PARTIAL DIFFERENTIAL EQUATIONS FOR SCIENCE AND ENGINEERING

Final Report

NOVEMBER 27, 2017

Thumwanit Napat

16B00133

Contents

Problem 1 Diffusion Equation 2
A. Effect of Initial Condition (By Neumann) [Q.1] 3
B. Observation of Conditions [Q.2+4] 5
- Desired Initial 5
- Cyclic 6
- Dirichlet 8
- Neumann (G=0) 10
Neumann (G=20) 12
C. Effect of d's value [Q.3] 13
D. Python Code 15
Problem 2 Burger's Equation 18
A. Discretization Algebraic Equation [Q.1] 19
B. Observation of Modelling and Condition [Q.2+3] 20
Initial Value 20
Plot Function 21
Cyclic (Constant) 22
Cyclic (Non-linear – 1) 24
Cyclic (Non-linear – 2) 25
Dirichlet (Non-linear) 26
Neumann (Non-linear, G=5) 27
C. Application of Burger's Equation [Q.4] 29
D. Python Code 30

Table of Figures

Figure 1 Initial condition set to 100 and only to regions aa 14

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]

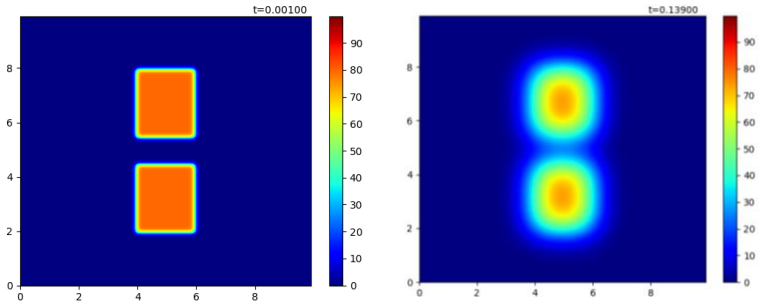
Initial Condition

```
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()
```

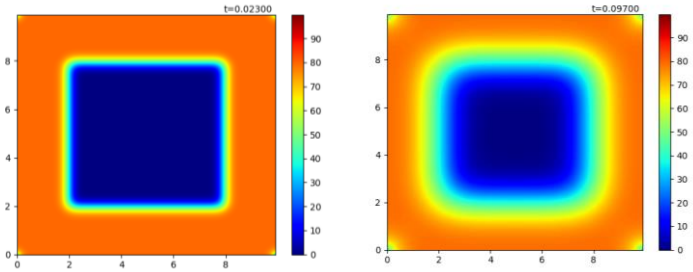
- couple blobs

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



- 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

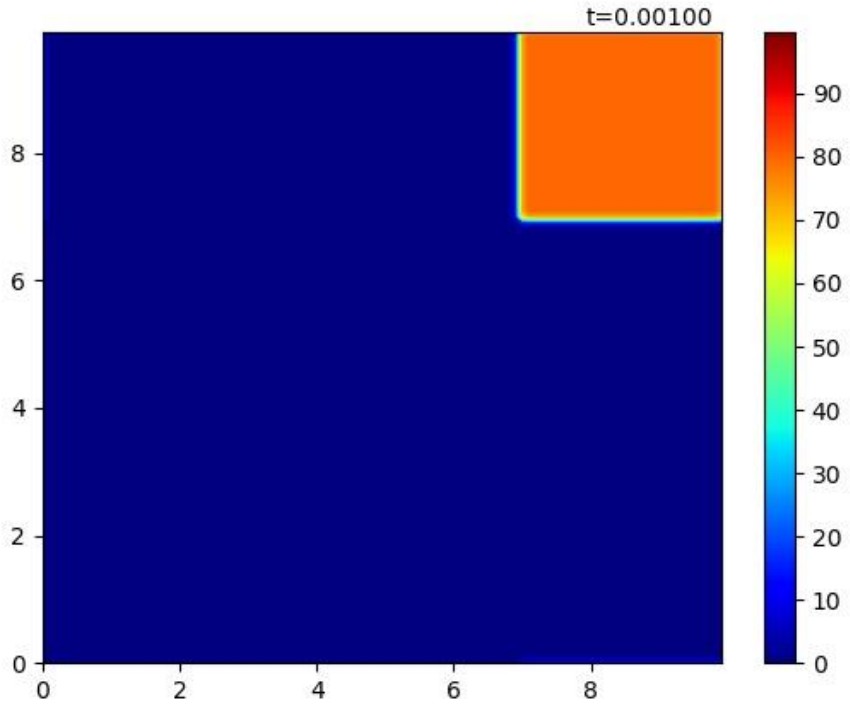


Figure 1 initial condition for observation

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):
    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
```

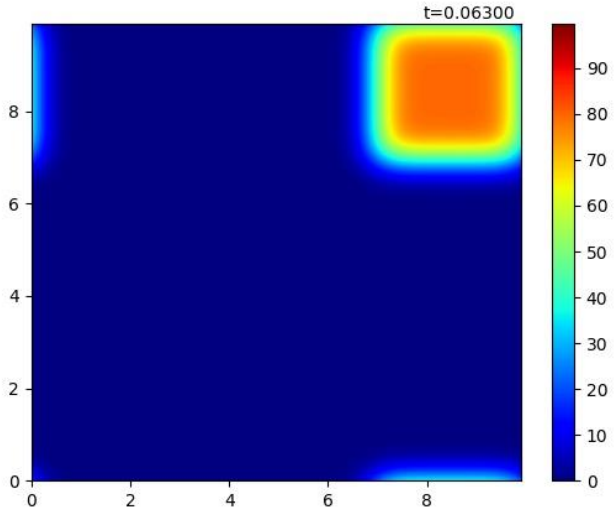


Figure 2 middle state of cyclic condition

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

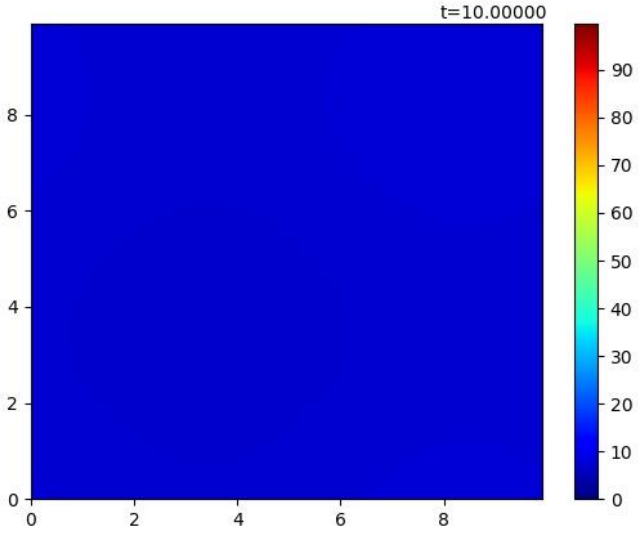


Figure 3 steady state of cyclic condition

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

```
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
```

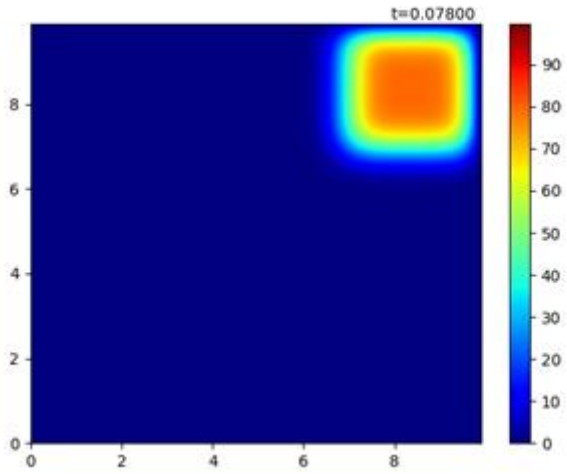


Figure 4 middle state of Dirichlet condition

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.

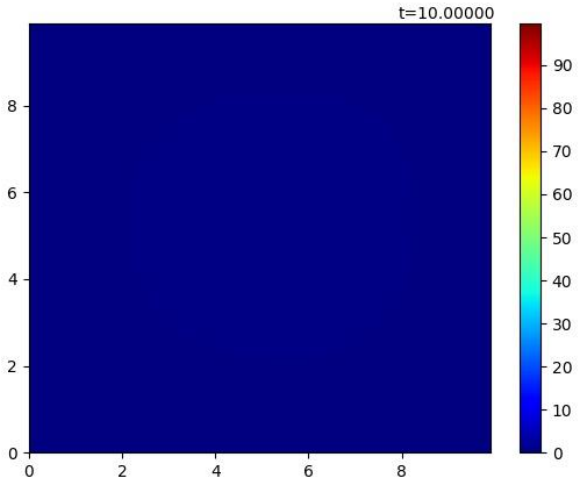


Figure 5 final state of Dirichlet condition

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
    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
```

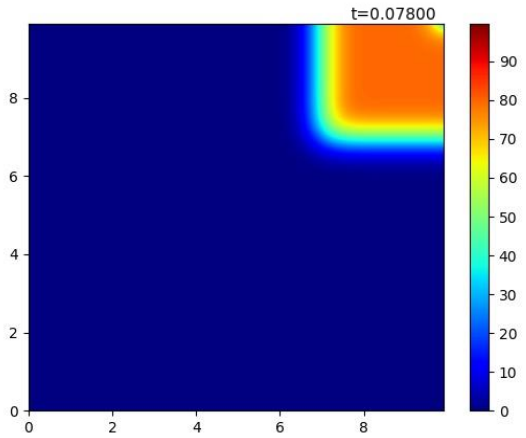


Figure 6 middle state of Neumann condition

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]
```

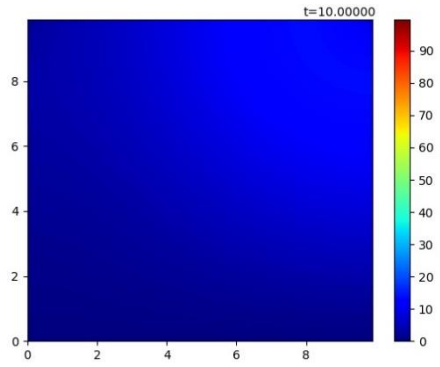


Figure 7 almost final state of Neumann condition

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.

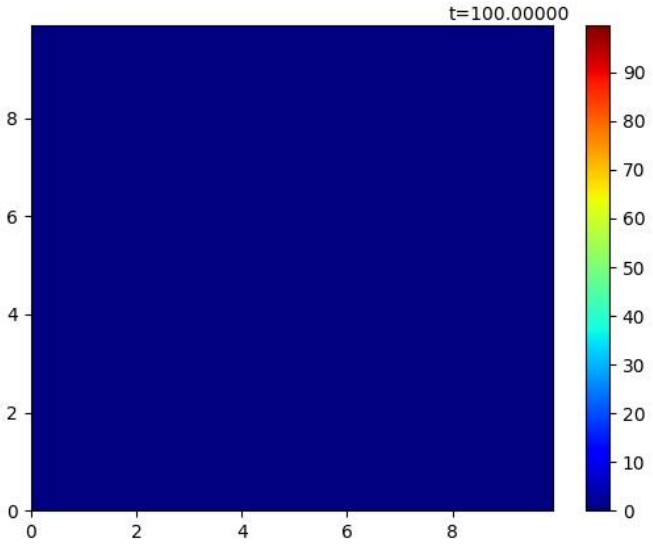


Figure 8 final state of Neumann condition

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

Neumann (G=20)

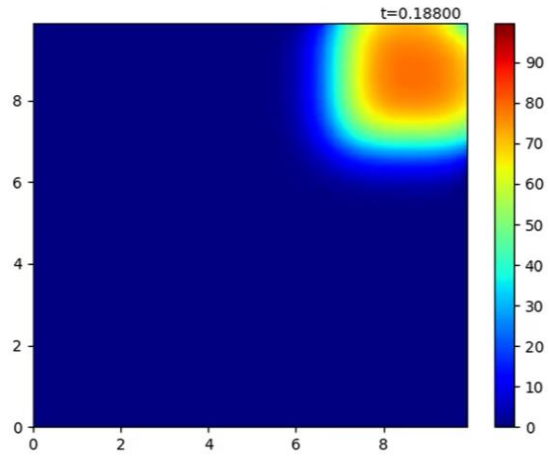


Figure 9 middle state of Neumann condition (G≠ 0)

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

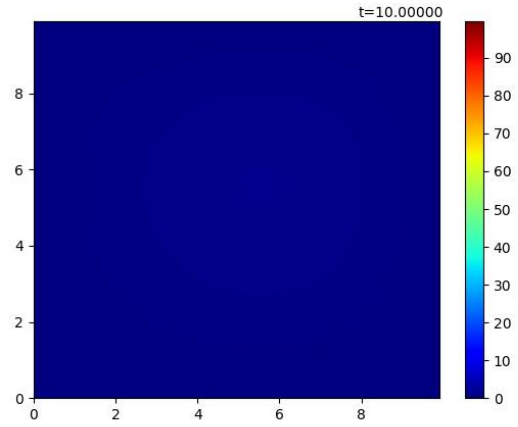


Figure 10 final state of Neumann condition (G≠ 0)

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

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

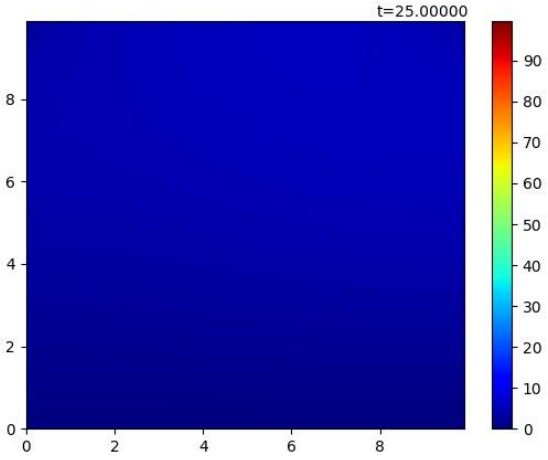


Figure 11 d=0.25 10000 loops

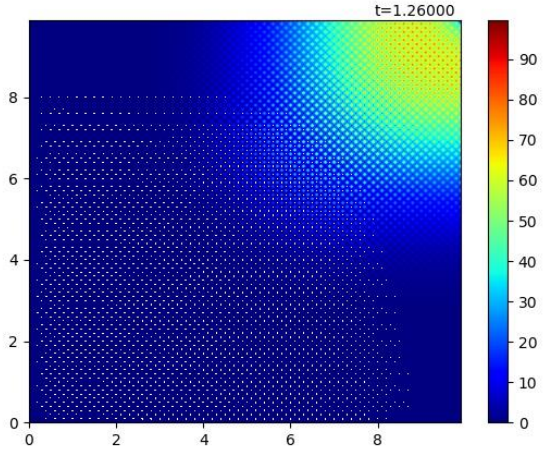


Figure 12 d=0.252 500 loops

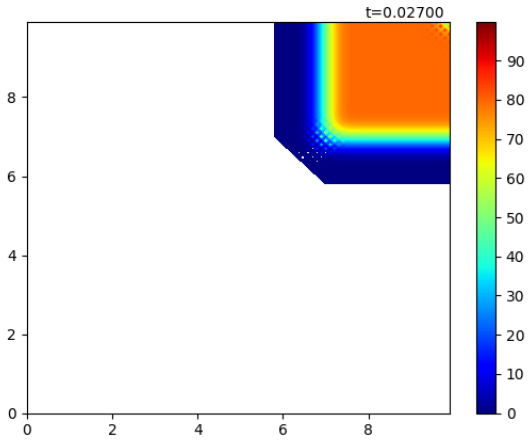


Figure 13 d = 0.3

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

```
import numpy as np
import matplotlib.pyplot as plt
import os
from matplotlib import animation
dir = os.path.dirname(os.path.realpath(__file__))
os.chdir(dir)

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

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

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

```

```
35. x = np.linspace(0,dx * (grid_x - 1), grid_x)
36. y = np.linspace(0,dy * (grid_y - 1), grid_y)
37. X,Y = np.meshgrid(x,y)
38. 
39. cmap = plt.cm.get_cmap("jet")
40. cmap.set_over('grey')
41. g = 20
42. levels = np.arange(0.,100.,0.2)
43. count = 1
44. icount = 0
45. T = init()
46. def cyc(n, plot=True):
47.     global T, icount
48.     Tn = T.copy()
49.     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)
50.     if plot:
51.         plot_activate(X,Y,n)
52.     icount += 1
53. 
54. def dir(n,plot=True):
55.     global T, icount
56.     T[0,:], T[:,0], T[grid_x-1,:], T[:,grid_y-1] = 0,0,0,0
57.     Tn = T.copy()
58.     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)
59.     if plot:
60.         plot_activate(X,Y,n)
61.     icount += 1
62. 
63. #FIX WALL
64. def neu(n, g=0,plot=True):
65.     global T, icount
66.     Tn = T.copy()
67.     left = np.roll(Tn,1,axis=0)
68.     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):

5. Derive and write the discretization as an algebraic equation
6. Investigate by modelling the differences when a (linear) is a constant and when a is u (non-linear) for a cyclic condition.
7. What happens when you set a Dirichlet boundary?
8. List potential applications for Burger's equation.

A. Discretization Algebraic Equation [Q.1]

By these equation

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

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

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

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
```

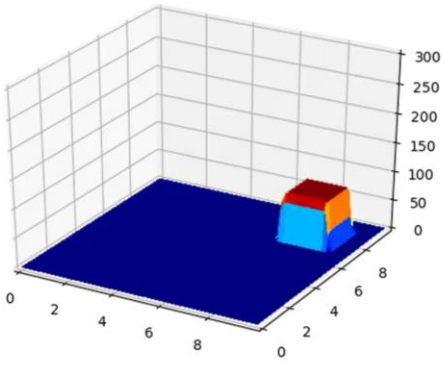


Figure 14 Initial for Observation

Plot Function

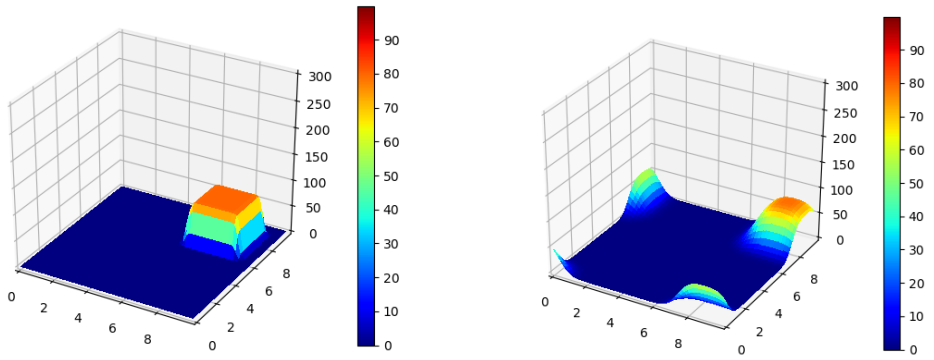
```
fig = plt.figure()
ax = fig.gca(projection='3d')
cl = plt.contourf(X,Y,T,levels,cmap=cmap)
plt.colorbar(cl)

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(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)
```

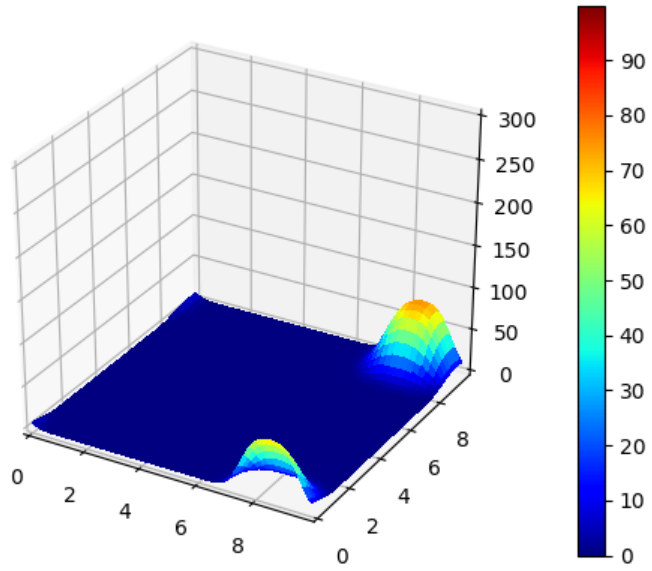


Different to the Diffusion equation, we included advection which cause the blob of heat to move along the $\hat{i}+\hat{j}$ vector (because the advection term $d_2\left(2U^n_{x,y}-U^n_{x-1,y}-U^n_{x,y-1}\right)$ allowed effect from x,y axis equally).

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

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

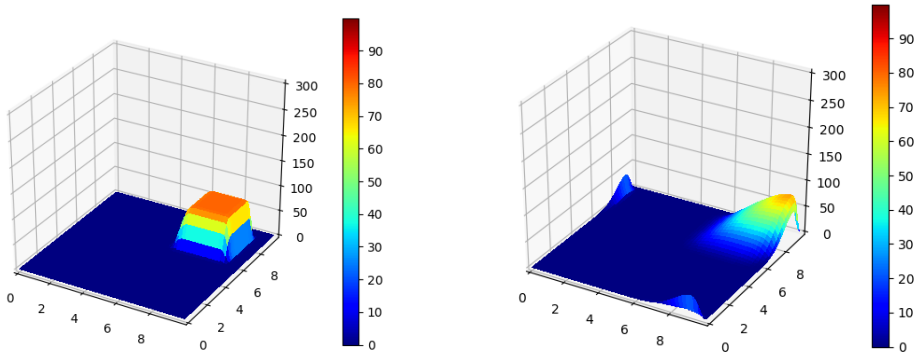
```
1.  def cyc_con(n, plot_type=1):
2.      global T,d1,d2
3.      Tn = T.copy()
4.      left = np.roll(Tn,1,axis=0)
5.      right = np.roll(Tn,-1,axis=0)
6.      up = np.roll(Tn,-1,axis=1)
7.      down = np.roll(Tn,1,axis=1)
8.      T = Tn-d1*(2 * Tn - 1.5*left - 0.5*down) + d2 * (left+right+up+down-4*Tn)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



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

Cyclic (Non-linear – 1)

```
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*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)
```

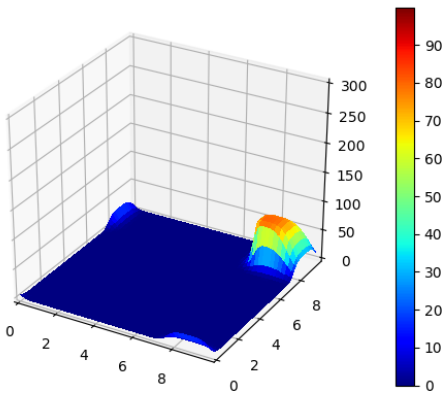


According to the figure, the tip of the heat blob moves faster. This can be assumed by the non-linear factor. By the point where has higher heat, the effect from advection will be stronger be because we multiply itself again from the formula. ($\odot$ is elements wise multiplication)

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

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 moves slower by multiplied with a value of a decreasing conversely to $U_{x,y}^{n}$ which we choose $\sqrt{80^2 - U_{x,y}^{n}}$ (80 is maximum of the heat range)

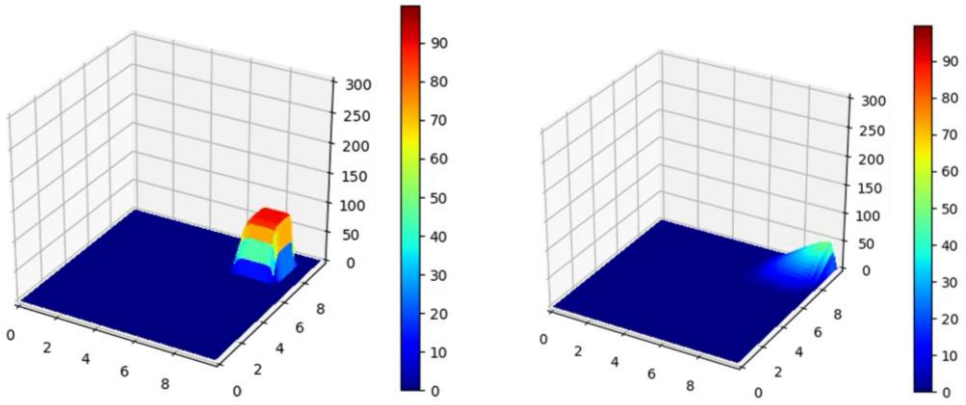
$$\frac{\Delta t}{\Delta x}\sqrt{80^2 - U_{x,y}^{n}} \odot \left(2U_{x,y}^{n} - U_{x-1,y}^{n} - U_{x,y-1}^{n}\right)$$

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)
```

and we set new initial condition

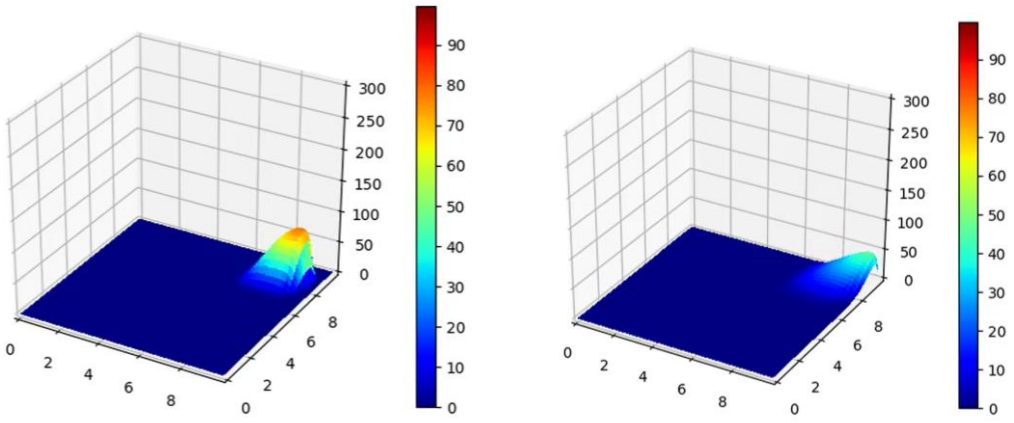
```
a = 2.
```



No heat leaks around the boundaries and the heat gradually lost from the system even faster that just diffusion equation. (the heat throw itself into the “black hole” wall)

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]
    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)
```



Extend from the Dirichlet condition, the wall's heat was affected by the blob of heat that was approaching the it. In this case G=5 so we can see some gradient or some slopes near the 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!)

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

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, $B_0 = e^{-t}\cos\left(\frac{\pi x}{2}\right)$.

Ref:

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()
```

```
35. fig = plt.figure()
36. ax = fig.gca(projection='3d')
37. cl = plt.contourf(X,Y,T,levels,cmap=cmap)
38. plt.colorbar(cl)
39. 
40. def plot_activate(X,Y,n):
41.     global T
42.     plt.cla()
43.     plt.clf()
44.     plt.xlim(0.,np.max(x))
45.     plt.ylim(0.,np.max(x))
46.     plt.contourf(X,Y,T,levels,cmap=cmap)
47.     plt.text(np.max(x)*0.8,np.max(y)+dy,"t=%01.5f"%(dt*n))
48. 
49. def plot3d_activate(X,Y,n):
50.     ax.cla()
51.     ax.set_xlim(0.,np.max(x))
52.     ax.set_ylim(0.,np.max(y))
53.     ax.set_zlim(0.,300)
54.     cl = ax.plot_surface(X,Y,T,linewidth=0,vmin=np.min(levels),vmax=np.max(lev
    els), cmap=cmap,antialiased=False)
55. 
56. def cyc_var(n, plot_type=1):
57.     global T,d1,d2
58.     Tn = T.copy()
59.     left = np.roll(Tn,1,axis=0)
60.     right = np.roll(Tn,-1,axis=0)
61.     up = np.roll(Tn,-1,axis=1)
62.     down = np.roll(Tn,1,axis=1)
63.     T = Tn-dt*(np.sqrt(80**2-
    Tn ** 2))/dx*(2 * Tn - left - down) + d2 * (left+right+up+down-4*Tn)
64.     if plot_type == 1:
65.         plot_activate(X,Y,n)
66.     if plot_type == 2:
67.         plot3d_activate(X,Y,n)
68. 
```

```
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'])
```