

Partial Differential Equations for Science and Engineering

Final Report

November 27, 2017

Thumwanit Napat

16B00133

Contents

[Problem 1 Diffusion Equation 2](#_Toc498290424)

[A. Effect of Initial Condition (By Neumann) [Q.1] 3](#_Toc498290425)

[B. Observation of Conditions [Q.2+4] 5](#_Toc498290426)

[- Desired Initial 5](#_Toc498290427)

[- Cyclic 6](#_Toc498290428)

[- Dirichlet 8](#_Toc498290429)

[- Neumann (G=0) 10](#_Toc498290430)

[Neumann (G=20) 12](#_Toc498290431)

[C. Effect of d’s value [Q.3] 13](#_Toc498290432)

[D. Python Code 15](#_Toc498290433)

[Problem 2 Burger’s Equation 18](#_Toc498290434)

[A. Discretization Algebraic Equation [Q.1] 19](#_Toc498290435)

[B. Observation of Modelling and Condition [Q.2+3] 20](#_Toc498290436)

[Initial Value 20](#_Toc498290437)

[Plot Function 21](#_Toc498290438)

[Cyclic (Constant) 22](#_Toc498290439)

[Cyclic (Non-linear – 1) 24](#_Toc498290440)

[Cyclic (Non-linear – 2) 25](#_Toc498290441)

[Dirichlet (Non-linear) 26](#_Toc498290442)

[Neumann (Non-linear, G=5) 27](#_Toc498290443)

[C. Application of Burger’s Equation [Q.4] 29](#_Toc498290444)

[D. Python Code 30](#_Toc498290445)

Table of Figures

[Figure 1 Initial condition set to 100 and only to regions aa 14](#_Toc497723333)

# Problem 1 Diffusion Equation

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



Discuss the following (add figures if necessary):

1. Influence of the initial condition. Test various initial conditions or distributions.
2. Investigate various boundary conditions:
   1. Dirichlet Boundary condition
   2. Neumann Boundary condition
3. Investigate what is the influence of 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.

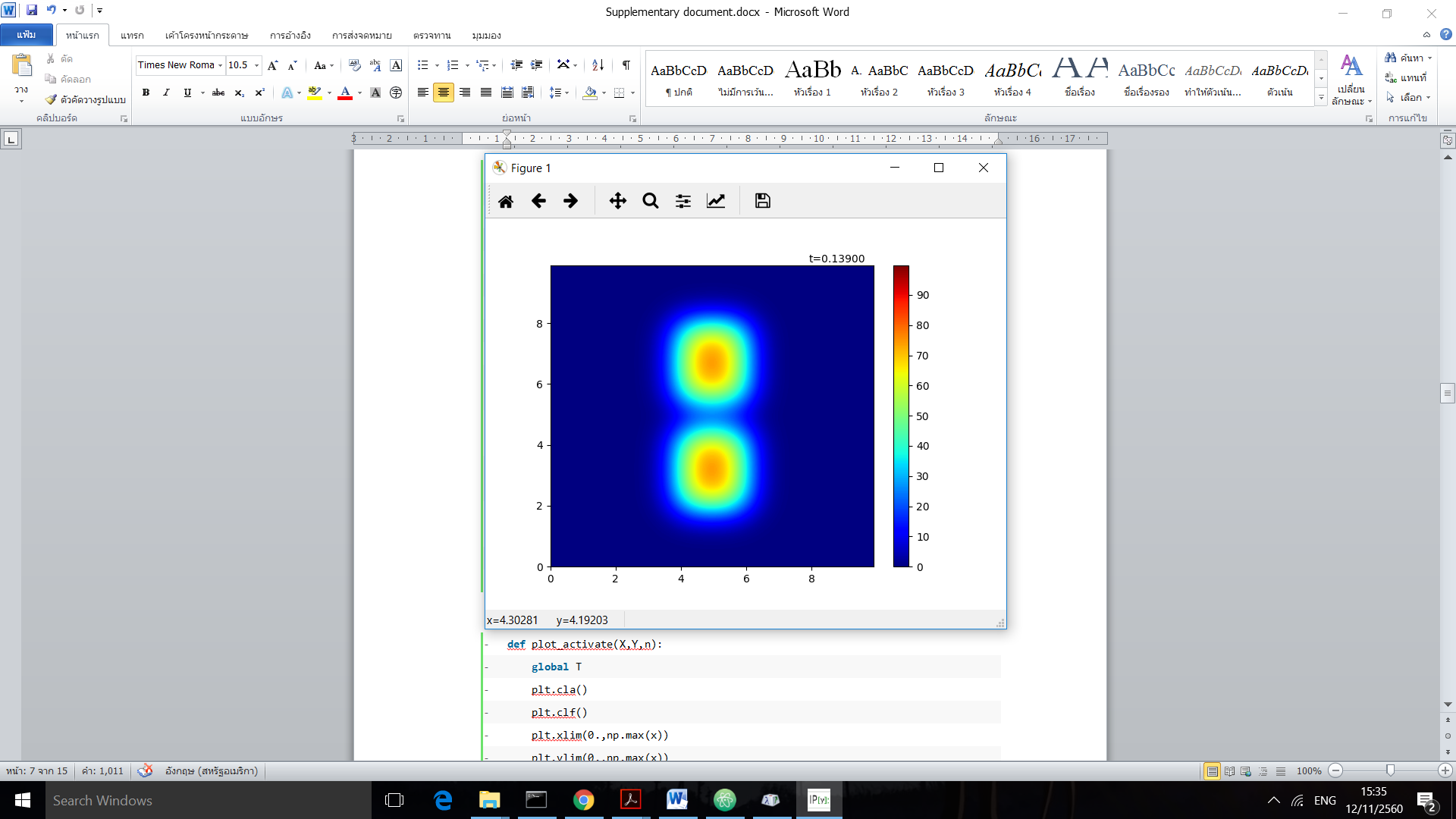
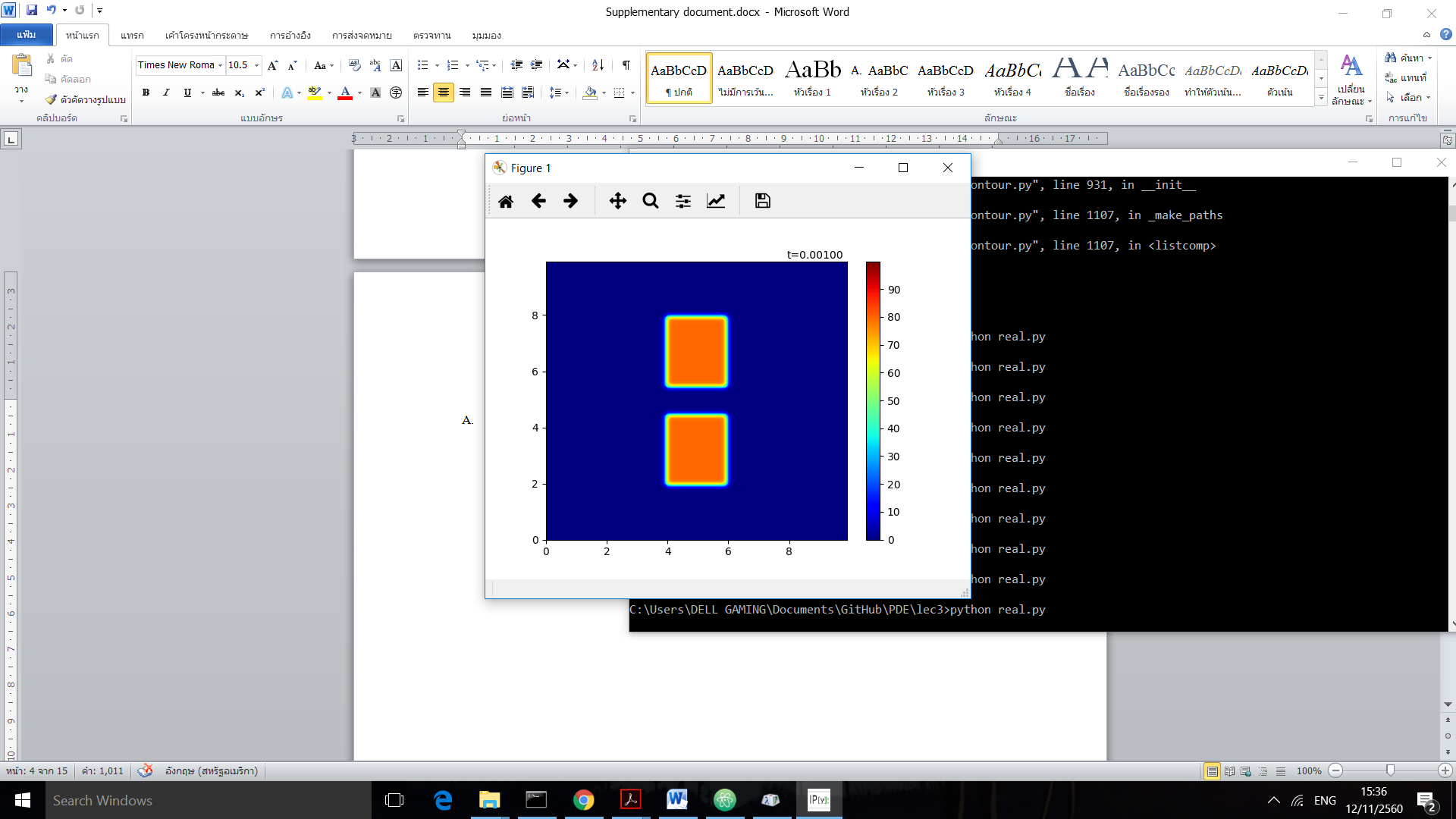
## Effect of Initial Condition (By Neumann) [Q.1]

Initial Condition

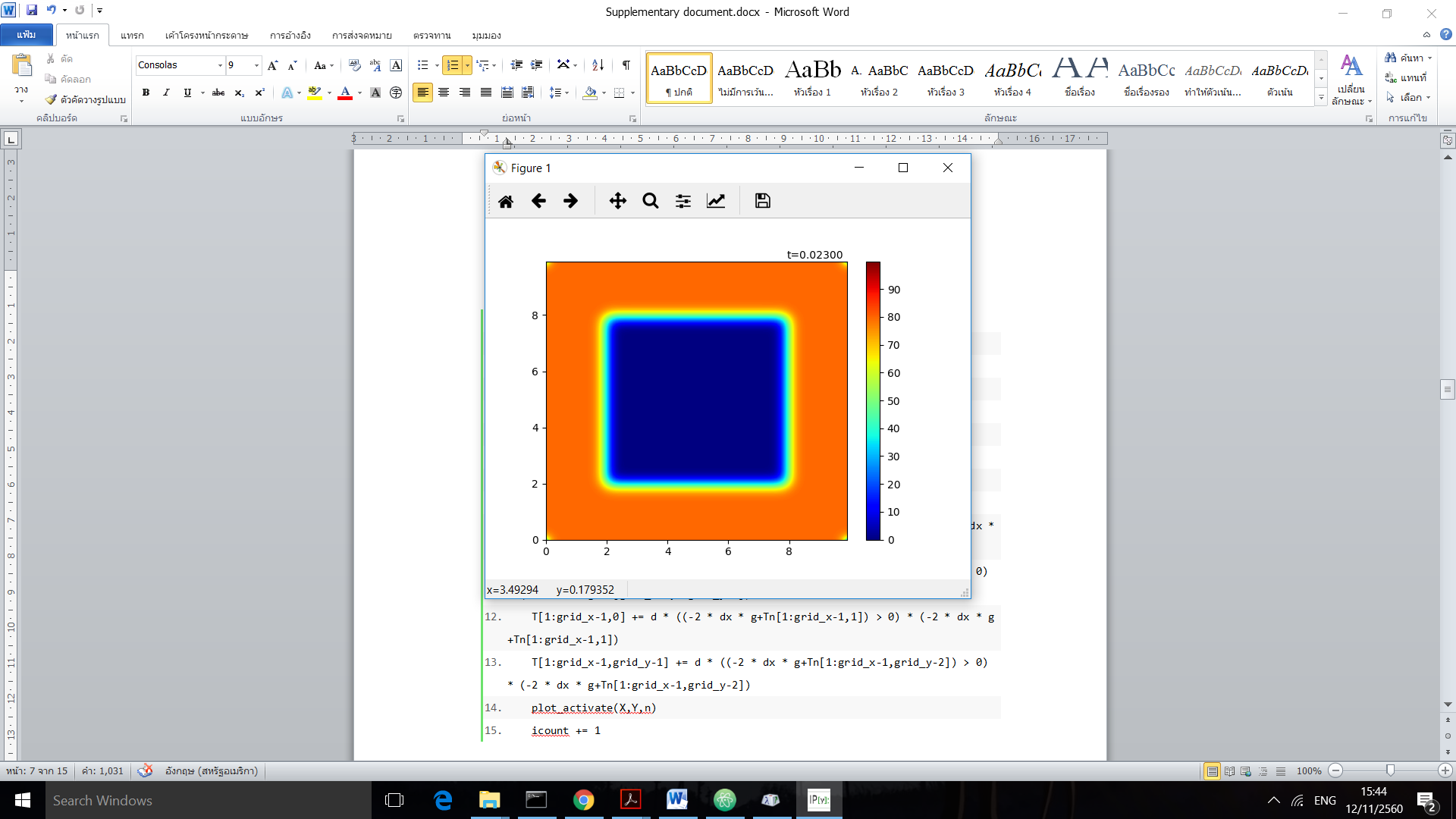
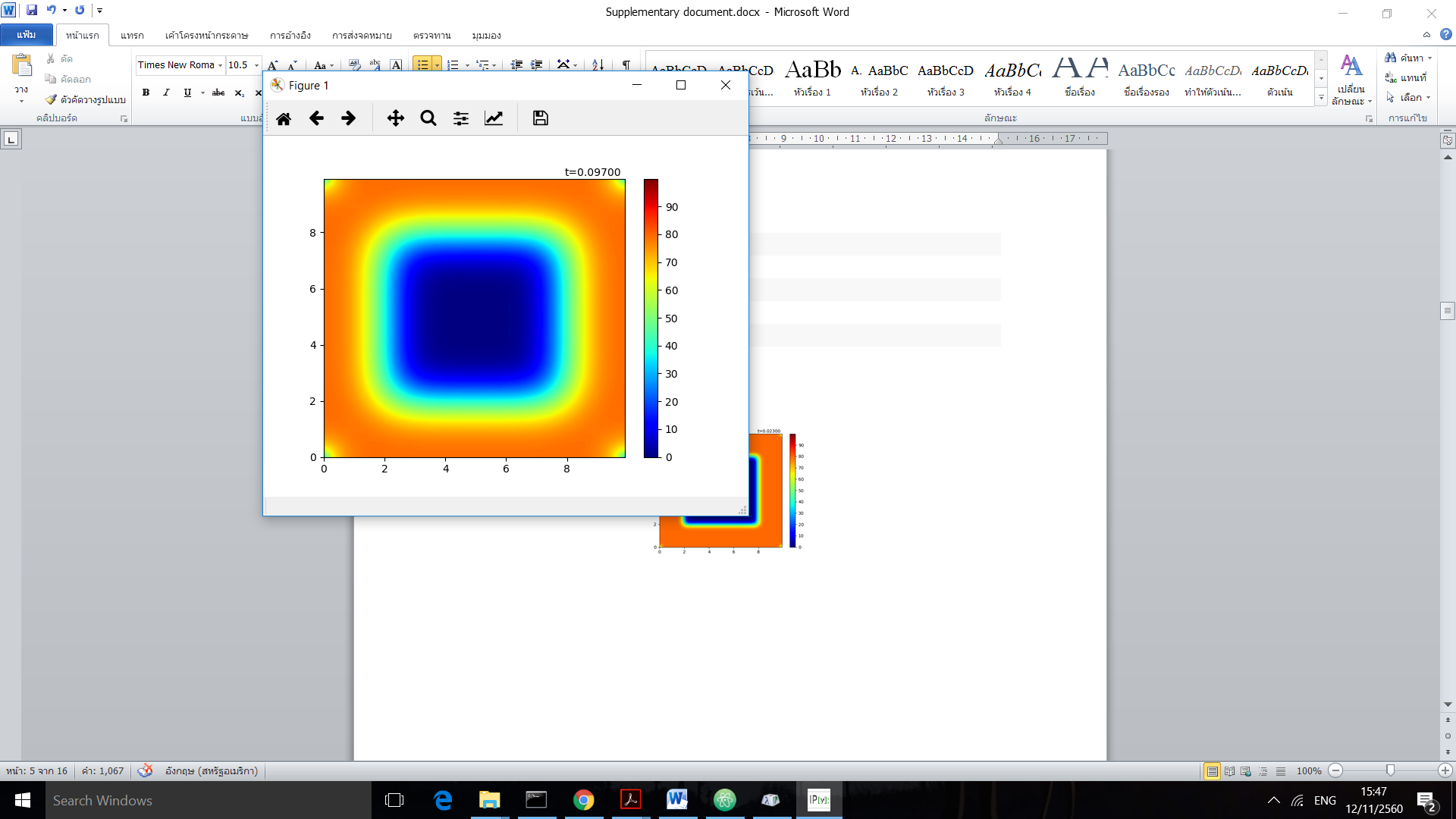
1. dx = 0.1
2. dy = dx
3. alpha = 1.
4. grid\_x = 100
5. grid\_y = 100
6. nt = 100
7. d = 0.1
8. dt = d \* (dx\*\*2)/alpha
10. **def** init():
11. T = np.zeros((grid\_x,grid\_y))
12. T[70:grid\_x, 70:grid\_y] = 80
13. **return** T
14. T = init()

* couple blobs

1. **def** init():
2. T = np.zeros((grid\_x,grid\_y))
3. T[20:45, 40:60] = 80.
4. T[55:80, 40:60] = 80.
5. **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

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

### Desired Initial

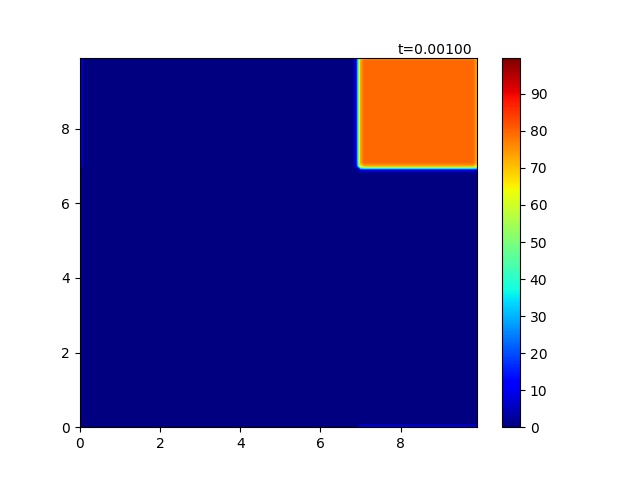


Figure 1 initial condition for observation

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

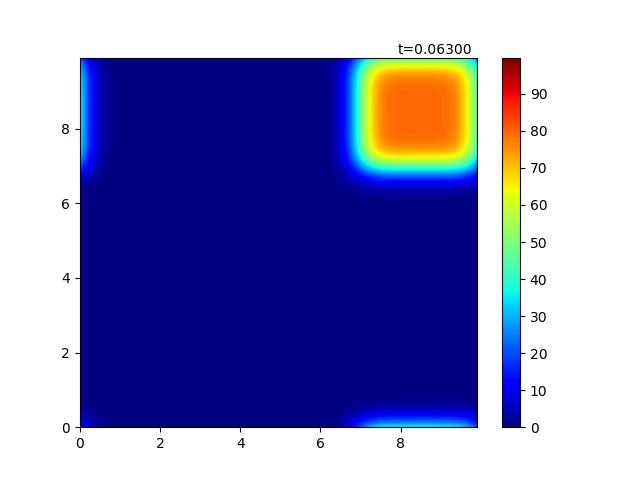


Figure 2 middle state of cyclic condition

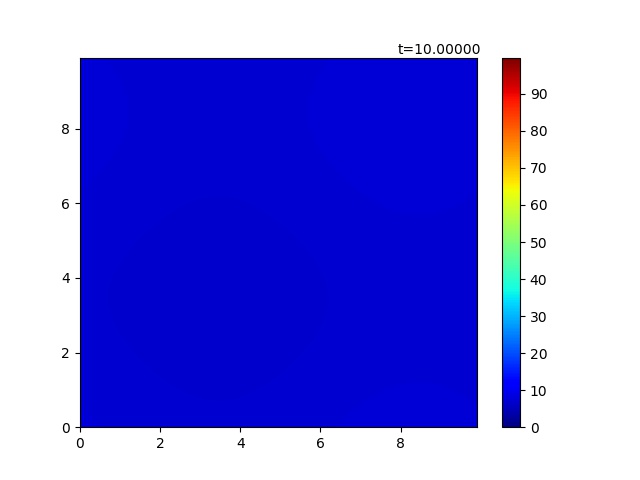
Unfortunately, our np.roll() function rolls over the matrix and the temperature somehow leaks to the other side. However, the dispersion can be observed cleary.

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

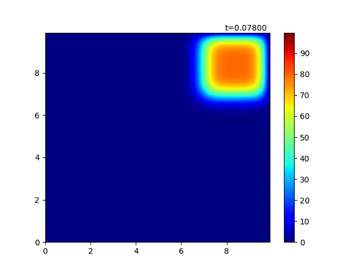


Figure 4 middle state of Dirichlet condition

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.

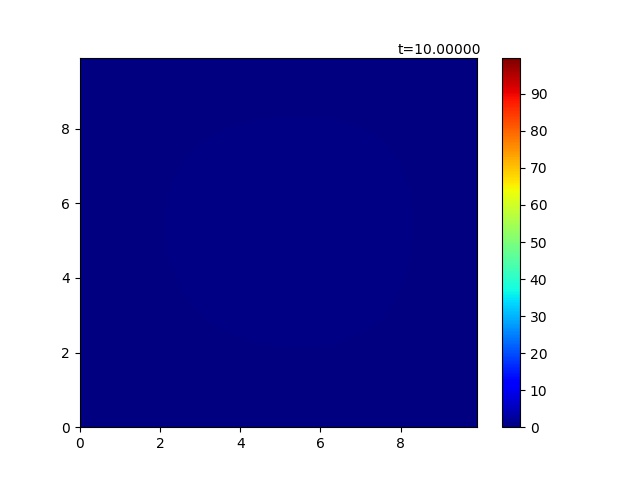


Figure 5 final state of Dirichlet condition

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

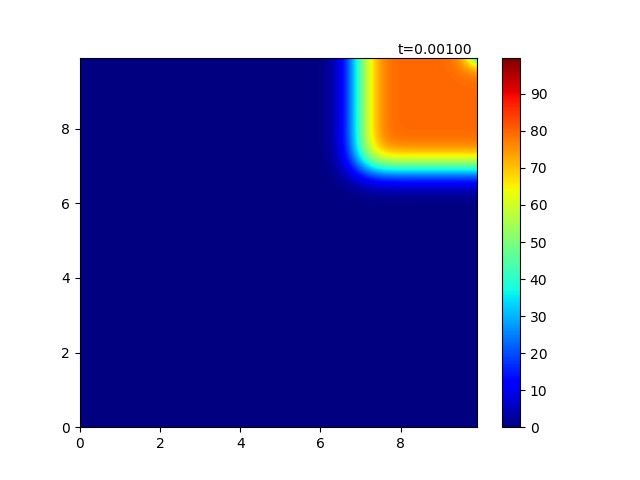
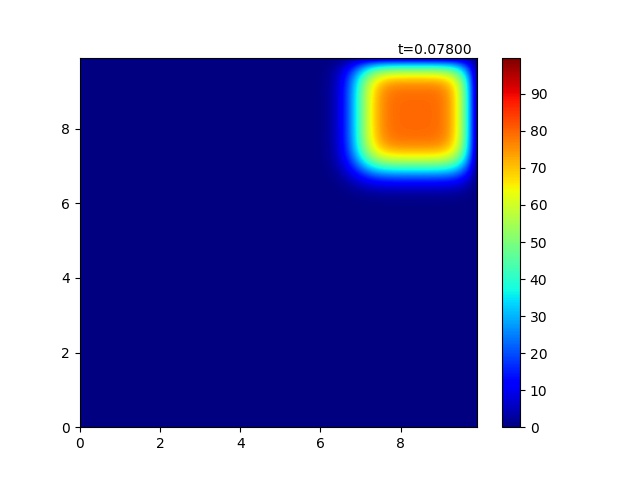


Figure 6 middle state of Neumann condition

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]

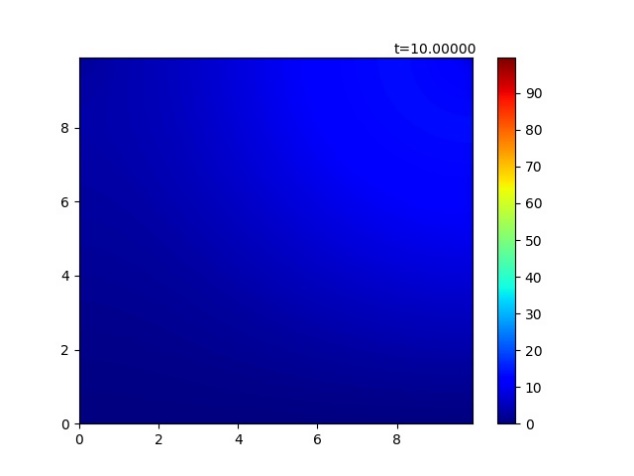


Figure 7 almost final state of Neumann condition

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.

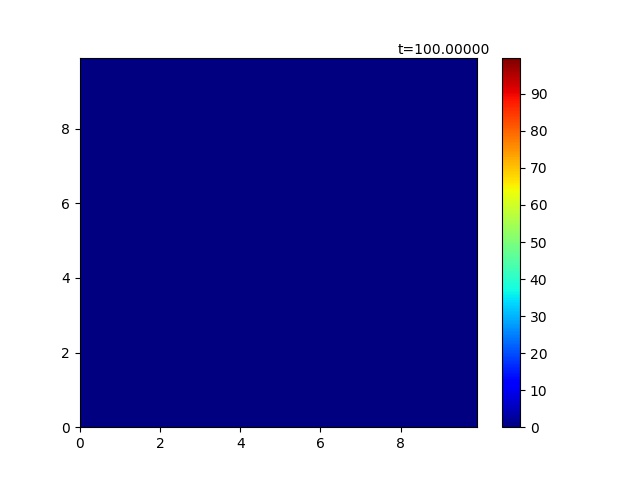


Figure 8 final state of Neumann condition

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)

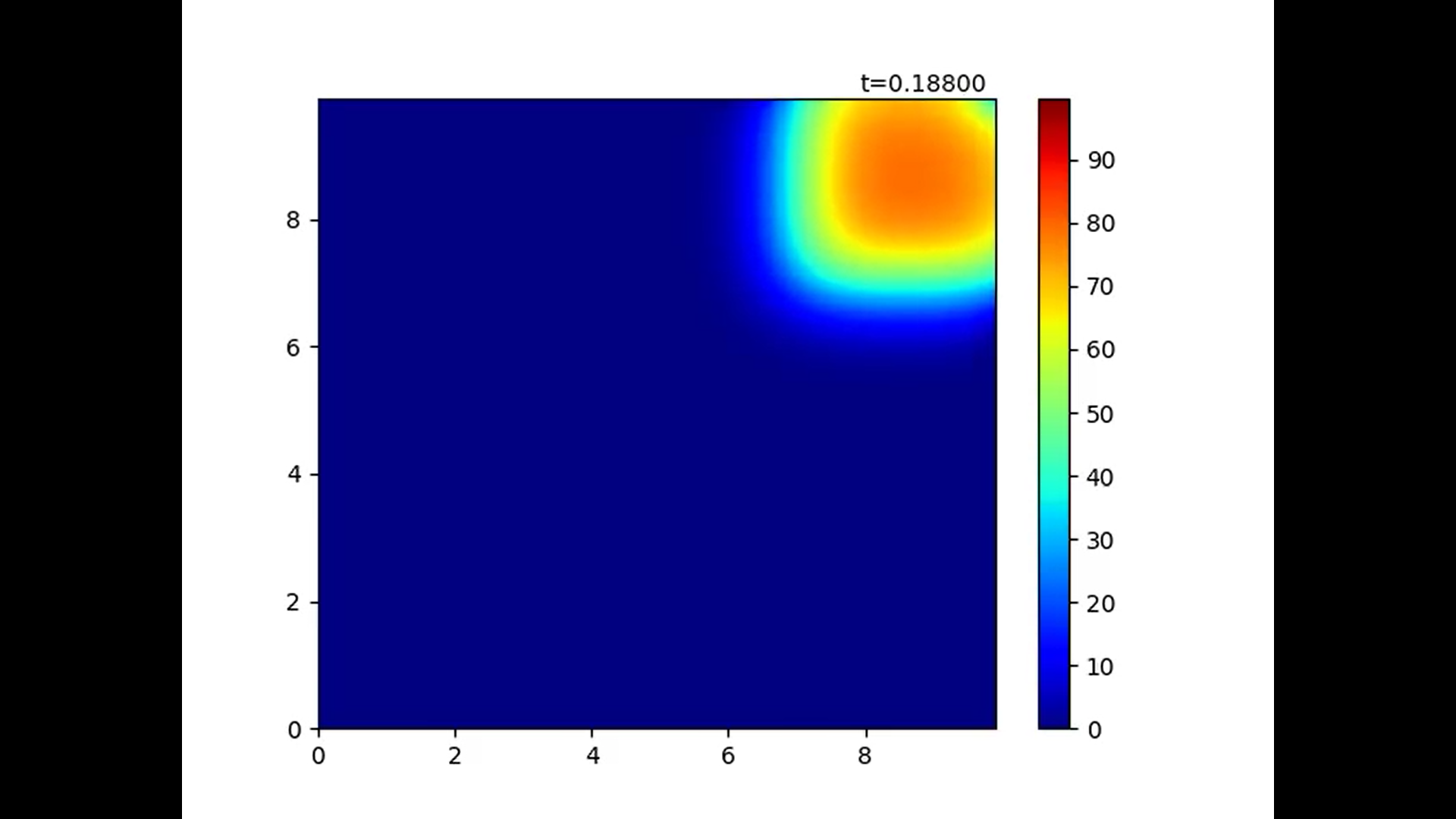


Figure 9 middle state of Neumann condition (G≠0)

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.

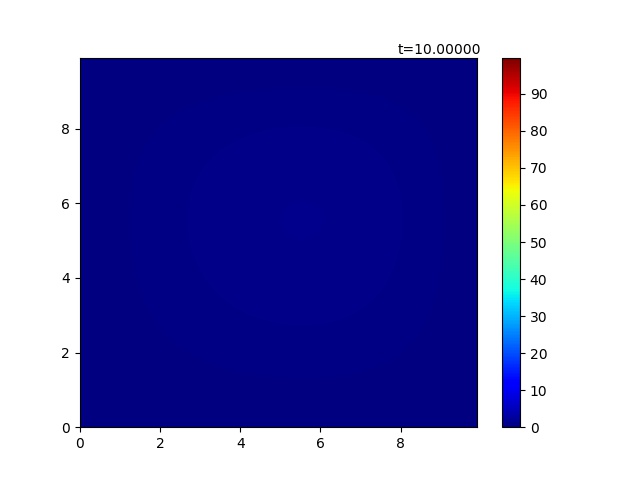


Figure 10 final state of Neumann condition (G≠0)

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

## Effect of d’s value [Q.3]

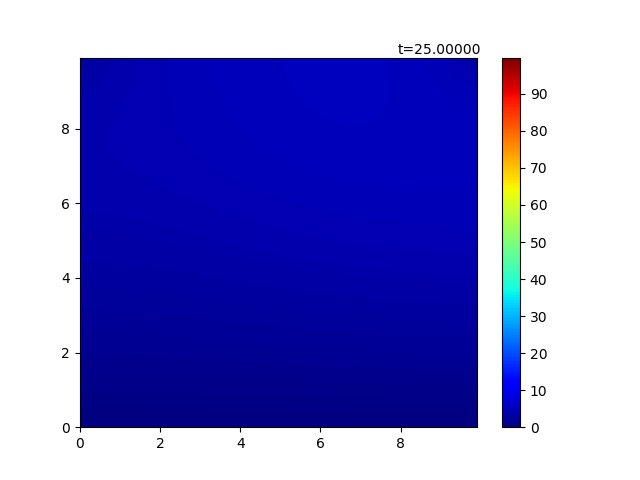


Figure 11 d=0.25 10000 loops

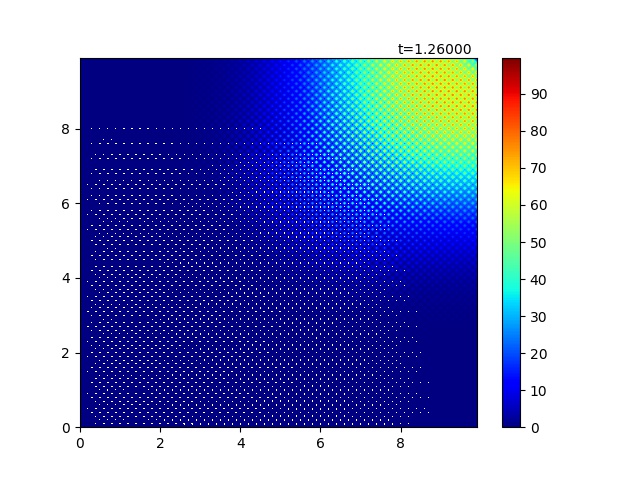


Figure 12 d=0.252 500 loops

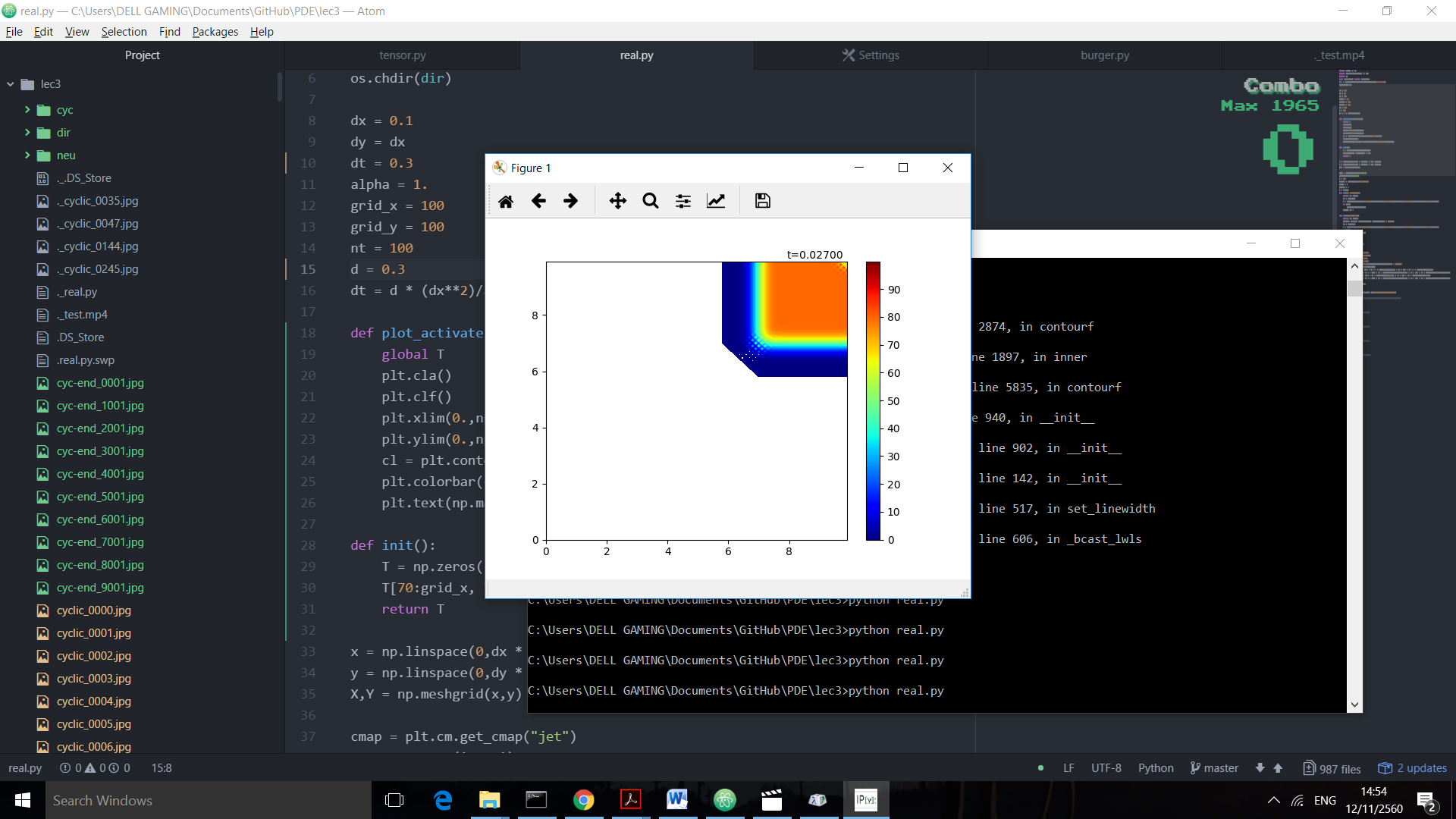


Figure 13 d = 0.3

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.

## 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)
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
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))
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
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)
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
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
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)
69. up = np.roll(Tn,-1,axis=1)
70. down = np.roll(Tn,1,axis=1)
71. left[0,:],right[grid\_x-1,:],up[:,grid\_y-1],down[:,0] = 0,0,0,0
72. T = Tn+d\*(up+down+right+left-4\*Tn)
73. 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])
74. 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])
75. 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])
76. 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])
77. **if** plot:
78. plot\_activate(X,Y,n)
79. icount += 1
80. fig = plt.figure()
81. a = animation.FuncAnimation(fig, neu, fargs=(g,),frames=200,interval=10)
82. 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.



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.

## Discretization Algebraic Equation [Q.1]

By these equation







We can derive the algebraic equation



If we let  and , we can rewrite our equation as



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

### Initial Value

1. dx = 0.1
2. dy = dx
3. a = 1.
4. v = 2.
5. grid\_x = 100
6. grid\_y = 100
7. nt = 100
8. d1 = 0.1
9. d2 = 0.05
10. dt = d2 \* (dx\*\*2)/v
12. **def** init():
13. T = np.zeros((grid\_x,grid\_y))
14. T[70:90, 70:90] = 80.
15. **return** T

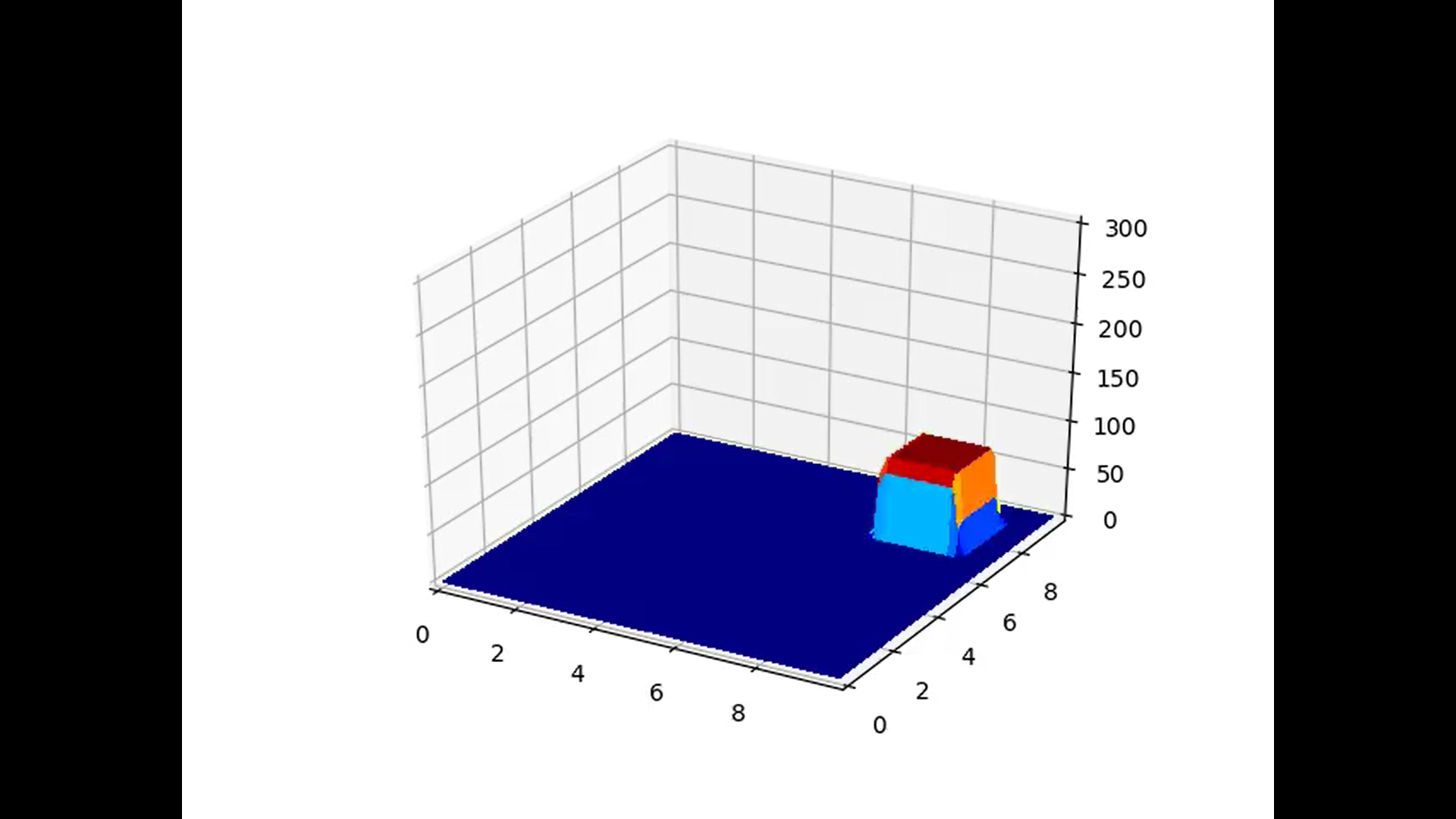


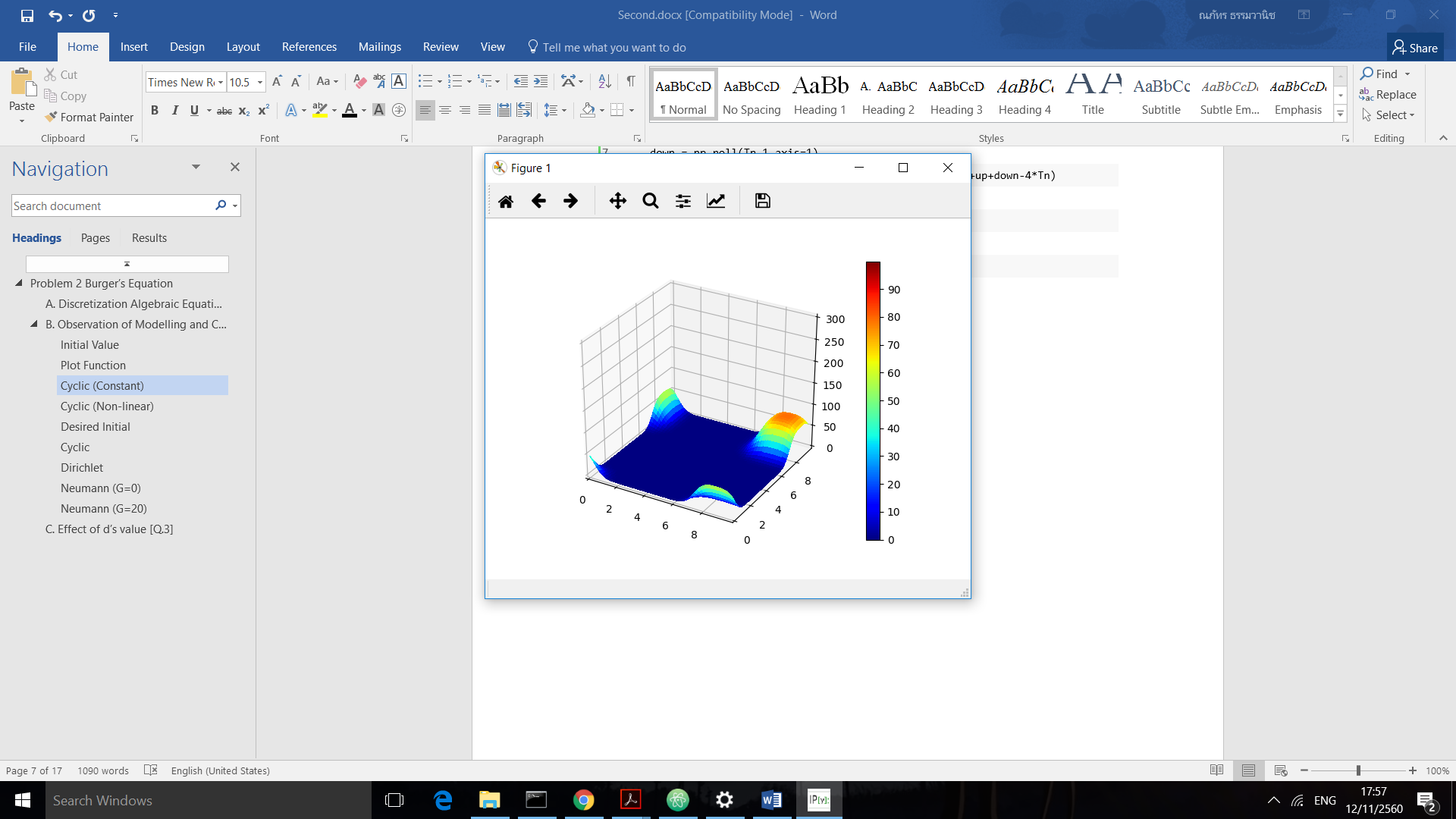
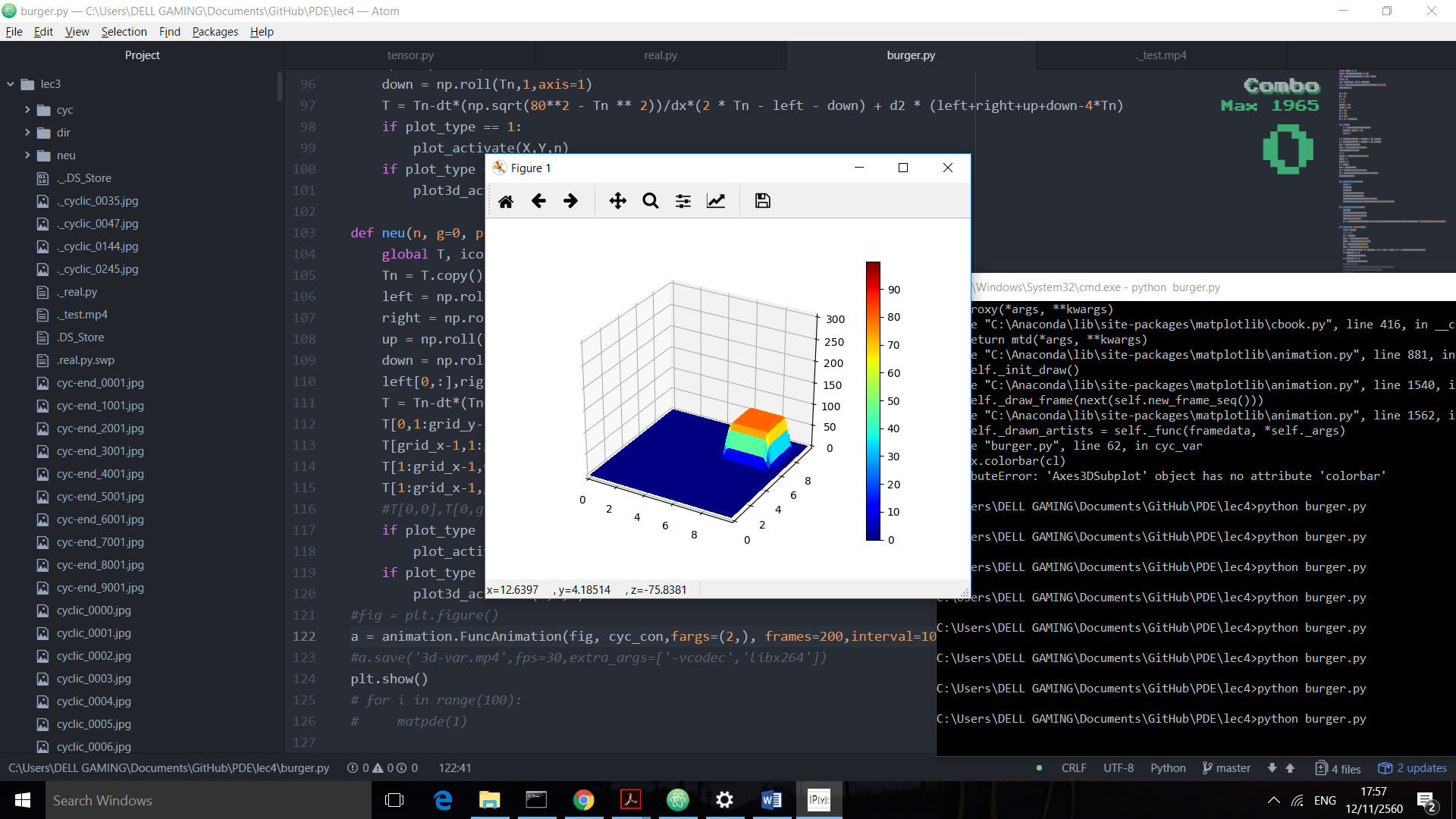
Figure 14 Initial for Observation

### 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)
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))
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(levels), cmap=cmap,antialiased=False)

### Cyclic (Constant)

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 - 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)

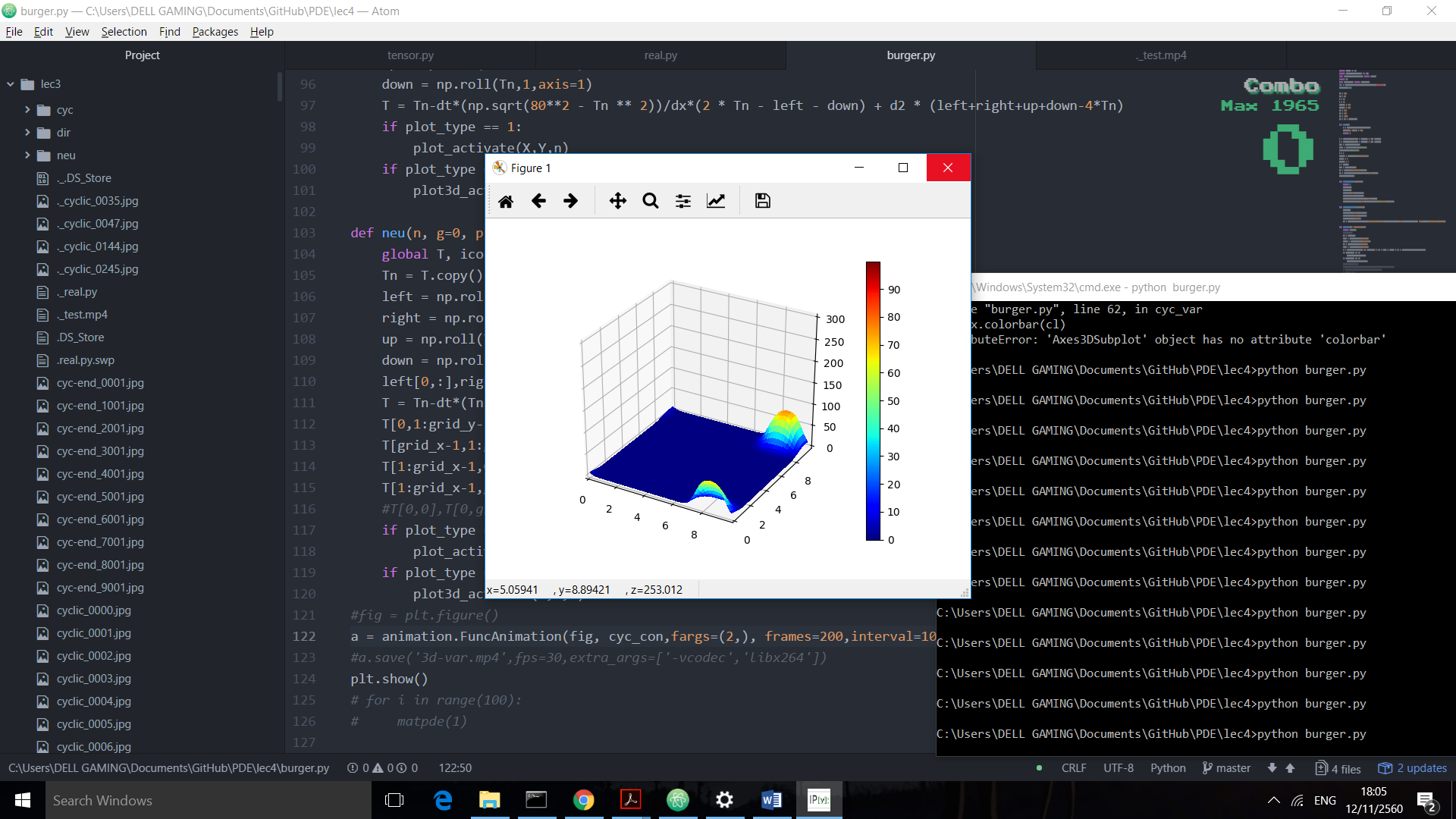


Different to the Diffusion equation, we included advection which cause the blob of heat to move along the  vector ( because the advection term allowed effect from x,y axis equally).

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



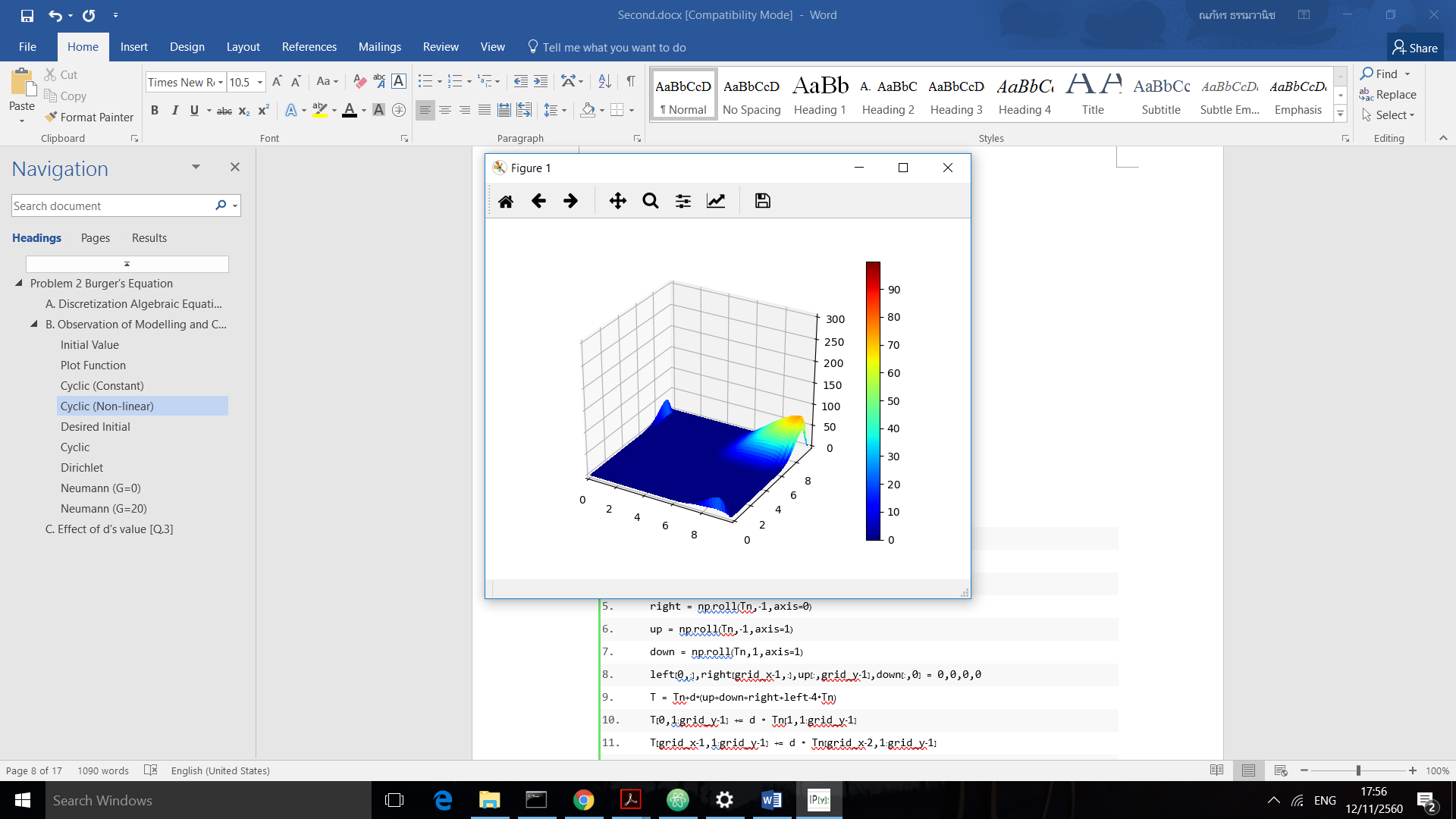
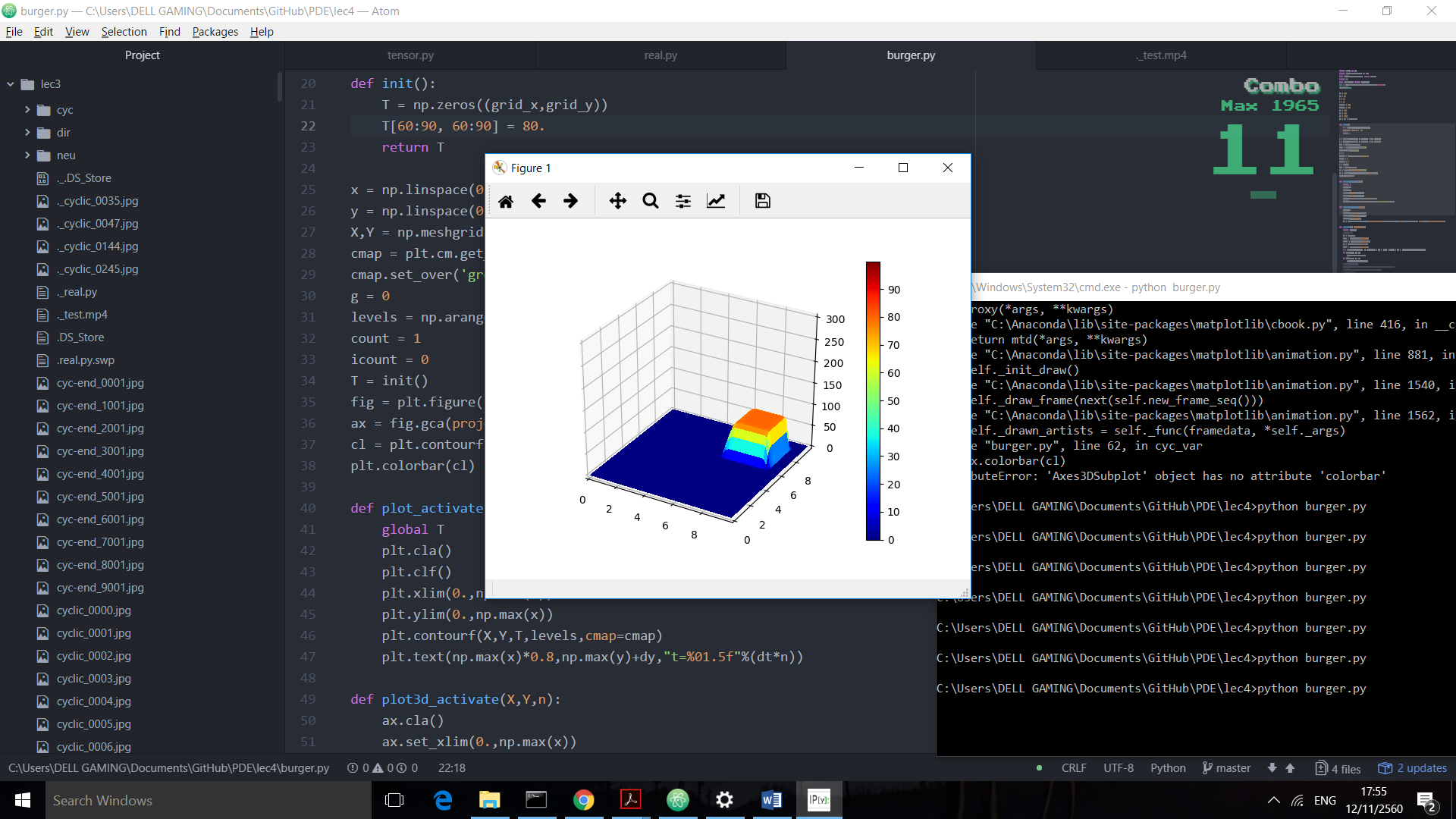
1. **def** cyc\_con(n, plot\_type=1):
2. **global** T,d1,d2
3. Tn = T.copy()
4. left = np.roll(Tn,1,axis=0)
5. right = np.roll(Tn,-1,axis=0)
6. up = np.roll(Tn,-1,axis=1)
7. down = np.roll(Tn,1,axis=1)
8. **T = Tn-d1\*(2 \* Tn - 1.5\*left - 0.5\*down) + d2 \* (left+right+up+down-4\*Tn)**
9. **if** plot\_type == 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)

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)

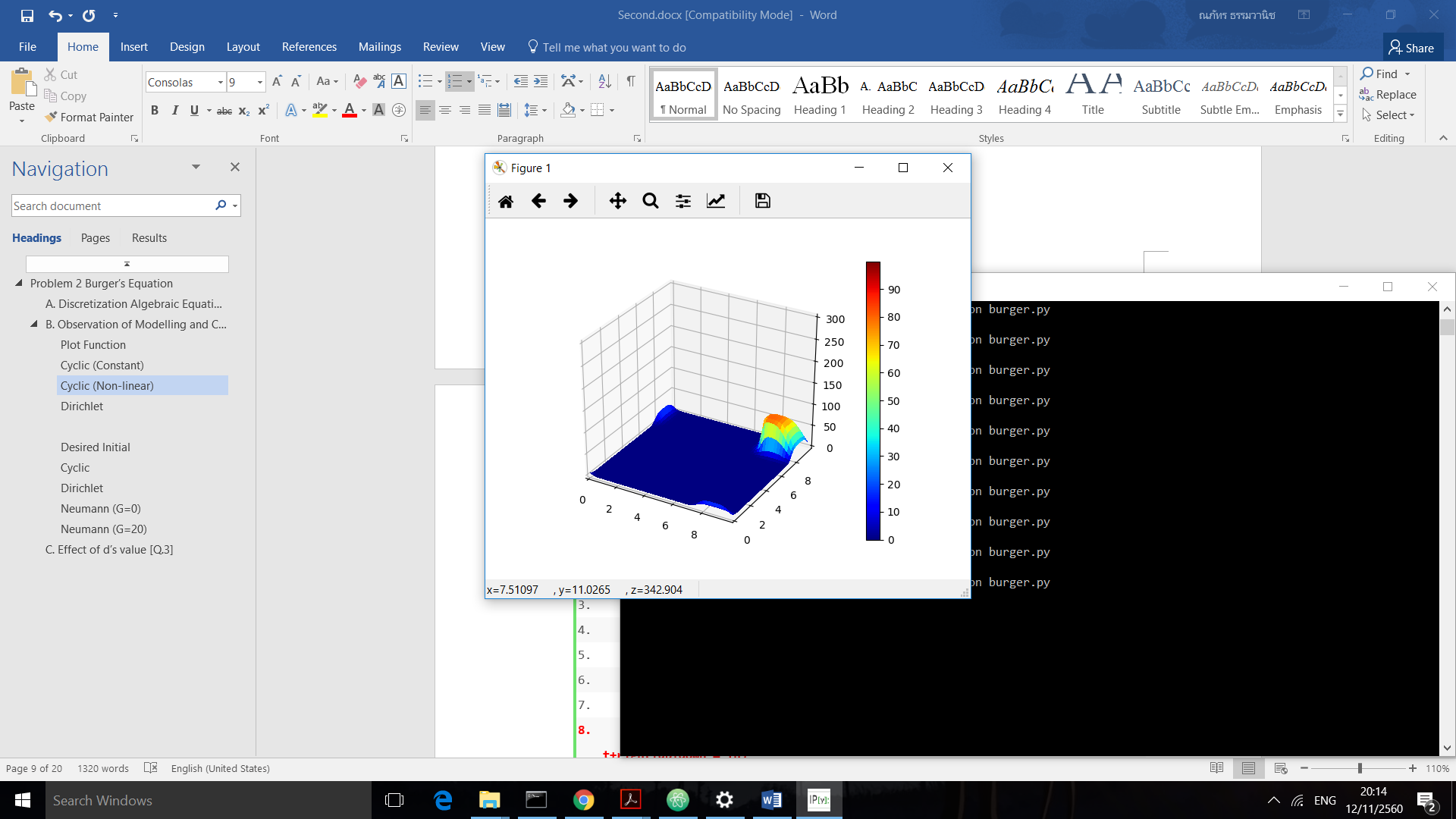


According to the figure, the tip of the heat blob moves faster. This can be assumed by the non-linear factor. By the point where has higher heat, the effect from advection will be stronger be because we multiply itself again from the formula. (is elements wise multiplication)



### 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  decreasing conversely to  which we choose (80 is maximum of the heat range)

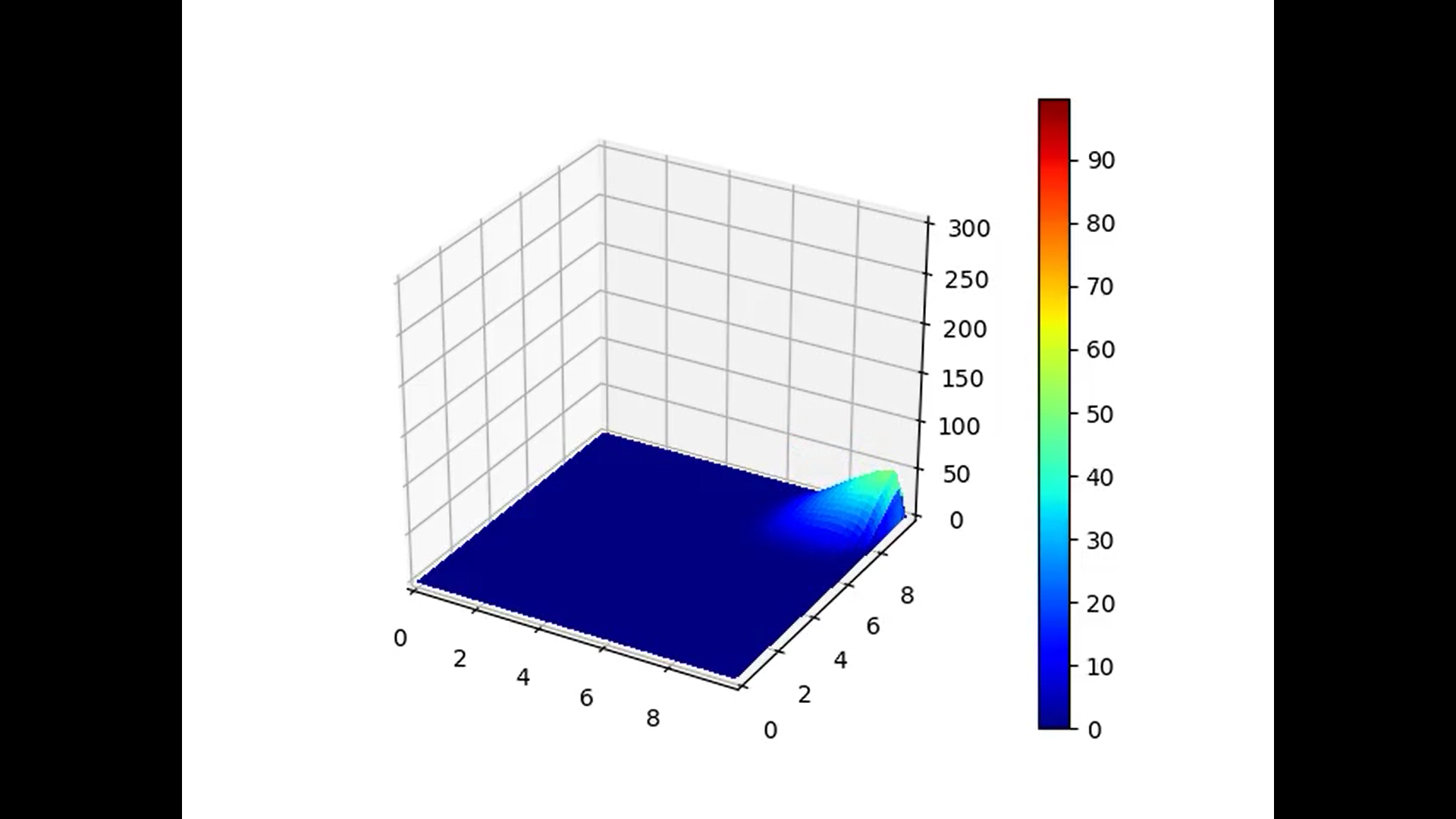
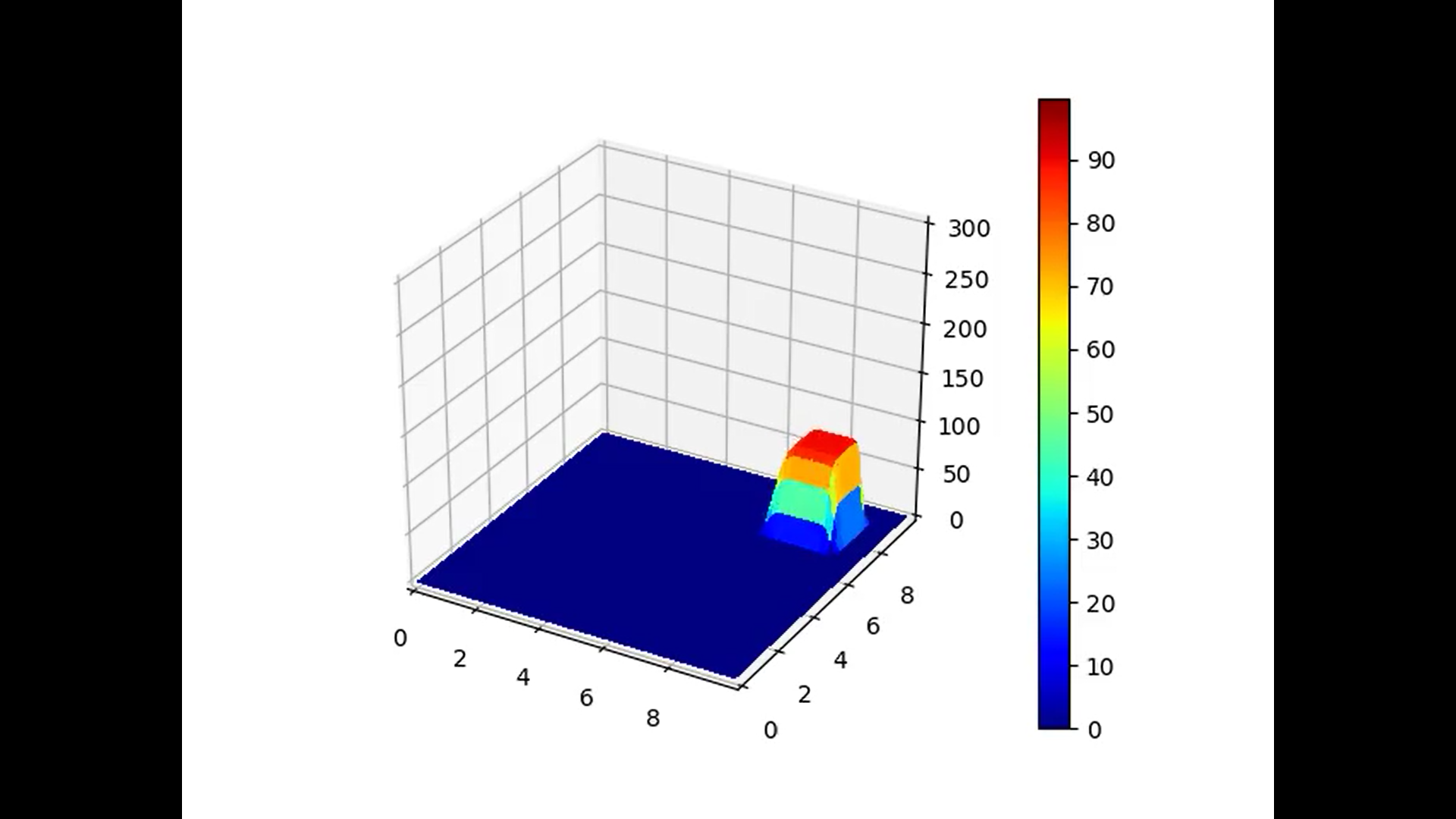


### Dirichlet (Non-linear)

1. **def** der\_var(n, plot\_type=1):
2. **global** T,d1,d2
3. T[0,:],T[:,0], T[grid\_x-1,:], T[:, grid\_y-1] = 0,0,0,0
4. Tn = T.copy()
5. left = np.roll(Tn,1,axis=0)
6. right = np.roll(Tn,-1,axis=0)
7. up = np.roll(Tn,-1,axis=1)
8. down = np.roll(Tn,1,axis=1)
9. T = Tn-dt \*Tn /dx\*(2 \* Tn - left - down) + d2 \* (left+right+up+down-4\*Tn)
10. **if** plot\_type == 1:
11. plot\_activate(X,Y,n)
12. **if** plot\_type == 2:
13. plot3d\_activate(X,Y,n)

and we set new initial condition

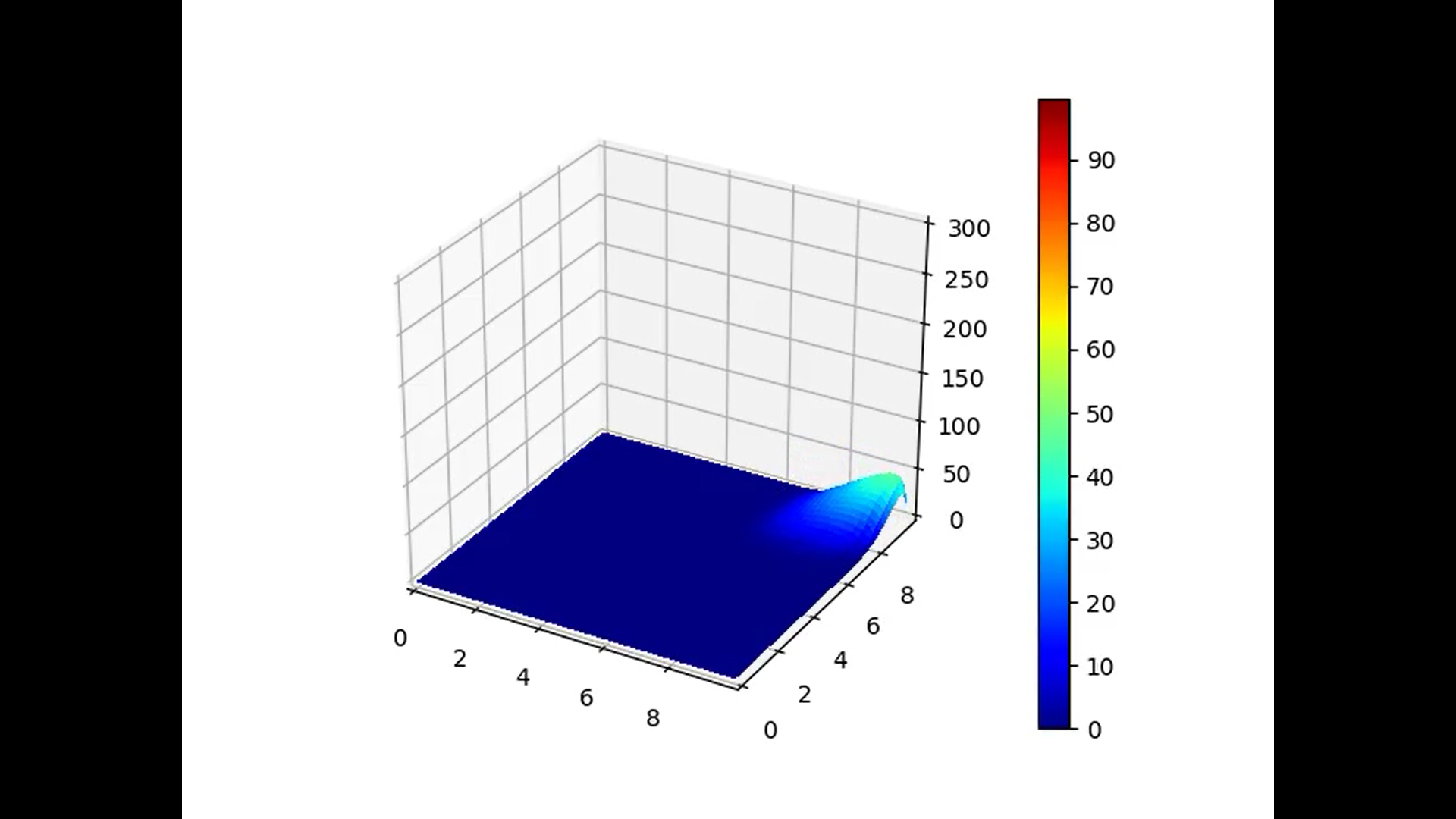
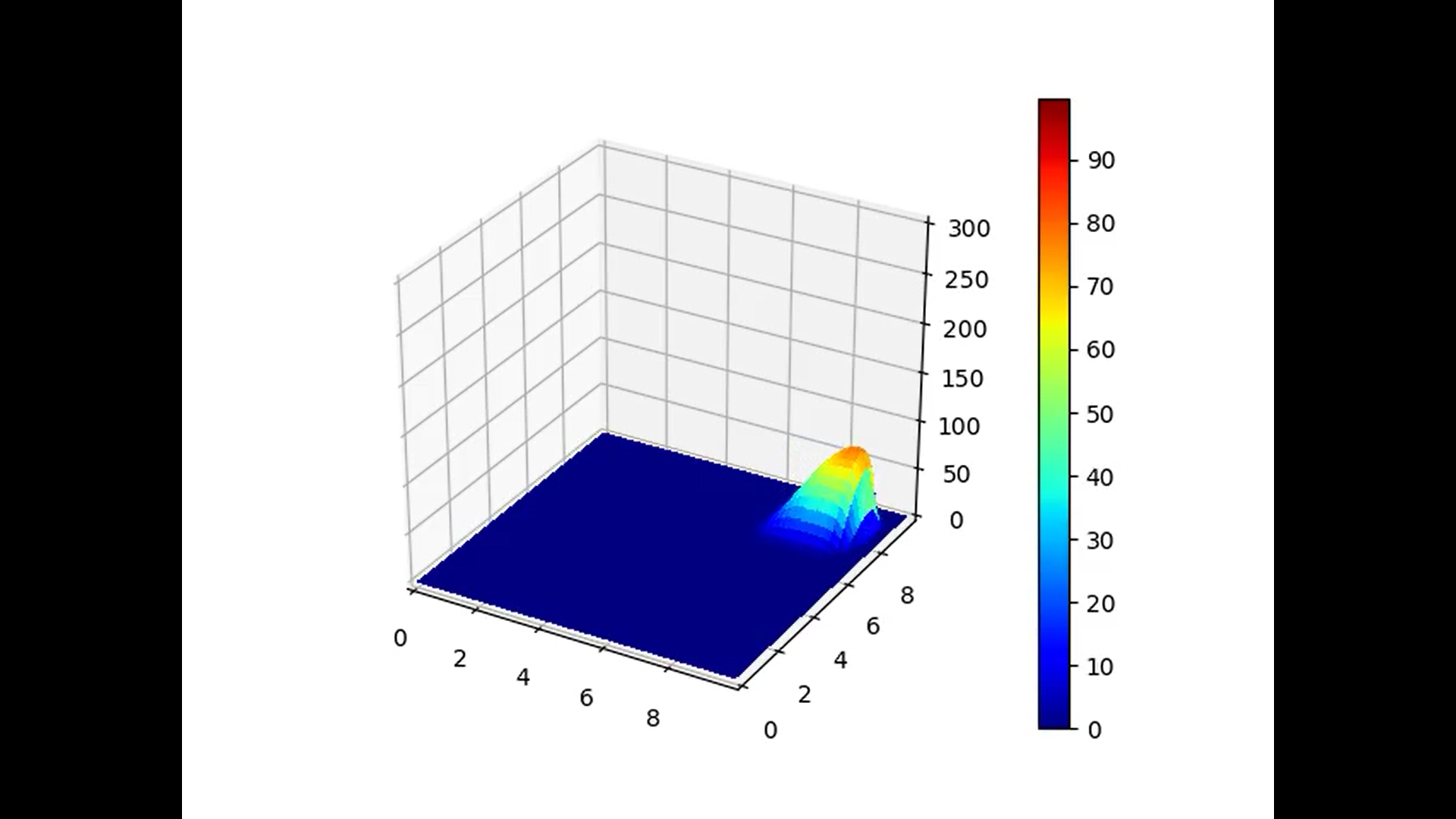
1. a = 2.



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)

1. **def** neu(n, g=0, 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. ghost1 = -2 \* dx \* g + Tn[1,1:grid\_y-1]
9. ghost2 = -2 \* dx \* g + Tn[grid\_x-2,1:grid\_y-1]
10. ghost3 = -2 \* dx \* g + Tn[grid\_x-2,1:grid\_y-1]
11. ghost4 = -2 \* dx \* g + Tn[1:grid\_x-1,grid\_y-2]
12. ghost1 = ghost1 > 0 \* ghost1  #prevent negative value
13. ghost2 = ghost2 > 0 \* ghost2
14. ghost3 = ghost3 > 0 \* ghost3
15. ghost4 = ghost4 > 0 \* ghost4
16. left[0,:],right[grid\_x-1,:],up[:,grid\_y-1],down[:,0] = 0,0,0,0
17. T = Tn-dt\* Tn/dx\*(2 \* Tn - left - down) + d2 \* (left+right+up+down-4\*Tn)
18. T[0,1:grid\_y-1] += d2 \* ghost1 + dt \* T[0,1:grid\_y-1]  / dx\*ghost1
19. T[grid\_x-1,1:grid\_y-1] += d2 \* ghost2
20. T[1:grid\_x-1,0] += d2 \* ghost3 + dt \* T[1:grid\_x-1,0]  / dx \* ghost3
21. T[1:grid\_x-1,grid\_y-1] += d2 \* ghost4
22. **if** plot\_type == 1:
23. plot\_activate(X,Y,n)
24. **if** plot\_type == 2:
25. 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.

## 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



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



1. 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  is the density of the cars (has meaning as bumper hit the front car!)



1. 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.



Where  is applied magnetic field. For example, .

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

## 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)
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
20. **def** init():
21. T = np.zeros((grid\_x,grid\_y))
22. T[70:90, 70:90] = 80.
23. **return** T
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)
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))
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(levels), cmap=cmap,antialiased=False)
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)
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)
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)
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)
102. down = np.roll(Tn,1,axis=1)
103. ghost1 = -2 \* dx \* g + Tn[1,1:grid\_y-1]
104. ghost2 = -2 \* dx \* g+Tn[grid\_x-2,1:grid\_y-1]
105. ghost3 = -2 \* dx \* g+Tn[grid\_x-2,1:grid\_y-1]
106. ghost4 = -2 \* dx \* g+Tn[1:grid\_x-1,grid\_y-2]
107. ghost1 = ghost1 > 0 \* ghost1
108. ghost2 = ghost2 > 0 \* ghost2
109. ghost3 = ghost3 > 0 \* ghost3
110. ghost4 = ghost4 > 0 \* ghost4
111. left[0,:],right[grid\_x-1,:],up[:,grid\_y-1],down[:,0] = 0,0,0,0
112. T = Tn-dt\*(np.sqrt(Tn \*\* 2))/dx\*(2 \* Tn - left - down) + d2 \* (left+right+up+down-4\*Tn)
113. T[0,1:grid\_y-1] += d2 \* ghost1 + dt \* T[0,1:grid\_y-1]/dx\*ghost1
114. T[grid\_x-1,1:grid\_y-1] += d2 \* ghost2
115. T[1:grid\_x-1,0] += d2 \* ghost3 + dt \* T[1:grid\_x-1,0]/dx\*ghost3
116. T[1:grid\_x-1,grid\_y-1] += d2 \* ghost4
117. **if** plot\_type == 1:
118. plot\_activate(X,Y,n)
119. **if** plot\_type == 2:
120. plot3d\_activate(X,Y,n)
121. a = animation.FuncAnimation(fig, der\_var,fargs=(2), frames=200,interval=10)
122. a.save('3d-der-var.mp4',fps=30,extra\_args=['-vcodec','libx264'])