat{j}$ vector (because the advection term $d_2\left(2U^n_{x,y}-U^n_{x-1,y}-U^n_{x,y-1}\right)$ allowed effect from x,y axis equally).

We tried to edit some part of the code following this equation

$$d_1\left(2U^n_{x,y} - 1.5U^n_{x-1,y} - 0.5U^n_{x,y-1}\right)$$

```
1.  def cyc_con(n, plot_type=1):
2.      global T,d1,d2
3.      Tn = T.copy()
4.      left = np.roll(Tn,1,axis=0)
5.      right = np.roll(Tn,-1,axis=0)
6.      up = np.roll(Tn,-1,axis=1)
7.      down = np.roll(Tn,1,axis=1)
8.      T = Tn-d1*(2 * Tn - 1.5*left - 0.5*down) + d2 * (left+right+up+down-4*Tn)
9.      if plot_type == 1:
10.         plot_activate(X,Y,n)
11.     if plot_type == 2:
12.         plot3d_activate(X,Y,n)
```

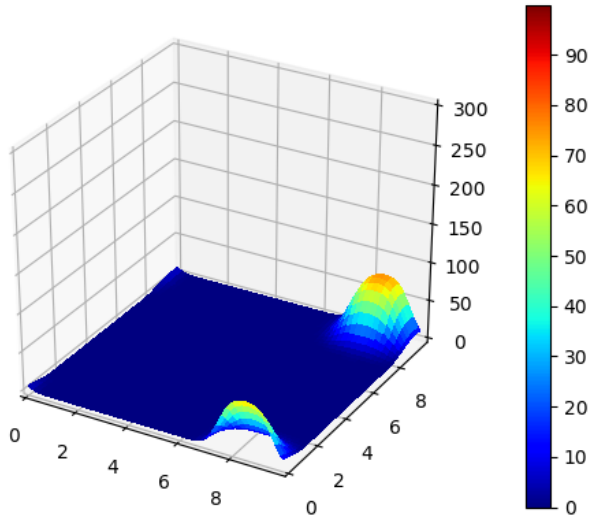


Figure 17 Cyclic Condition for Burger's Equation - 2

If look carefully, it can be seen that the blob movement shifted to the left compared to the previous result.

Cyclic (Non-linear – 1)

```
1.  def cyc_var(n, plot_type=1):
2.      global T,d1,d2
3.      Tn = T.copy()
4.      left = np.roll(Tn,1,axis=0)
5.      right = np.roll(Tn,-1,axis=0)
6.      up = np.roll(Tn,-1,axis=1)
7.      down = np.roll(Tn,1,axis=1)
8.      T = Tn-dt*Tn / dx*(2 * Tn - left - down) + d2 * (left+right+up+down-
    4*Tn)
9.      if plot_type == 1:
10.         plot_activate(X,Y,n)
11.     if plot_type == 2:
12.         plot3d_activate(X,Y,n)
```

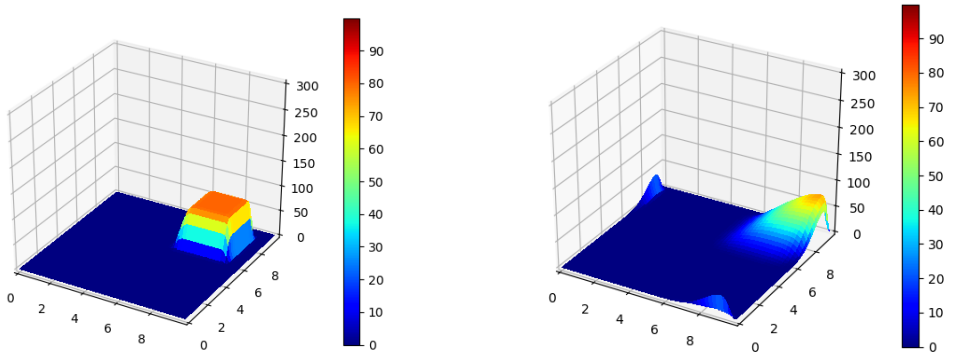


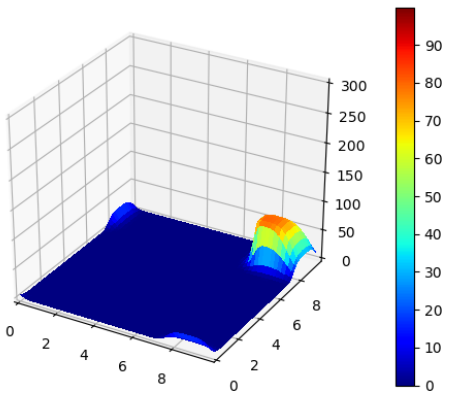
Figure 18 Cyclic Condition for Burger's Equation - Non-linear

According to the figure, the tip of the heat blob moves faster. This can be assumed by the non-linear factor. By the point where has higher heat, the effect from advection will be stronger be because we made it multiply itself again from the formula. ($\odot$ is elements wise multiplication)

$$\frac{\Delta t}{\Delta x} U^n_{x,y} \odot \left(2U^n_{x,y} - U^n_{x-1,y} - U^n_{x,y-1}\right)$$

Cyclic (Non-linear – 2)

```
1.  def cyc_var(n, plot_type=1):
2.      global T,d1,d2
3.      Tn = T.copy()
4.      left = np.roll(Tn,1,axis=0)
5.      right = np.roll(Tn,-1,axis=0)
6.      up = np.roll(Tn,-1,axis=1)
7.      down = np.roll(Tn,1,axis=1)
8.      T = Tn-
    dt*(np.sqrt(80**2 - Tn ** 2))/dx*(2 * Tn - left - down) + d2 * (left+right+up+
    down-4*Tn)
9.      if plot_type == 1:
10.         plot_activate(X,Y,n)
11.     if plot_type == 2:
12.         plot3d_activate(X,Y,n)
```



In this case, we made the point with more heat to move slower by multiplied with a value of a decreasing conversely to $U_{x,y}^{n}$ which we chose $\sqrt{80^2 - U_{x,y}^{n\ 2}}$ (80 was maximum of the heat range). This kind of non-linear advection is similar to the "congestion" model which will be mentioned in the application section.

Figure 19 Cyclic Condition for Burger's Equation - Non-linear 2

$$\frac{\Delta t}{\Delta x}\sqrt{80^2 - U_{x,y}^{n\ 2}} \odot \left(2U_{x,y}^{n} - U_{x-1,y}^{n} - U_{x,y-1}^{n}\right)$$

Dirichlet (Non-linear)

```
def der_var(n, plot_type=1):
    global T,d1,d2
    T[0,:],T[:,0], T[grid_x-1,:], T[:, grid_y-1] = 0,0,0,0
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    T = Tn-dt *Tn /dx*(2 * Tn - left - down) + d2 * (left+right+up+down-
4*Tn)
    if plot_type == 1:
        plot_activate(X,Y,n)
    if plot_type == 2:
        plot3d_activate(X,Y,n)
```

No heat leaks around the boundaries and the heat gradually lost from the system even faster than just diffusion equation. (the heat throw itself into the “black hole” wall). Overall, the movement and diffusion did not change much from the cyclic condition.

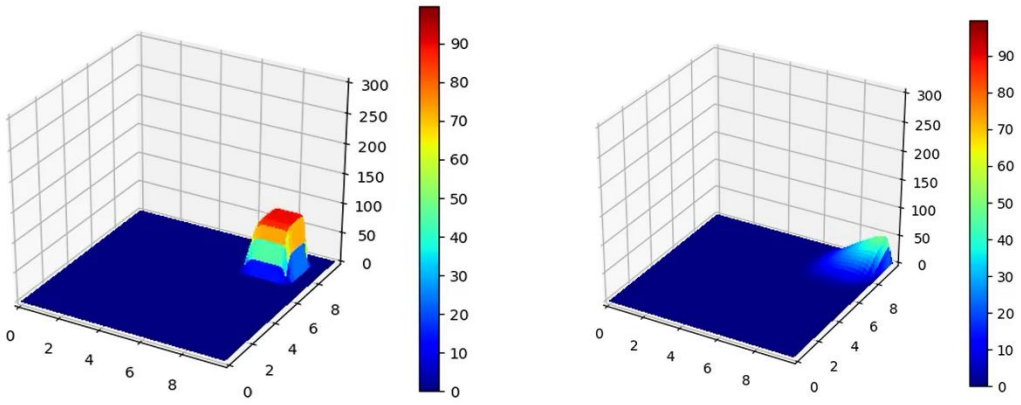


Figure 20 Dirichlet's Condition for Burger's Equation

Neumann (Non-linear, G=5)

```
def neu(n, g=0, plot_type=1):
    global T,d1,d2
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    ghost1 = -2 * dx * g + Tn[1,1:grid_y-1]  #create Ghost wall
    ghost2 = -2 * dx * g + Tn[grid_x-2,1:grid_y-1]
    ghost3 = -2 * dx * g + Tn[grid_x-2,1:grid_y-1]
    ghost4 = -2 * dx * g + Tn[1:grid_x-1,grid_y-2]
    ghost1 = ghost1 > 0 * ghost1  #prevent negative value
    ghost2 = ghost2 > 0 * ghost2
    ghost3 = ghost3 > 0 * ghost3
    ghost4 = ghost4 > 0 * ghost4
    left[0,:],right[grid_x-1,:],up[:,grid_y-1],down[:,0] = 0,0,0,0
    T = Tn-dt* Tn/dx*(2 * Tn - left - down) + d2 * (left+right+up+down-4*Tn)
    T[0,1:grid_y-1] += d2 * ghost1 + dt * T[0,1:grid_y-1]  / dx*ghost1
    T[grid_x-1,1:grid_y-1] += d2 * ghost2
    T[1:grid_x-1,0] += d2 * ghost3 + dt * T[1:grid_x-1,0]  / dx * ghost3
    T[1:grid_x-1,grid_y-1] += d2 * ghost4
    if plot_type == 1:
        plot_activate(X,Y,n)
    if plot_type == 2:
        plot3d_activate(X,Y,n)
```

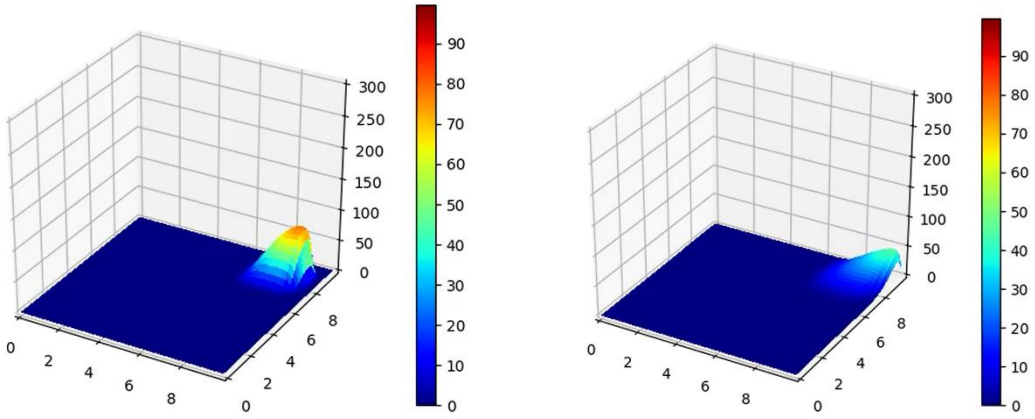


Figure 21 Neumann's Condition for Burger's Equation

Extended from the Dirichlet condition, the wall's heat was affected by the blob of heat that was approaching the frame. In this case G=5 so we can see some gradients or slopes near the wall as same as the diffusion equation case.

```
1.      T[0,1:grid_y-1] += d2 * ghost1 + dt * T[0,1:grid_y-1]  / dx*ghost1
```

$$-\frac{\Delta t}{\Delta x}U^n_{0,y}\odot\left(2U^n_{0,y}-U^n_{-1,y}-U^n_{0,y-1}\right)\overset{Ghost}{\Rightarrow}\frac{\Delta t}{\Delta x}U^n_{0,y}\odot\left(U^n_{-1,y}\right)$$

$$-\frac{\Delta t}{\Delta x}U^n_{x,0}\odot\left(2U^n_{x,0}-U^n_{x-1,0}-U^n_{x,-1}\right)\overset{Ghost}{\Rightarrow}\frac{\Delta t}{\Delta x}U^n_{x,0}\odot\left(U^n_{x,-1}\right)$$

According to the equation above, we need to add the "Ghost" term for those values that did not exist. Because we implemented space backward for the advection, there were only 2 terms we needed to add by the ghost wall.

C. Application of Burger's Equation [Q.4]

1. Used in Fluid Mechanic

   We describe the motion of fluid with viscosity by Navier–Stokes equations which is the equation cannot be solved for analytical solution right now. The general formula is too long so we will refer only the case of incompressible fluid Navier-Stokes in convection

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu\nabla^2 u = -\nabla w + g$$

   If we consider in the simpler case such as homogenous differential equation (no internal and external source), it will become non-linear burger equation.

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u = \nu\nabla^2 u$$

2. Toy model

   Toy model is given a mean as a tool used to understand some behavior of the system but in simpler way rather than a generally complex model. One interesting model is traffic flow.

   Given that ρ is the density of the cars (ρ_{max} has meaning as bumper hit the front car!).

   We can see the term $[(1-\rho)\rho]_x$ which implied that more car reduces the advection speed.

$$\rho_t + [(1-\rho)\rho]_x = \epsilon\rho_{xx}; \epsilon = \frac{D}{v_{max}x_0}$$

3. Nuclear fusion reactor

   In the nuclear fusion reactor, there is a part called "Lithium blanket" where playing role as a cooler for the reactor. Lithium blanket is contained of liquid lithium and controlled by magnetic force. Consequently, there is need to study how to control the flow of this blanket. The model is as similar to Navier-Stokes Burger's equation with external source.

$$u_t + uu_x = vu_{xx} + B_0$$

   Where B_0 is applied magnetic field. For example, $B_0 = e^{-t}\cos\left(\frac{\pi x}{2}\right)$.

Reference:

Application of Generalize Burger's Equation. (n.d.). Retrieved from Shodhganga: http://shodhganga.inflibnet.ac.in/bitstream/10603/37622/11/11_chapter%204.pdf

Landajuela, M. (2011). Burgers equation. *bcam*, 2-3.

Viscous Burgers equation physical meaning. (2014, 7 23). Retrieved from Physics Stackexchange: https://physics.stackexchange.com/questions/127771/viscous-burgers-equation-physical-meaning

D. Python Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from mpl_toolkits.mplot3d import Axes3D
4.  import os
5.  from matplotlib import animation
6.  dir = os.path.dirname(os.path.realpath(__file__))
7.  os.chdir(dir)
8.
9.  dx = 0.1
10. dy = dx
11. a = 2.
12. v = 2.
13. grid_x = 100
14. grid_y = 100
15. nt = 100
16. d1 = 0.1
17. d2 = 0.05
18. dt = d2 * (dx**2)/v
19.
20. def init():
21.     T = np.zeros((grid_x,grid_y))
22.     T[70:90, 70:90] = 80.
23.     return T
24.
25. x = np.linspace(0,dx * (grid_x - 1), grid_x)
26. y = np.linspace(0,dy * (grid_y - 1), grid_y)
27. X,Y = np.meshgrid(x,y)
28. cmap = plt.cm.get_cmap("jet")
29. cmap.set_over('grey')
30. g = 5
31. levels = np.arange(0.,100.,0.2)
32. count = 1
33. icount = 0
34. T = init()
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')
cl = plt.contourf(X,Y,T,levels,cmap=cmap)
plt.colorbar(cl)

def plot_activate(X,Y,n):
    global T
    plt.cla()
    plt.clf()
    plt.xlim(0.,np.max(x))
    plt.ylim(0.,np.max(x))
    plt.contourf(X,Y,T,levels,cmap=cmap)
    plt.text(np.max(x)*0.8,np.max(y)+dy,"t=%01.5f"%(dt*n))

def plot3d_activate(X,Y,n):
    ax.cla()
    ax.set_xlim(0.,np.max(x))
    ax.set_ylim(0.,np.max(y))
    ax.set_zlim(0.,300)
    cl = ax.plot_surface(X,Y,T,linewidth=0,vmin=np.min(levels),vmax=np.max(levels), cmap=cmap,antialiased=False)

def cyc_var(n, plot_type=1):
    global T,d1,d2
    Tn = T.copy()
    left = np.roll(Tn,1,axis=0)
    right = np.roll(Tn,-1,axis=0)
    up = np.roll(Tn,-1,axis=1)
    down = np.roll(Tn,1,axis=1)
    T = Tn-dt*(np.sqrt(80**2-Tn ** 2))/dx*(2 * Tn - left - down) + d2 * (left+right+up+down-4*Tn)
    if plot_type == 1:
        plot_activate(X,Y,n)
    if plot_type == 2:
        plot3d_activate(X,Y,n)

```

```
69. def cyc_con(n, plot_type=1):
70.     global T,d1,d2
71.     Tn = T.copy()
72.     left = np.roll(Tn,1,axis=0)
73.     right = np.roll(Tn,-1,axis=0)
74.     up = np.roll(Tn,-1,axis=1)
75.     down = np.roll(Tn,1,axis=1)
76.     T = Tn-d1*(2 * Tn - 1.5*left - 0.5*down) + d2 * (left+right+up+down-
    4*Tn)
77.     if plot_type == 1:
78.         plot_activate(X,Y,n)
79.     if plot_type == 2:
80.         plot3d_activate(X,Y,n)
81.
82. def der_var(n, plot_type=1):
83.     global T,d1,d2
84.     T[0,:],T[:,0], T[grid_x-1,:], T[:, grid_y-1] = 0,0,0,0
85.     Tn = T.copy()
86.     left = np.roll(Tn,1,axis=0)
87.     right = np.roll(Tn,-1,axis=0)
88.     up = np.roll(Tn,-1,axis=1)
89.     down = np.roll(Tn,1,axis=1)
90.     T = Tn-
    dt*(np.sqrt(Tn ** 2))/dx*(2 * Tn - left - down) + d2 * (left+right+up+down-
    4*Tn)
91.     if plot_type == 1:
92.         plot_activate(X,Y,n)
93.     if plot_type == 2:
94.         plot3d_activate(X,Y,n)
95.
96. def neu(n, g=0, plot_type=1):
97.     global T,d1,d2
98.     Tn = T.copy()
99.     left = np.roll(Tn,1,axis=0)
100.     right = np.roll(Tn,-1,axis=0)
101.     up = np.roll(Tn,-1,axis=1)
```

```
    down = np.roll(Tn,1,axis=1)
    ghost1 = -2 * dx * g + Tn[1,1:grid_y-1]
    ghost2 = -2 * dx * g+Tn[grid_x-2,1:grid_y-1]
    ghost3 = -2 * dx * g+Tn[grid_x-2,1:grid_y-1]
    ghost4 = -2 * dx * g+Tn[1:grid_x-1,grid_y-2]
    ghost1 = ghost1 > 0 * ghost1
    ghost2 = ghost2 > 0 * ghost2
    ghost3 = ghost3 > 0 * ghost3
    ghost4 = ghost4 > 0 * ghost4
    left[0,:],right[grid_x-1,:],up[:,grid_y-1],down[:,0] = 0,0,0,0
    T = Tn-
dt*(np.sqrt(Tn ** 2))/dx*(2 * Tn - left - down) + d2 * (left+right+up+down-
4*Tn)
    T[0,1:grid_y-1] += d2 * ghost1 + dt * T[0,1:grid_y-1]/dx*ghost1
    T[grid_x-1,1:grid_y-1] += d2 * ghost2
    T[1:grid_x-1,0] += d2 * ghost3 + dt * T[1:grid_x-1,0]/dx*ghost3
    T[1:grid_x-1,grid_y-1] += d2 * ghost4
    if plot_type == 1:
        plot_activate(X,Y,n)
    if plot_type == 2:
        plot3d_activate(X,Y,n)
a = animation.FuncAnimation(fig, der_var,fargs=(2), frames=200,interval=10)
a.save('3d-der-var.mp4',fps=30,extra_args=['-vcodec','libx264'])
```

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,

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

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, (2) Leith's Method, (3) CIP Method, and (4) analytical solution.

1. Construct f plots along x for t=100, 300, 500, 700. Compare the results of each method and discuss the errors accompanied by each method.

Note: Make sure the code is constructed neatly. Place comments using "#" character.

A. Initial Condition

Plot Function

```
def plot_line(n, folder): #Usual Plot Function
    global model,f
    plt.clf()
    plt.cla()
    plt.ylim(-.5,1.5)
    plt.plot(model,f)
    plt.text(80.,1.51,'t=%05.2f'%(n)) #display time step, not actual time
    plt.text(10.,1.51,folder) #type of analysis
    plt.savefig('%s/timestep_%04i.jpg'%(folder, n))
```

Initial Value

```
f = np.zeros_like(model)
g = np.zeros_like(model)
f[40:60] = 1.
g[40], g[60] = 1./dx, -1./dx  #(1-0)/dx
dx = 1.
C = 0.9
u = 1.
dt = np.abs(C * dx/u) #prevent negative
```

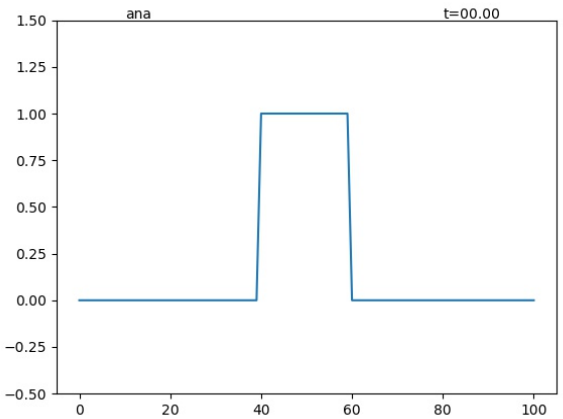


Figure 22 Initial Condition for Advection Equation

B. Functions

Upwind scheme

```
def upwind(n, toplot=True):
    global f
    fn = f.copy()
    usign = int(np.sign(u))
    f = fn + C * (np.roll(fn,usign,0)-fn)
    if toplot:
        plot_line(n, 'upwind') #save file in folder 'upwind'

```

Leith's or Lax-Wendroff method

```
def lex(n, toplot=True):
    global f
    fn = f.copy()
    c = fn
    b = 1/(2*dx) * (np.roll(fn,-1,0) - np.roll(fn,1,0))
    a = 1/(2*dx**2) * (np.roll(fn,-1,0) - 2*fn + np.roll(fn,1,0))
    f = a * (u*dt)**2 - b*(u*dt) + c
    if toplot:
        plot_line(n, 'lex')

```

CIP Method

```
def cip(n, toplot=True):
    global f,g
    fn = f.copy()
    gn = g.copy()
    usign = np.int(np.sign(u))
    x_iiup = (-usign*dx) #x_iup - x_i sign deped on the sign of stream
    a = -2*(np.roll(fn,usign,0)-fn)/x_iiup**3
+ (gn + np.roll(gn,usign,0))/x_iiup**2
    b = -3*(-np.roll(fn,usign,0)+fn)/x_iiup**2
- (2*gn + np.roll(gn,usign,0))/x_iiup
    eps = -u*dt
    f = a * eps**3 + b*eps**2+gn*eps+fn
    g = (3*a*eps**2+2*b*eps+gn)
    if toplot:
        plot_line(n, 'cip')

```

Analytical Solution

```
fana = f.copy()  #copy the initial value to a constant
def ana(n, toplot=True):
    global f
    f = np.roll(fana,np.int(np.floor(u*n*dt/dx)%fana.shape[0])) #move function
 according to the stream description below
    if toplot:
        plot_line(n, 'ana')
```

Unfortunately that x is discontinuous which $u\Delta t$ may not be able to fit exactly by Δx so we estimated the amount of block Δx to shift at any time step n by using floor function $N = \left\lfloor \frac{un\Delta t}{\Delta x} \right\rfloor = \lfloor nC \rfloor$. To prevent from too large number (in case of long run), we roll the array by the value a satisfying

$$N \equiv a (\mathrm{mod}\, b); b = L / dt, 0 \le a < b$$

C. Time Step

(500, 700 descriptions will be omitted and discussed once in Discussion part)

(t in the figure indicates time step **not** actual time)

t = 100

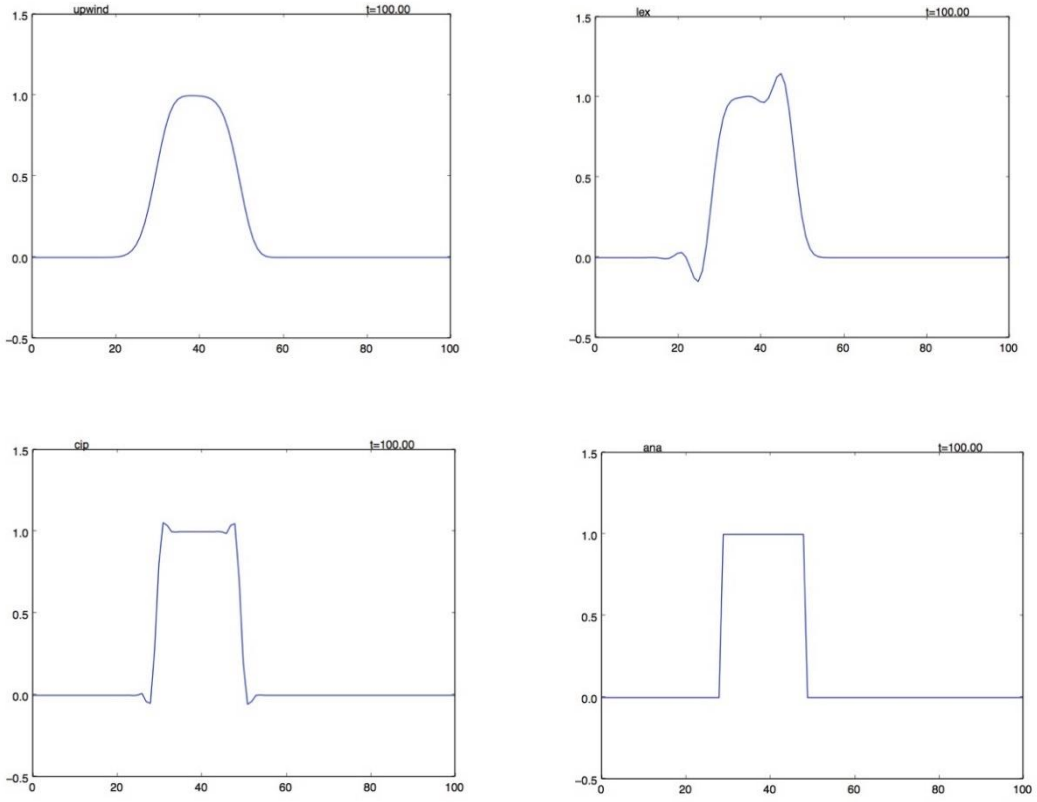


Figure 23 t=100 State

Upwind scheme:

The plot lost its shape and maximum altitude.

Leith's method:

Some shooting appears at the border.

CIP method:

The shape is conserved with a small overshooting around the corner.

t = 300

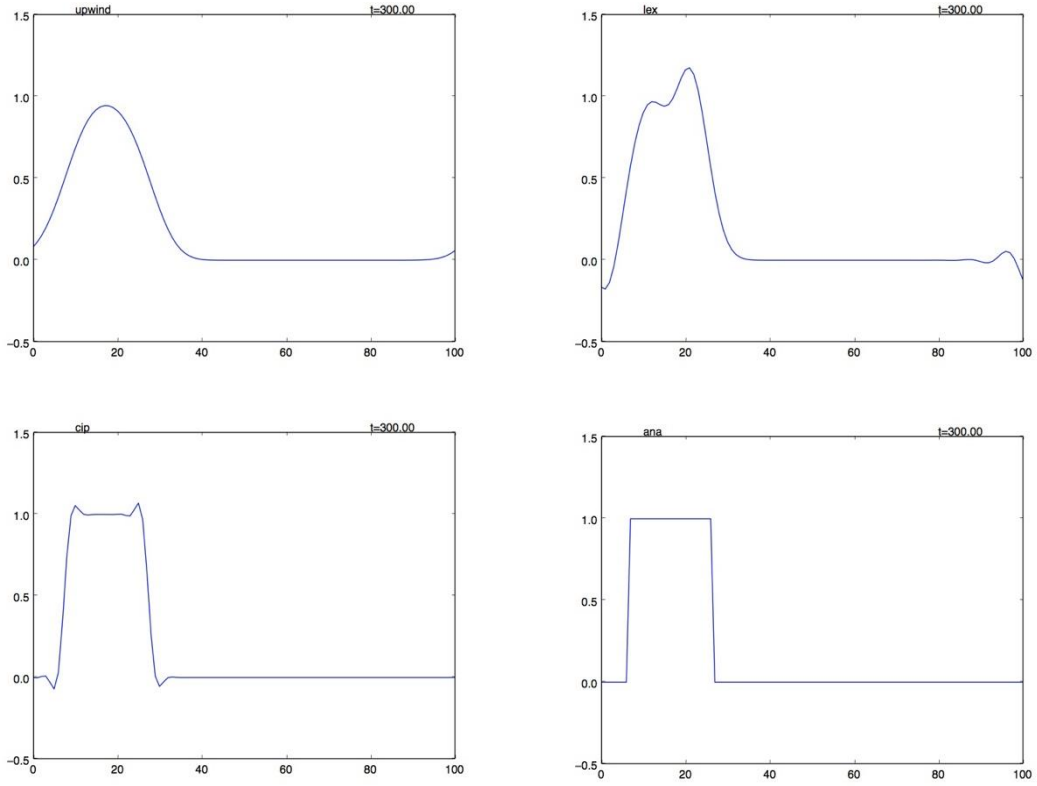


Figure 24 t=300 State

Upwind scheme:

kept losing its shape, maximum altitude and clearly became flat.

Leith's method:

more wavy shootings appear at the corners though the altitude is not significantly changed.

CIP method:

No significant change to the shape, considerably stable.

t = 500

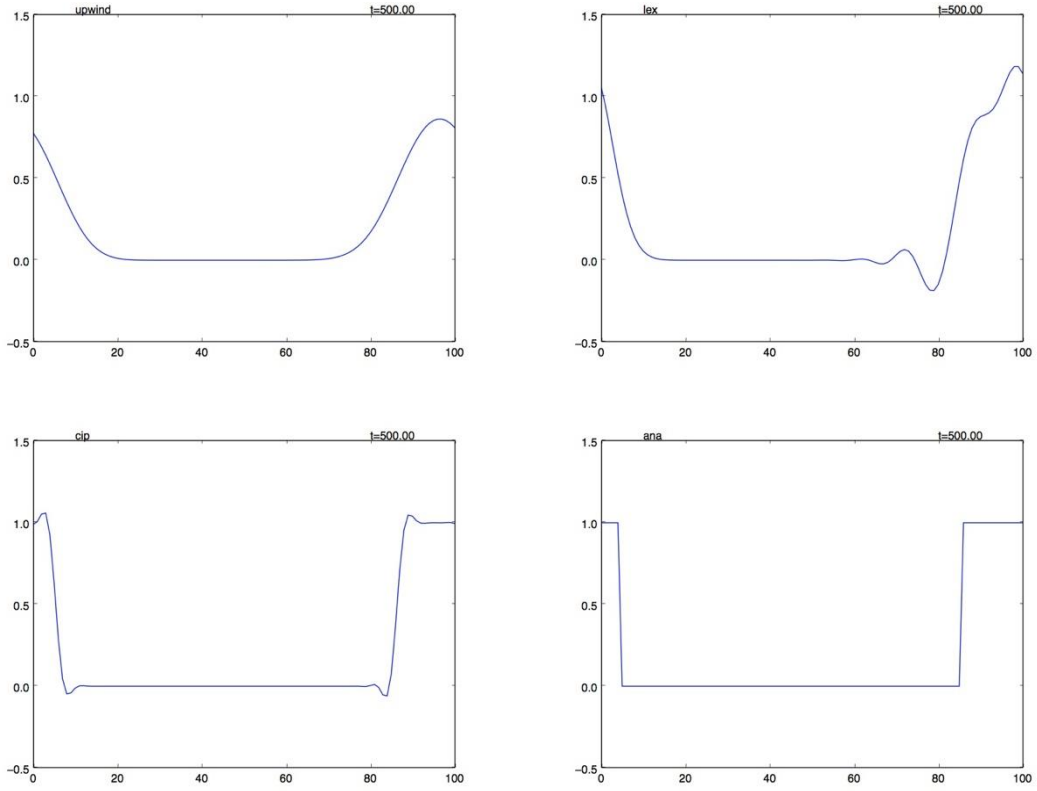


Figure 25 t=500 State

t = 700

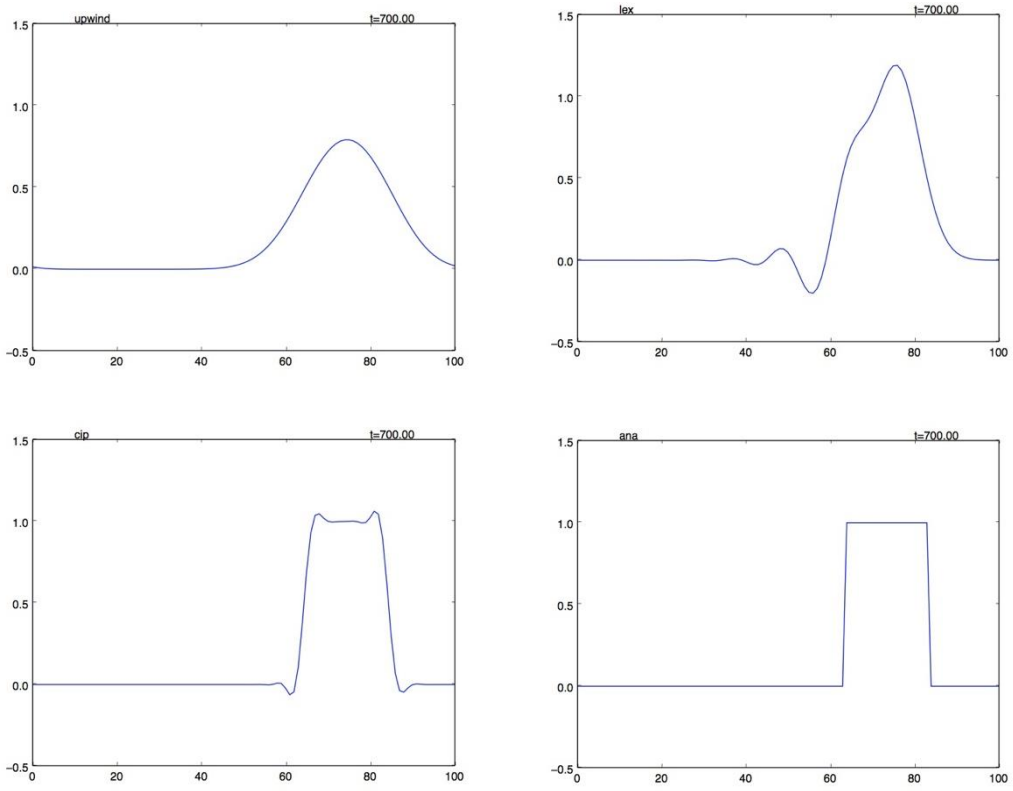


Figure 26 t=700 State

Discussion

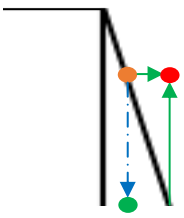


Figure 27 upwind error

As the left figure, the upstream scheme dissipating started from the corners. Because of the linear interpolation, error appeared at the position with non-linear step. According to the figure, the red point's value should be as same as the green point but it was estimated as the orange point. Analogously, the dissipation is similar to the landslide phenomenon.

For the Lax-Wendroff Method, using 2^{nd} order polynomial, did not appear any problem about dissipating but several wavy shootings occurred. This can be assumed from the property of Parabolic itself. Due to the existence of 2^{nd} degree, the interpolation can be both convex or concave which make it not dissipated (overall estimation may work out better).

Furthermore, from the upwind scheme, assuming backward space and forward time.

$$\frac{f_i^{n+1} - f_i^n}{\Delta t} + u\frac{f_i^n - f_{i-1}^n}{\Delta x} = 0$$

And Taylor's expansion at point (x_i, t^n)

$$f_i^{n+1} = f_i^n + \Delta t\left(\frac{\partial f}{\partial t}\right)_i^n + \frac{\Delta t^2}{2}\left(\frac{\partial^2 f}{\partial t^2}\right)_i^n + \dots$$

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

Substitute in to the upwind scheme equation

$$\left(\frac{\partial f}{\partial t}\right)_i^n + u\left(\frac{\partial f}{\partial x}\right)_i^n = \underbrace{-\frac{\Delta t}{2}\left(\frac{\partial^2 f}{\partial t^2}\right)_i^n - \frac{\Delta t^2}{6}\left(\frac{\partial^3 f}{\partial t^3}\right)_i^n + \frac{u\Delta x}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i^n - \frac{u\Delta x^2}{6}\left(\frac{\partial^3 f}{\partial x^3}\right)_i^n + \dots}_{Truncation\ error}$$

Applying several algebra and calculus operation (detail in reference), eventually, we reached the final form

$$\frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} = \underbrace{\frac{u\Delta x}{2}(1-C)\frac{\partial^2 f}{\partial x^2}}_{diffusion\ term} + \underbrace{\frac{u\Delta x^2}{6}(3C - 2C^2 - 1)\frac{\partial^3 f}{\partial x^3}}_{dispersion\ term} + \dots$$

Where $C = u\frac{dt}{dx}$

Which we can see that the dissipation in the upwind scheme caused by "diffusion term" from the truncation error. If $u \to 1$, the diffusion term will lose its effect and the plot would dissipate slower.

For the 3rd order term, it is considered as "dispersion term". For example, Korteweg–de Vries equation is a non-linear dispersion equation similar to 1-D advection equation but with 3rd order differential term. This effect can be seen in Lax-Wendroff Method which its truncation error leaded with an odd order term.

As usual, which error the scheme will encounter depended on the leading term which has the largest influence. We can determine by the leading term of the truncation error, even for dissipation and odd for dispersion.

The truncation error of CIP method also leaded by even order as same as upwind scheme but suffer less dissipation because of third order estimation. However, eventually it should lose its amplitude.

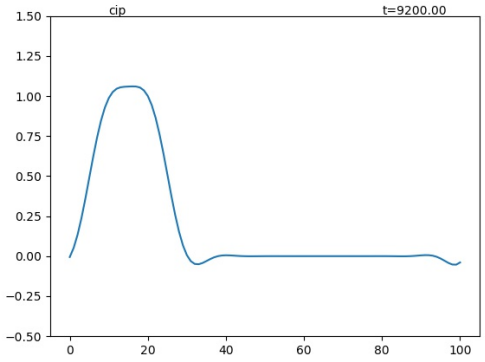


Figure 28 CIP dissipation

Reference:

http://www.mathematik.uni-dortmund.de/~kuzmin/cfdintro/lecture10.pdf

http://twister.caps.ou.edu/CFD2007/Chapter3_3.pdf

Korteweg–de Vries equation. (2017, October 23). Retrieved November 24, 2017, from https://en.wikipedia.org/wiki/Korteweg%E2%80%93de_Vries_equation

D. Python Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from mpl_toolkits.mplot3d import Axes3D
4.  import os
5.  from matplotlib import animation
6.  dir = os.path.dirname(os.path.realpath(__file__))
7.  os.chdir(dir)
8.  dx = 1.
9.  model = np.arange(0,100 + dx, dx)
10. f = np.zeros_like(model)
11. g = np.zeros_like(model)
12. f[40:60] = 1.
13. g[40], g[60] = 1./dx, -1./dx
14. C = 0.6
15. u = 1.
16. dt = np.abs(C * dx/u)
17.
18. def plot_line(n, folder): #Usual Plot Function
19.     global model,f
20.     plt.clf()
21.     plt.cla()
22.     plt.ylim(-.5,1.5)
23.     plt.plot(model,f)
24.     plt.text(80.,1.51,'t=%05.2f'%(n))
25.     plt.text(10.,1.51,folder) #type of analysis
26.     plt.savefig('%s/timestep_%04i.jpg'%(folder, n)) #save in folder[type] time
    step[n]
27.
28. def upwind(n, toplot=True):
29.     global f
30.     fn = f.copy()
31.     usign = int(np.sign(u))
32.     f = fn + C * (np.roll(fn,usign,0)-fn)
33.     if toplot:
```

```
        plot_line(n, 'upwind') #save file in folder 'upwind'

def lex(n, toplot=True):
    global f
    fn = f.copy()
    c = fn
    b = 1/(2*dx) * (np.roll(fn,-1,0) - np.roll(fn,1,0))
    a = 1/(2*dx**2) * (np.roll(fn,-1,0) - 2*fn + np.roll(fn,1,0))
    f = a * (u*dt)**2 - b*(u*dt) + c
    if toplot:
        plot_line(n, 'lex')

def cip(n, toplot=True):
    global f,g
    fn = f.copy()
    gn = g.copy()
    usign = np.int(np.sign(u))
    x_iiup = (-usign*dx) #x_iup - x_i sign deped on the sgn of stream
    a = -2*(np.roll(fn,usign,0)-
fn)/x_iiup**3 + (gn + np.roll(gn,usign,0))/x_iiup**2
    b = -3*(-
np.roll(fn,usign,0)+fn)/x_iiup**2 - (2*gn + np.roll(gn,usign,0))/x_iiup
    eps = -u*dt
    f = a * eps**3 + b*eps**2+gn*eps+fn
    g = (3*a*eps**2+2*b*eps+gn)
    if toplot:
        plot_line(n, 'cip')


fana = f.copy()
def ana(n, toplot=True):
    global f
    usign = np.int(np.sign(u))
    f = np.roll(fana,np.int(np.floor(usign*n*dt/dx)%fana.shape[0])) #move func
tion according to the stream
    if toplot:
```

```
        plot_line(n, 'ana')

# fig = plt.figure()
# a = animation.FuncAnimation(fig, upwind,frames=200,interval=10)
# plt.show()
for i in range(10000):
    toplot = False
    if i in [100,300,500,700,9200]:
        toplot = True
    cip(i, toplot)
```