

# 530.767 CFD

## Spring 2024

### HW 3–Haobo Zhao

March 15, 2024

In this project, we discretized the viscous Burgers' equation using the Central Difference scheme for spatial differentiation, the Forward Euler for the convection term, and the Backward Euler for the diffusion term to handle temporal derivatives. We employed an ADI solver to obtain a tridiagonal matrix and implemented the TDMA to derive the results.

We investigated the effects of grid size, boundary conditions, and viscosity on the outcomes. Our findings indicate that as the grid becomes finer, diffusion tends to be slower and less pronounced, mirroring the impact of reduced viscosity.

We also compared the effects of Dirichlet and Neumann boundary conditions where Neumann boundary condition shows more close to the accurate result as the interior flow do not hugely effected by the artifically limit of the domain like Dirichlet boundary condition.

## Contents

<b>1</b>	<b>Question Review</b>	<b>2</b>
<b>2</b>	<b>1. Discretized Equation</b>	<b>3</b>
<b>3</b>	<b>2. Solver for Computation Domain</b>	<b>3</b>
3.1	ADI Algoritihm . . . . .	3
3.1.1	ADI Discretization . . . . .	3
3.1.2	ADI to Tridiagronal Matrix . . . . .	4
3.1.3	ADI Algorithm . . . . .	5
3.2	Solver Structure . . . . .	5
<b>4</b>	<b>3. Vorticity Contour Result</b>	<b>6</b>
4.1	Result Shown . . . . .	6
4.2	Observation . . . . .	9
4.2.1	Comparsion among time . . . . .	9
4.2.2	Comparsion of different BC . . . . .	9
4.2.3	Comparsion of different grid size . . . . .	9
<b>5</b>	<b>4. Vorticity Contour Result at <math>\mu = 0.001</math></b>	<b>10</b>
5.1	Result shown . . . . .	10
5.2	Viscus effect observation . . . . .	12

5.3	Oscillation Observation and Analysis . . . . .	13
<b>6</b>	<b>5. Accuracy &amp; Wavenumber Analysis</b>	<b>13</b>
6.1	Modified Wavenumber . . . . .	13
6.2	Result Analysis with modified wavenumber . . . . .	14
<b>7</b>	<b>6. Effect of Outflow Boundary–Extended Boundary</b>	<b>14</b>
<b>8</b>	<b>Conclusion</b>	<b>16</b>
	<b>Appendix</b>	<b>17</b>

# 1 Question Review

Consider a Gaussian vortex in a freestream for which the velocity field is given by

$$u = 1 - V_t (y - y_0) \exp\left(\frac{1 - (r/r_0)^2}{2}\right),$$

$$v = V_t (x - x_0) \exp\left(\frac{1 - (r/r_0)^2}{2}\right),$$

where  $V_t = 0.25$ ,  $x_0 = 0.5$ ,  $y_0 = 0.5$ ,  $r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$  and  $r_0 = 0.1$ .

Assume that the convection of this vortex in a flow can be modeled by the viscous Burger's equation:

$$u_t + uu_x + vv_y = \mu(u_{xx} + u_{yy}),$$

$$v_t + uv_x + vv_y = \mu(v_{xx} + v_{yy}),$$

1. Discretize the above equation using a central difference scheme in space, Forward Euler for the convection term and backward Euler for the diffusion term.
2. Solve the above on a computation domain of size  $L_x \times L_y = 2 \times 1$  with  $(u, v) = (1, 0)$  on the left, top and bottom boundaries and two different outflow conditions
  - (a)  $(u, v) = (1, 0)$  on the right boundary
  - (b)  $(u_x, v_x) = (0, 0)$  on the right boundary
3. Examine the convection of the vortex from time  $t = 0$  to 2.0 with  $\mu = 0.01$  by plotting snapshots of vorticity contours at chosen intervals ( $t = 0, 0.5, 1.0, 1.5, 2.0$ ) and examine the effect of grid resolution by employing grids with spacings of  $(\Delta x, \Delta y) = 0.02, 0.01, 0.005$ .
4. Repeat (3) with  $\mu = 0.001$ .
5. Discuss your results with context to the expected accuracy (i.e modified wavenumber) of the scheme.
6. Discuss the effect of the two different outflow boundary conditions on your computer solution. One way to characterize the effect of the outflow boundary condition is to rerun the simulation on a domain that extends beyond  $x = 2$  and then compare your solution to this one.

You are welcome to use any method of choice for solving the discrete equations. Typical choices are ADI coupled with a TDMA solver or Gauss-Seidel to solve the penta-diagonal system that results for the 2D problem. You are welcome to use any solvers/codes you developed earlier in this course or in Numerical Methods. Include a printout of your computer code.

## 2 1. Discretized Equation

$$\underbrace{\frac{\partial U}{\partial t} + u \frac{\partial U}{\partial x} + v \frac{\partial U}{\partial y}}_{\text{convection}} = \underbrace{\mu \left( \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} \right)}_{\text{diffusion}}$$

We discretized this equation using central difference in space, Forward Euler for the convection term and backward Euler for the diffusion term.

For u, the discretized Equation is:

$$\begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} &= \mu \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\ \frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} + u_{i,j}^n \frac{u_{i+1,j}^n - u_{i-1,j}^n}{2\Delta x} + v_{i,j}^n \frac{u_{i,j+1}^n - u_{i,j-1}^n}{2\Delta y} &= \mu \left( \frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j-1}^{n+1}}{\Delta y^2} \right) \end{aligned}$$

Similarly, for v, the discretized Equation is:

$$\begin{aligned} \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} &= \mu \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \\ \frac{v_{i,j}^{n+1} - v_{i,j}^n}{\Delta t} + u_{i,j}^n \frac{v_{i+1,j}^n - v_{i-1,j}^n}{2\Delta x} + v_{i,j}^n \frac{v_{i,j+1}^n - v_{i,j-1}^n}{2\Delta y} &= \mu \left( \frac{v_{i+1,j}^{n+1} - 2v_{i,j}^{n+1} + v_{i-1,j}^{n+1}}{\Delta x^2} + \frac{v_{i,j+1}^{n+1} - 2v_{i,j}^{n+1} + v_{i,j-1}^{n+1}}{\Delta y^2} \right) \end{aligned}$$

## 3 2. Solver for Computation Domain

### 3.1 ADI Algorithm

As our viscous Burger's Equation is showing below:

$$u_t = -(uu_x + vu_y) + \mu(u_{xx} + v_{yy})$$

Shown above, we could discretize this equation using central difference in space, Forward Euler for the convection term, and backward Euler for the diffusion term, could get:

$$\frac{\Delta_t u^n}{\Delta t} = -(u \frac{\delta_x u^n}{\Delta x} + v \frac{\delta_y u^n}{\Delta y}) + \mu \left( \frac{\delta_{xx}^2 u^{n+1}}{\Delta x^2} + \frac{\delta_{yy}^2 u^{n+1}}{\Delta y^2} \right)$$

#### 3.1.1 ADI Discretization

Applying Alternating Direction Implicit (ADI) method, the equation could be consider as two equations:

Form  $n$  to  $n + \frac{1}{2}$ :

$$\frac{u^{n+\frac{1}{2}} - u^n}{\Delta t/2} - \mu \left( \frac{\delta_{xx}^2 u^{n+\frac{1}{2}}}{\Delta x^2} \right) = - \left( u \frac{\delta_x u^n}{\Delta x} + v \frac{\delta_y u^n}{\Delta y} \right) + \mu \left( \frac{\delta_{yy}^2 u^n}{\Delta y^2} \right)$$

Form  $n + \frac{1}{2}$  to  $n + 1$ :

$$\frac{u^{n+1} - u^{n+\frac{1}{2}}}{\Delta t/2} - \mu \left( \frac{\delta_{yy}^2 u^{n+1}}{\Delta y^2} \right) = - \left( u \frac{\delta_x u^{n+\frac{1}{2}}}{\Delta x} + v \frac{\delta_y u^{n+\frac{1}{2}}}{\Delta y} \right) + \mu \left( \frac{\delta_{xx}^2 u^{n+\frac{1}{2}}}{\Delta x^2} \right)$$

### 3.1.2 ADI to Tridiagonal Matrix

Summarize our discretization of the equation:

(1) Form  $n$  to  $n + \frac{1}{2}$ :

$$\left[ \frac{1}{\frac{\Delta t}{2}} - \mu \frac{\delta_{xx}^2}{\Delta x^2} \right] u^{n+\frac{1}{2}} = \left[ \frac{1}{\frac{\Delta t}{2}} - u^n \frac{\delta_x}{\Delta x} - v^n \frac{\delta_y}{\Delta y} + \mu \frac{\delta_{yy}^2}{\Delta y^2} \right] u^n$$

Could get both sides:

$$\left\{ \begin{array}{l} \left[ \frac{1}{\frac{\Delta t}{2}} - \mu \frac{\delta_{xx}^2}{\Delta x^2} \right] (i, j) = \underbrace{-\frac{\mu}{\Delta x^2} [(i-1, j) + (i+1, j)]}_{a,c} + \underbrace{\left[ \frac{1}{\frac{\Delta t}{2}} + \frac{\mu}{\Delta x^2} \right]}_b (i, j) \\ D(i) = (i, j) \frac{1}{\frac{\Delta t}{2}} - u(i, j) \left[ \frac{(i+1, j) - (i-1, j)}{2\Delta x} \right] - v(i, j) \left[ \frac{(i, j+1) - (i, j-1)}{2\Delta y} \right] + \mu \left[ \frac{(i, j+1) + (i, j-1) - 2(i, j)}{\Delta y^2} \right] \end{array} \right.$$

Where, (i,j) means the variable we are going to get, a,b,c,D are vectors form tri-diagonal matrix, we solve it using Thomas Algorithm (TDMA) method.

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ a & b & c & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & a & b & c \\ 0 & \dots & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} U_0 \\ U_1 \\ \vdots \\ U_{n-1} \\ U_n \end{pmatrix} = \begin{pmatrix} U_0 \\ D \\ \vdots \\ D \\ U_n \end{pmatrix}$$

(2) Form  $n + \frac{1}{2}$  to  $n + 1$ :

$$\left[ \frac{1}{\Delta t} - \mu \frac{\delta_{yy}^2}{\Delta y^2} \right] u^{n+1} = \left[ \frac{1}{\frac{\Delta t}{2}} - u^{n+\frac{1}{2}} \frac{\delta_x}{\Delta x} - v^n \frac{\delta_y}{\Delta y} + \mu \frac{\delta_{xx}^2}{\Delta x^2} \right] u^{n+\frac{1}{2}}$$

Where, (i,j) means the variable we are going to get, a,b,c,d are vectors get into TDMA.

$$\left\{ \begin{array}{l} \left[ \frac{1}{\frac{\Delta t}{2}} - \mu \frac{\delta_{yy}^2}{\Delta y^2} \right] (i, j) = \underbrace{-\frac{\mu}{\Delta y^2} [(i-1, j) + (i+1, j)]}_{a,c} + \underbrace{\left[ \frac{1}{\frac{\Delta t}{2}} + \frac{\mu}{\Delta y^2} \right]}_b (i, j) \\ D(j) = (i, j) \frac{1}{\frac{\Delta t}{2}} - u(i, j) \left[ \frac{(i+1, j) - (i-1, j)}{2\Delta x} \right] - v(i, j) \left[ \frac{(i, j+1) - (i, j-1)}{2\Delta y} \right] + \mu \left[ \frac{(i, j+1) + (i, j-1) - 2(i, j)}{\Delta x^2} \right] \end{array} \right.$$

Similarly, a, b, c, D perform tri-diagonal matrix as same arrangement as before, where only a,b,c and D are different

### 3.1.3 ADI Algorithm

#### Algorithm 1 Alternating Direction Implicit (ADI) Method

```

1: for timestep =  $n = 0$  to  $N - 1$  do                                ▶ Loop over time steps
2:   // Step from  $n$  to  $n + \frac{1}{2}$ 
3:   for each each  $j$  lines do                                       ▶ Loop over each line
4:      $a=b=c=(i_{max}-1) \times 1$  space,  $a[i] = c[i] = a$ ,  $b[i] = b$ 
5:      $d=(i_{max}+1) \times 1$  space,  $d[i]$ 
6:      $a=c=[0] + a + [0]$ ,  $b=[1] + b + [1]$ ,  $d[0]=U[0, j]$ ,  $d[i_{max}] = U[i_{max}, j]$ 
7:      $U_{half}[i, j] = \text{TDMA}(a,b,c,d)$ 
8:   end for
9:   // Step from  $n + \frac{1}{2}$  to  $n + 1$ 
10:  for each each  $i$  lines do                                         ▶ Loop over each column
11:     $a=b=c=(j_{max}-1) \times 1$  space,  $a[j] = c[j] = a$ ,  $b[j] = b$ 
12:     $d=(j_{max}+1) \times 1$  space,  $d[j]$ 
13:     $a=c=[0] + a + [0]$ ,  $b=[1] + b + [1]$ ,  $d[0]=U_{half}[i, 0]$ ,  $d[j_{max}] = U_{half}[i, j_{max}]$ 
14:     $U_{new}[i, j] = \text{TDMA}(a,b,c,d)$ 
15:  end for
16:   $U_{new} = \text{BC}(U_{new})$ ,  $U = U_{new}$ 
17: end for

```

### 3.2 Solver Structure

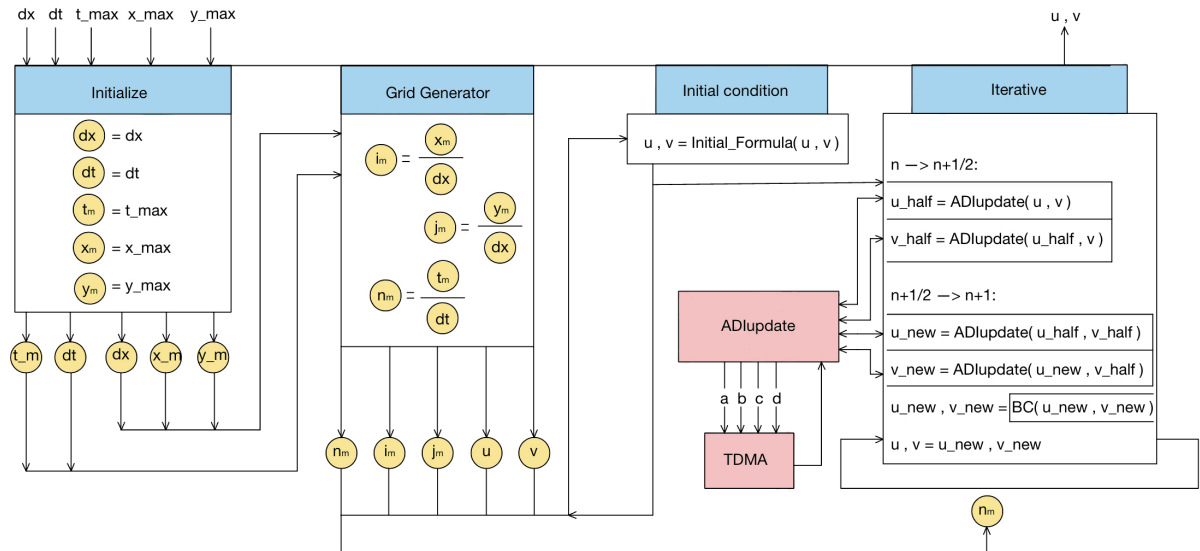


Figure 1: Solver Structure

Our solver structure is showing above, where the variable in the circle (yellow) means private, and the private function (blue) contains major solve steps.

## 4 3. Vorticity Contour Result

### 4.1 Result Shown

Using  $\Delta t = 0.001$ , which let the scheme satisfy stability condition, we get the result for grid size= 0.02 is showing below:

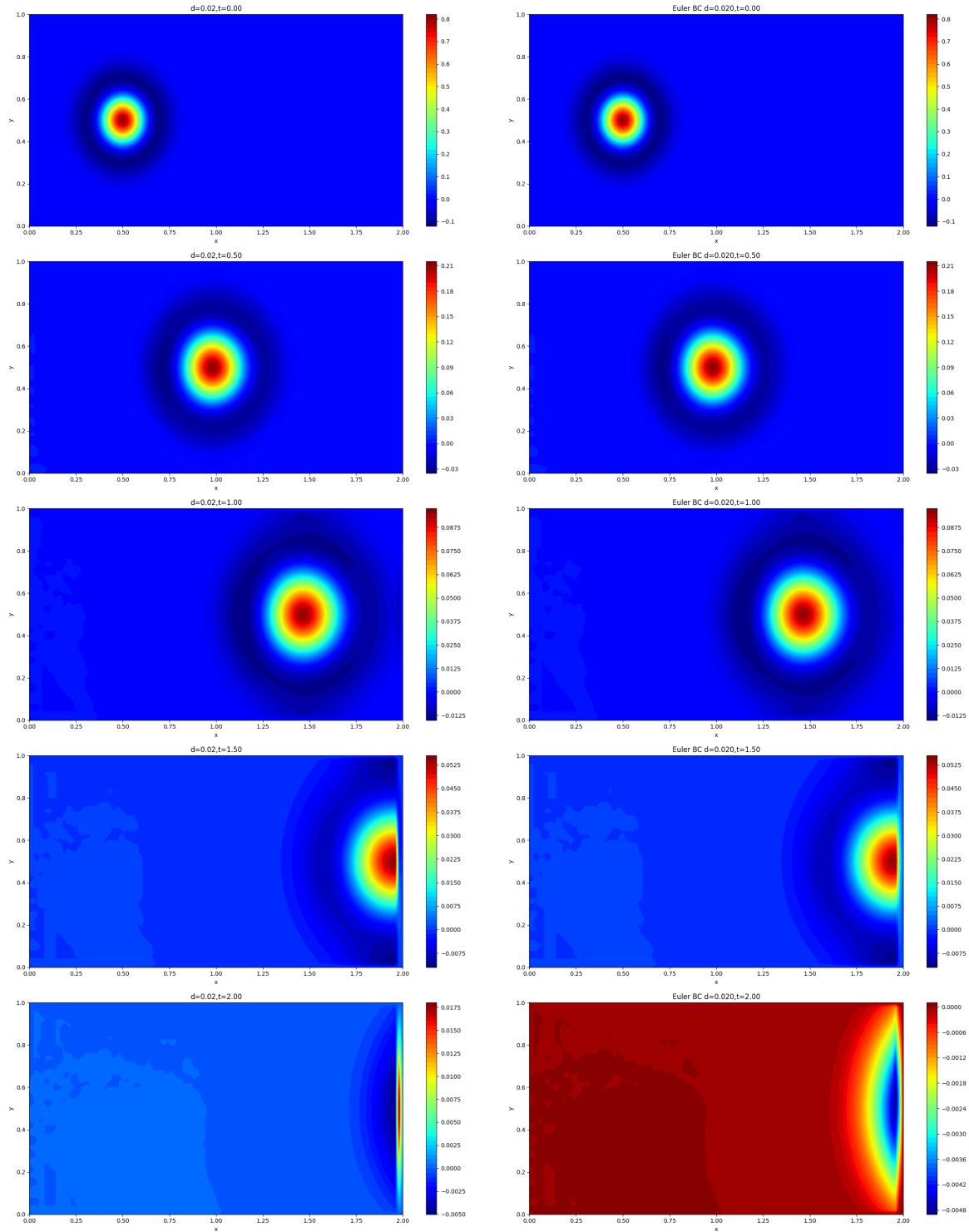


Figure 2: Grid size 0.02, dirichlet BC vs Neumann BC

Left is the Dirichlet boundary condition(Boundary condition a), the right side is Neumann boundary condition, time 0, 0.5, 1.0, 1.5, 2.0.

For grid size 0.02, could see the vortex is moving from left to right, and touch the right boundary at  $t=1.5$ . By compare the two boundary conditions, we could notice for Neumann boundary condition, the boundary effect on vortex is much smaller than the result in Dirichlet boundary condition. The same observation on the finer grids:

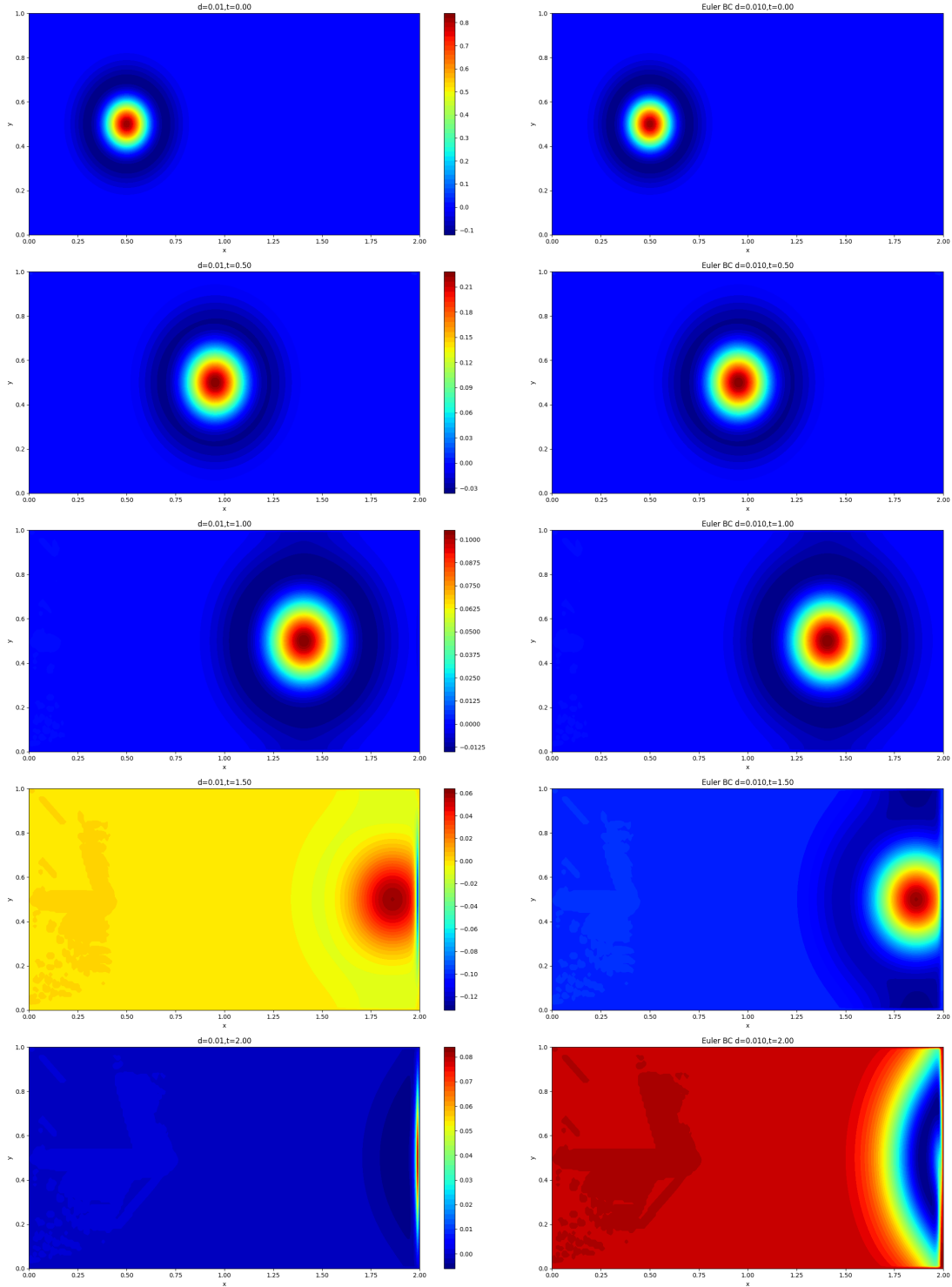


Figure 3: Grid size 0.01, dirichlet BC vs Neumann BC

For the finest grid  $\Delta = 0.005$ , the result is much more accurate than the others, where we could see the vortex center's value (peak value) is larger than the other grids, and the vortex influence region is smaller than other grids at the same time point.

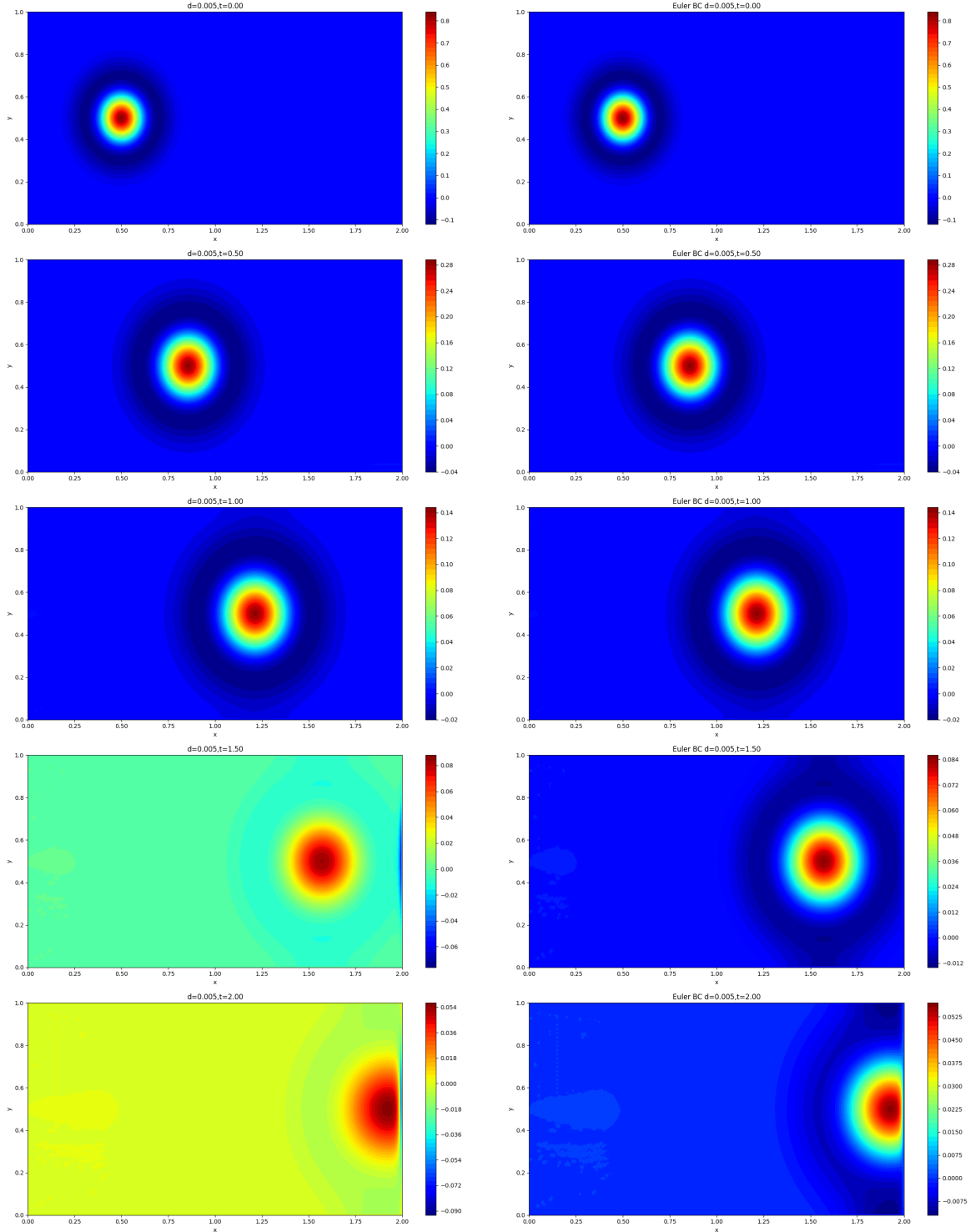


Figure 4: Grid size 0.005, dirichlet BC vs Neumann BC



## 4.2 Observation

### 4.2.1 Comparson among time

Based on the result among different time, we could observe the vortex moving from left to right, while the vortex effective range is getting bigger and bigger, and the amplitude of the vortex center is decreasing.

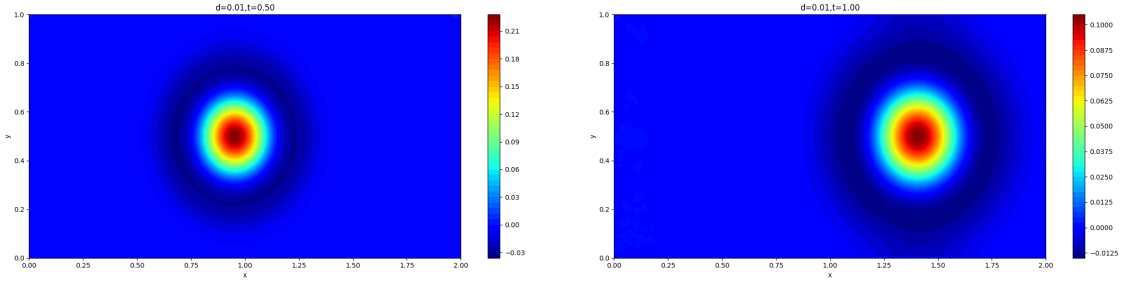


Figure 5: Comparson of  $t=0.5$  and  $t=1$ , grid  $d=0.01$

### 4.2.2 Comparson of different BC

We have examined two different boundary condition (BC), one is Dirichlet boundary condition, where its right boundary its been artificially set to  $u = 1, v = 0$ , and Neumann boundary condition, where at the boundary, the  $u$  and  $v$  gradience normal to the boundary is zero.

By compare the left (Dirichlet BC) and right (Neumann BC), we could see the boundary effection on the vortex at Neumann BC is smaller than Dirichlet BC.

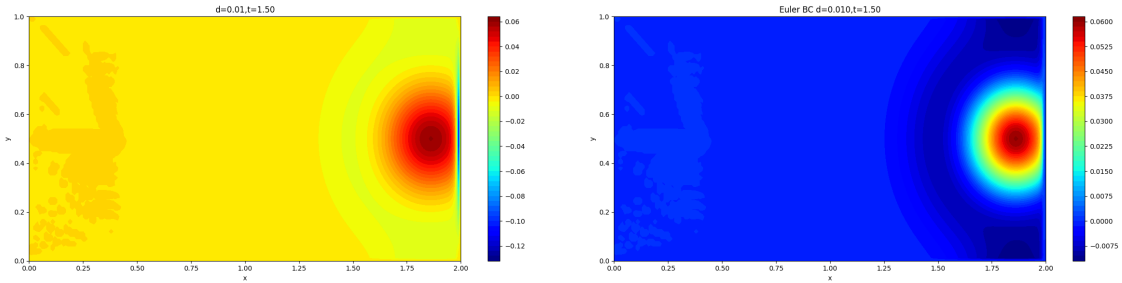


Figure 6: Comparson of Dirichlet BC and Neumann BC at  $t=1.5$

### 4.2.3 Comparson of different grid size

As showing above, we have examined the grid size at 0.02, 0.01, and 0.005. Compare different grid size's result, could find the finer grid as, the slower the diffusion speed it get.

## 5 4. Vorticity Contour Result at $\mu = 0.001$

### 5.1 Result shown

Changing  $\mu$  from 0.01 to 0.001, the result in different size grid is showing below:

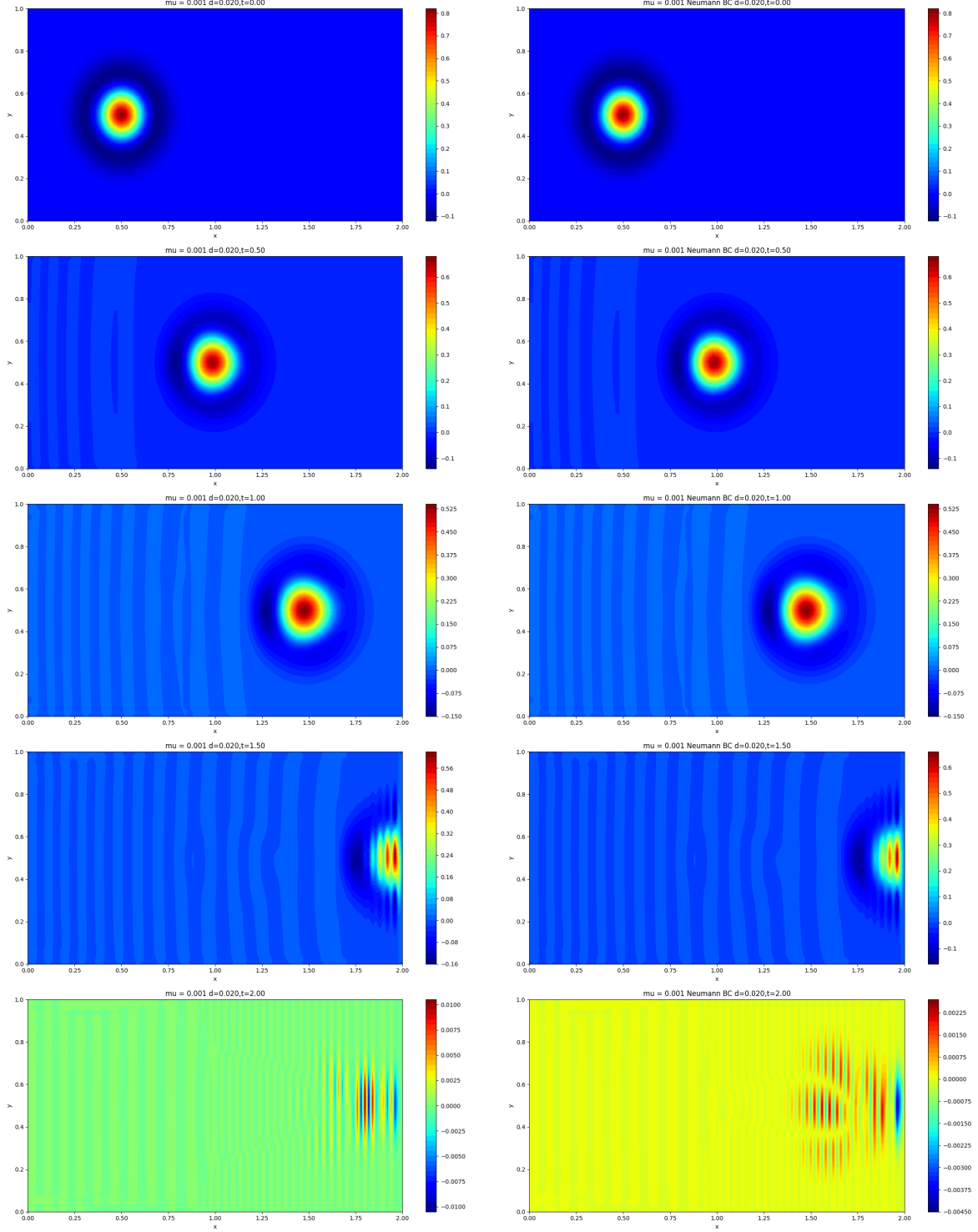


Figure 7: Grid size 0.020,  $\mu = 0.001$ , dirichlet BC vs Neumann BC

We could find the vortex is much smaller than  $\mu = 0.01$  result, which this  $\mu = 0.001$ , let

diffusion speed is slower.

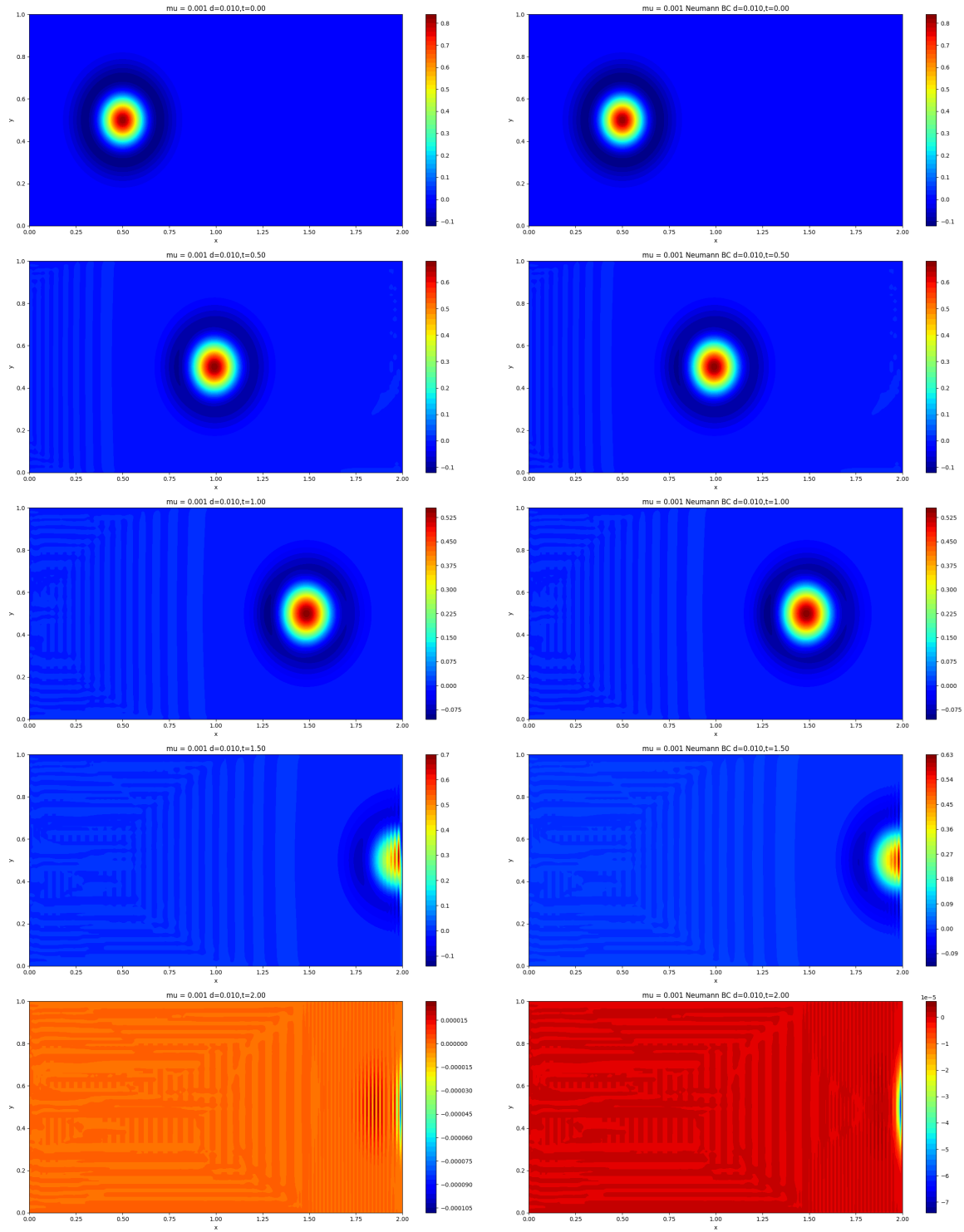


Figure 8: Grid size 0.010,  $\mu = 0.001$ , dirichlet BC vs Neumann BC

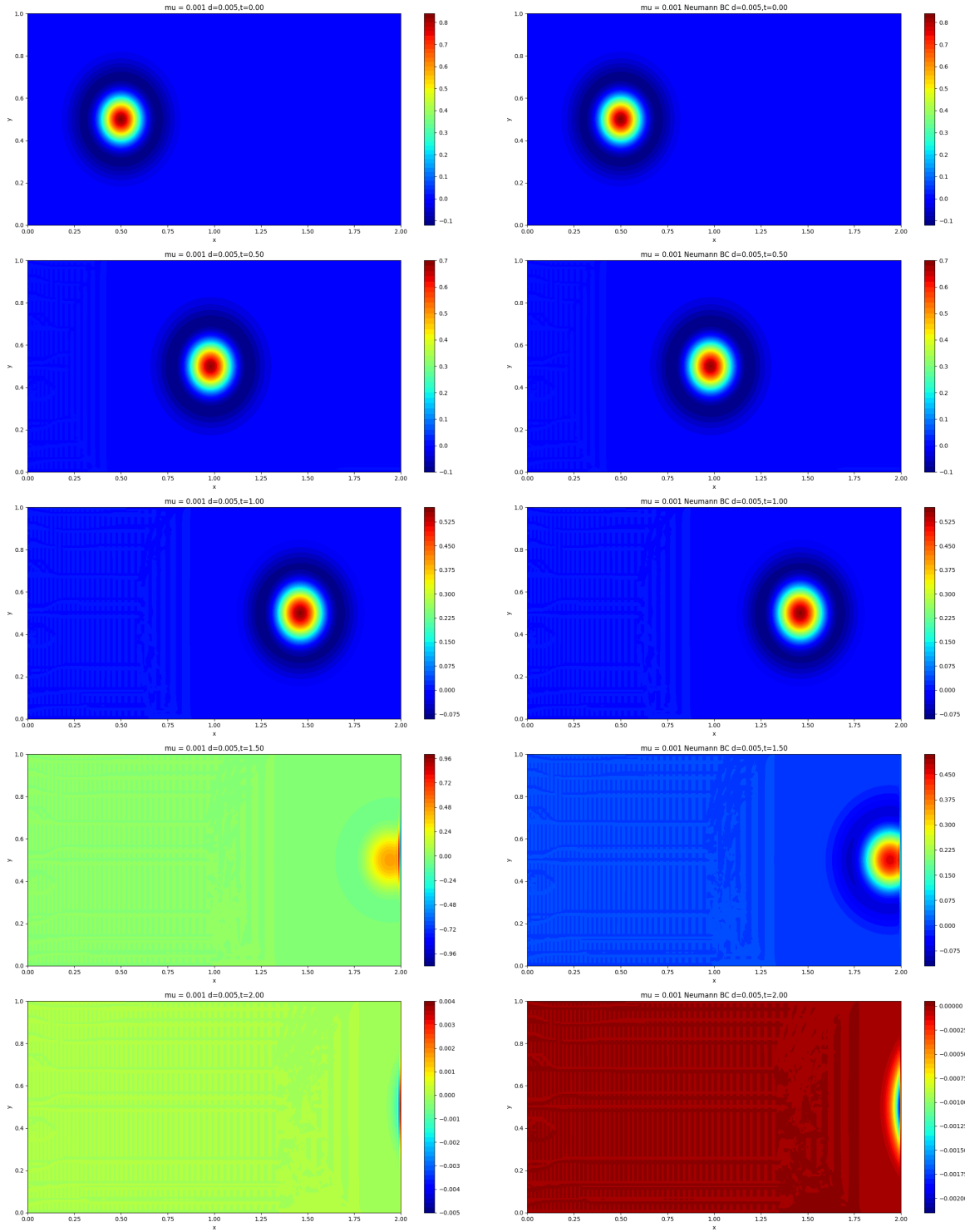


Figure 9: Grid size 0.005,  $\mu = 0.001$ , dirichlet BC vs Neumann BC

## 5.2 Viscus effect observation

Compare our result in  $\mu = 0.001$  and  $\mu = 0.01$  (Pervious section), we could easily observed the vortex at  $\mu = 0.001$  is diffused much slower than the pervious section result. In more detailed, is with time going, the vortex's effect range is gowing slower, and the vortex center (peak) value remains larger than the  $\mu = 0.01$  case.

### 5.3 Oscillation Observation and Analysis

As we observed oscillation wave in the result of  $\mu = 0.001$ , we also observed the oscillation effect shows more obvious on the coarse grid, it do not show obviously at the grid size = 0.005.

As we have done in HW2, we use  $Pe_{\Delta x}$  to analysis why this effect could happen:

$Pe_{\Delta x}$	$\mu = 0.01$	$\mu = 0.001$
$\Delta x = 0.02$	2	20
$\Delta x = 0.01$	1	10
$\Delta x = 0.005$	0.5	5

Table 1:  $Pe_{\Delta x}$  at  $U = 1$  Comparison

We know as the  $Pe_{\Delta x}$  is pretty large, it could cause negative term which led to oscillation. This is not means the scheme is unstable, but the effect is unstable.

Compare the formal case where  $\mu = 0.01$ , we could find in this condition, the finer grid is, the low  $Pe_{\Delta x}$  be, and the  $Pe_{\Delta x}$  is much smaller at  $\mu = 0.01$ , and become much bigger at  $\mu = 0.001$ , where it cause oscillation, could correspond to our case result.

## 6 5. Accuracy & Wavenumber Analysis

### 6.1 Modified Wavenumber

As modified wavenumber analysis:

$$u = u^{ik(x+y)}$$

Could get

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial y} = iku$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial y^2} = -k^2 u$$

1st Central difference:

$$\frac{\delta_x u}{\Delta x} = u \frac{u^{ik\Delta x} - u^{-ik\Delta x}}{2\Delta x} = iu \frac{\sin k\Delta x}{\Delta x}$$

where,

$$k'_x = \frac{\sin k\Delta x}{\Delta x}$$

2nd Central difference:

$$\frac{\delta_x^2 u}{\Delta x^2} = u \frac{u^{ik\Delta x} + u^{-ik\Delta x} - 2}{\Delta x^2} = u \frac{2(\cos k\Delta x - 1)}{\Delta x^2}$$

Could get

$$k'_{xx} = \sqrt{\frac{2(1 - \cos k\Delta x)}{\Delta x^2}}$$

The figure compare modified wavenumber and exact wavenumber is showing below:

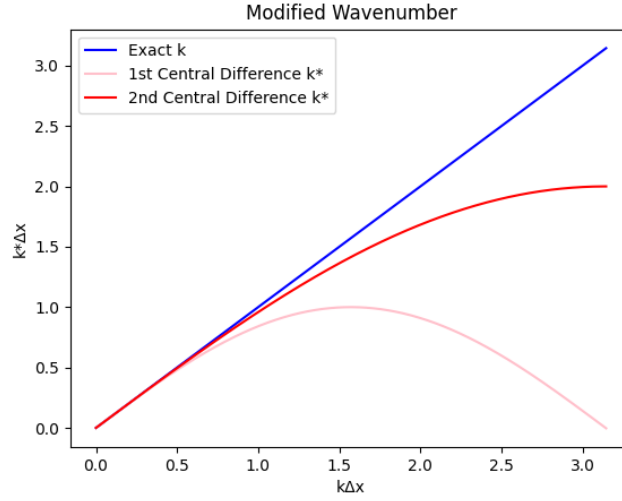


Figure 10: 1st and 2nd CD Modified Wavenumber Compare to Exact

## 6.2 Result Analysis with modified wavenumber

From the figure of modified wavenumber, we could as find the  $\Delta$  getting small as we using finer grid, for each  $k$ , the  $k\Delta x$  is moving to the left, which comming to the region that the scheme's modified wavenumber is closer to the exact wavenumber, resulted that when we using finer grid, the result become more accurate.

As the Central difference scheme's modified wavenumber only contains the real part, as the real part's difference make the dispersion error, in this case specific, it make the vortex's wave travel faster in the same time period. More specifically, as the grid size ( $\Delta$ ) larger, the more diffused effect shown in the vortex, the smaller the peak values in the same time period.

This could explain the diffusion result shown in our result: the finer the grid become, the diffusion speed comes more slower, maitain the accuracy of the diffusion result.

## 7 6. Effect of Outflow Boundary–Extended Boundary

To examine the effect of the outflow condition, we use extended domain:  $x_{max}=3$ .

The comparsion at  $t=1.5$ ,  $d=0.02$  is showing below:

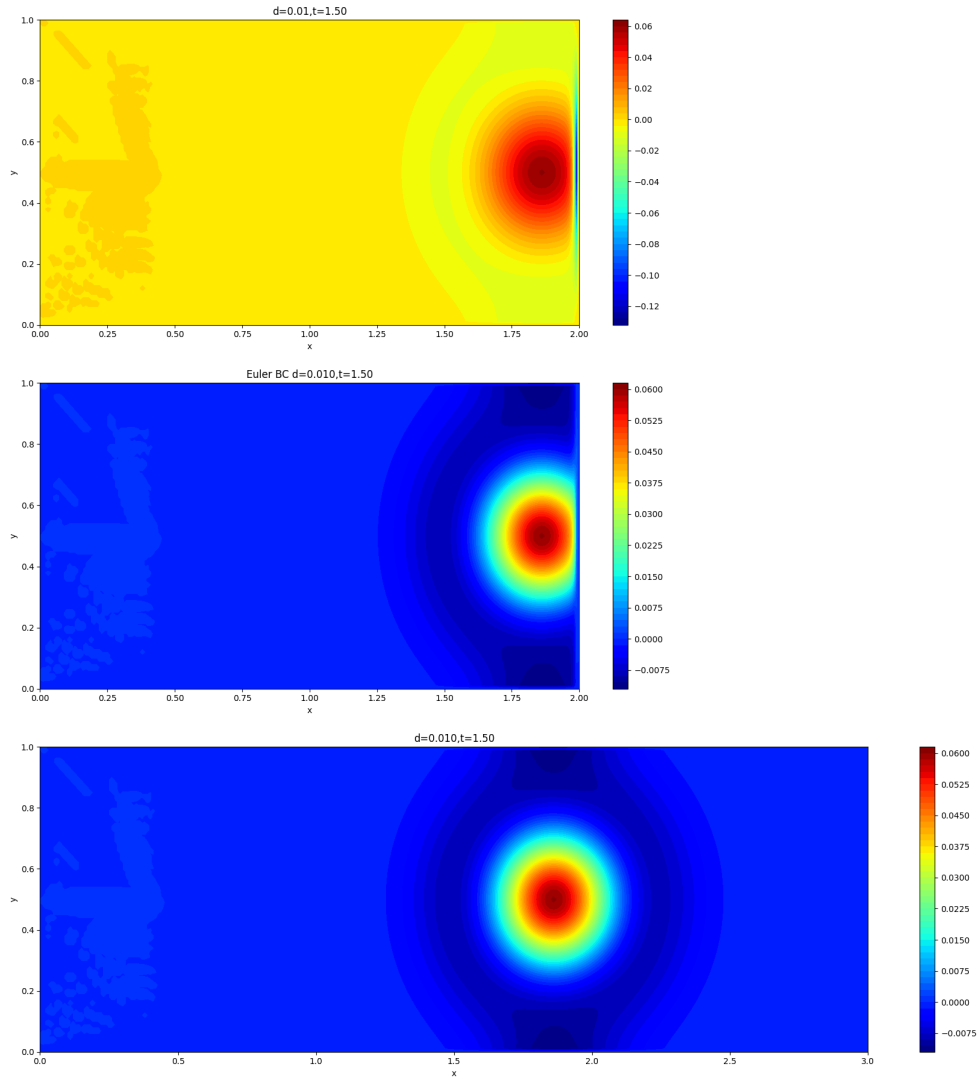


Figure 11: Comparson of Dirichlet BC reuslt, Neumann BC result, and Extended outflow result

We could find that the extended domain result is much similiar to the Neumann Boundary condition, where the vortex do not been enormously impacted by the artifically boundary.

For more specific analysis, the Dirichlet boundary condition artifically set the outflow condition on the boundary is not effected by the flow condition inside, which in this case is our vortex. In Neumann boundary condition, it sets the boundary is effected by the flow condition inside the domain, which could more fit the vortex propagation without artifically boundary effect.

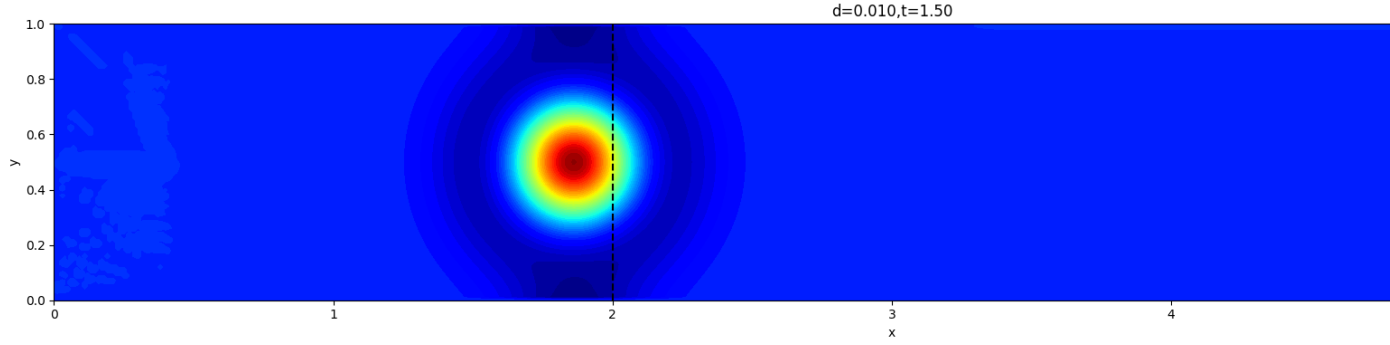


Figure 12: Extended outflow to  $x=6$  result

We also examined much more larger domain ( $x_{max} = 6$ ), could also show is much similar to the Neumann boundary condition, where the boundary do not have much artificially effect to the flow condition inside, where makes it could simulate kind of free-flow condition

## 8 Conclusion

In this project, we discretized the viscous Burgers' equation using the Central Difference scheme for spatial differentiation, the Forward Euler for the convection term, and the Backward Euler for the diffusion term to handle temporal derivatives. We then employed ADI solver to obtain a tridiagonal matrix, which updated by lines at first half time step and update by columns at the second time step. We then implemented TDMA to solve the tridiagonal matrix and derive the results.

We investigated the effects of grid size, boundary conditions, and viscosity on the outcomes, also the effect of the  $Pe_{\Delta x}$ . Our findings indicate that as the grid becomes finer, diffusion tends to be slower and less pronounced, mirroring the impact of reduced viscosity. However,  $Pe_{\Delta x}$  too large could also cause oscillation, where shown in the result of  $\mu = 0.001$ , the large the grid size we set, the smaller  $\mu$  we use, the large  $Pe_{\Delta x}$  we get, and the large oscillation we obtained.

We also compared the effects of Dirichlet and Neumann boundary conditions where Neumann boundary condition shows more close to the accurate result as the interior flow do not hugely effected by the artificially limit of the domain like Dirichlet boundary condition.



## Appendix

```
1
2 import numpy as np #use it in all class u,v
3
4 import math # use it in Init to get initial value
5 import copy # use it in TDMA and Update to avoid influence original
  data
6
7 import matplotlib.pyplot as plt # use it to draw vorticity contour
8
9
10
11 def TDMA(a, b, c, d):
12     # TDMA solver
13     # b: main diagonal a: down diagonal c: up diagonal
14     # lenth of a , b, c, d is as same as D, with no use of a[0],
  a[max], c[0], c[max]
15
16     # | b0  c0  0  ...  0  |
17     # | a1  b1  c1  ...  0  |
18     # | 0  a2  b2  ...  0  |
19     # | ...  ...  ...  ...  ...  |
20     # | 0  ...  an-2  bn-2  cn-2|
21     # | 0  ...  0  an-1  bn-1|
22
23     import copy
24     do = copy.deepcopy(d)
25     ao = copy.deepcopy(a)
26     bo = copy.deepcopy(b)
27     co = copy.deepcopy(c)
28     N = len(d)
29     xo = np.zeros(N)
30
31     for rowi in range(1,N):
32         k = ao[rowi]/bo[rowi-1]
33         bo[rowi] -= co[rowi-1]*k
34         do[rowi] -= do[rowi-1]*k
35
36     xo[N-1] = do[N-1]/bo[N-1]
37     for rowi in range(N-2,-1,-1):
38         xo[rowi] = (do[rowi]-co[rowi]*xo[rowi+1])/bo[rowi]
39
40     return xo
41
42
43
44 class ADI_Solver:
45     def __init__(self, x_max, y_max, t_max, dx, dy, dt, mu):
46         self.x_max = x_max
47         self.y_max = y_max
48         self.t_max = t_max
49         self.dx = dx
50         self.dy = dy
51         self.dt = dt
52         self.mu = mu
53
54     def Grid_Generate(self):
```

```

55         self.i_max = int(self.x_max/self.dx)
56         self.j_max = int(self.y_max/self.dy)
57         self.n_max = int(self.t_max/self.dt)
58
59         self.u = np.zeros([ self.i_max+1 , self.j_max+1 ], dtype =
float)
60         self.v = np.zeros([ self.i_max+1 , self.j_max+1 ], dtype =
float)
61
62
63     def Initialize(self):
64         for i in range(0, self.i_max+1):
65             for j in range(0, self.j_max+1):
66                 self.u[i][j], self.v[i][j] = self.initial_formula(i, j)
67
68         # print(self.u) ## TESTING
69
70     def initial_formula(self, i, j):
71         V_t = 0.25
72         x_0 = 0.5
73         y_0 = 0.5
74         r_0 = 0.1
75
76         r2 = (i*self.dx-x_0)**2+(j*self.dy-y_0)**2
77
78         u_init = 1 - V_t * (j*self.dy-y_0) * math.exp( (1-( r2 )/(r_0
**2)) / (2) )
79         v_init = V_t * (i*self.dx-x_0) * math.exp( (1-( r2 )/(r_0**2))
/ (2) )
80
81         return u_init , v_init
82
83
84     def BC(self, u, v):
85
86         for i in range(0, self.i_max+1):
87             u[i][0] = 1
88             v[i][0] = 0
89             u[i][-1] = 1
90             v[i][-1] = 0
91
92         for j in range(0, self.j_max+1):
93             u[0][j] = 1
94             v[0][j] = 0
95             u[-1][j] = 1
96             v[-1][j] = 0
97             # u[self.i_max-1][j], v[self.i_max-1][j] = u[self.i_max][j
], v[self.i_max][j]
98         return u, v
99
100
101     def Iterative(self):
102         self.u, self.v = self.BC(self.u, self.v)
103
104         for n in range(0, self.n_max):
105             self.u, self.v = self.BC(self.u, self.v)
106
107         u_new, v_new = copy.deepcopy(self.u), copy.deepcopy(self.v)

```

```

108         u_new, v_new = self.K_iteration(self.u, self.v)
109
110
111         self.u, self.v = u_new, v_new
112         print(n)
113
114     return self.u, self.v
115
116
117
118
119
120
121
122     def K_iteration(self, u, v):
123
124         u_half, v_half = copy.deepcopy(u), copy.deepcopy(v)
125         u_new, v_new = copy.deepcopy(u), copy.deepcopy(v)
126
127
128         # update u:
129         # u{n} to u{n+1/2} line update:
130         for j in range(1, self.j_max): # flag=flag(0/1,0/1), 1:donig
0:other
131             flag = (0,1)
132             u_half[:,j] = self.UPDATE( u , v, u, j, self.i_max, flag)
133             # Update function: Update(u, v, doing(u/v), doing(row=j
/column=i), doingAnother, flag)
134
135
136
137         u_new, v_new = self.BC(u_new, v_new)
138         u_half, v_half = self.BC(u_half, v_half )
139
140
141         # update v:
142         for j in range(1, self.j_max): # flag=flag(0/1,0/1), 1:donig
0:other
143             flag = (0,1)
144             v_half[:,j] = self.UPDATE( u_half , v, v, j, self.i_max,
flag)
145
146
147         u_new, v_new = self.BC(u_new, v_new)
148         u_half, v_half = self.BC(u_half, v_half )
149
150
151         # u{n+1/2} to u{n+1} column update:
152         for i in range(1, self.i_max-1):
153             flag = (1,0)
154             u_new[i,:] = self.UPDATE( u_half , v_half, u_half, i, self.
j_max, flag)
155
156
157         u_new, v_new = self.BC(u_new, v_new)
158         u_half, v_half = self.BC(u_half, v_half )
159
160

```

```

161         for i in range(1, self.i_max):
162             flag = (1, 0)
163             v_new[i, :] = self.UPDATE( u_new , v_half , v_half , i , self.
164 j_max, flag)
165
166
167         u_new, v_new = self.BC(u_new, v_new)
168         u_half, v_half = self.BC(u_half, v_half )
169
170         return u_new , v_new
171
172
173     def UPDATE(self, u, v, U, Idoing, Ianother_max, flag):
174
175         delta = self.dx * flag[1] + self.dy*flag[0]
176
177
178         r = (self.mu*self.dt)/(2*delta**2)
179
180
181         # Creating a, b, c, D
182         a = np.full(Ianother_max-1, -r, dtype = float)
183         b = np.full(Ianother_max-1, 1 + 4*r, dtype = float)
184         c = np.full(Ianother_max-1, -r, dtype = float)
185         d = np.full(Ianother_max+1, 0, dtype = float)
186
187         for ij in range(1, Ianother_max):
188             i, j = Idoing*flag[0] + ij* flag[1] , Idoing*flag[1] + ij*
189 flag[0]
190
191             cox = (u[i, j]*self.dt/self.dx /2)
192             coy = (v[i, j]*self.dt/self.dy /2)
193             rx = (self.mu*self.dt/(2*self.dx**2))
194             ry = (self.mu*self.dt/(2*self.dy**2))
195             rxy = rx * flag[1] + ry*flag[0]
196             d[ij] = U[i, j]-cox*((U[i+1, j]-U[i-1, j])/(2))-coy*((U[i, j
197 +1]-U[i, j-1])/(2))+rxy*(U[i+flag[0], j+flag[1]]+U[i-flag[0], j-flag[1]])
198             d[0], d[-1] = U[i*flag[0], j*flag[1]], U[i*flag[0]-1, j*flag
199 [1]-1]
200
201
202         # added boundary get full matrix:
203         a = np.concatenate((np.array([0]), a, np.array([0])), axis=0)
204         b = np.concatenate((np.array([1]), b, np.array([1])), axis=0)
205         c = np.concatenate((np.array([0]), c, np.array([0])), axis=0)
206
207
208         u_UPDATE = TDMA(a, b, c, d)
209
210
211         # | 1    0    0    ...    0    0    |
212         # | a1  b1  c1    ...    0    0    |
213         # | 0   a2  b2    ...    0    0    |
214         # | ...  ...  ...    ...    ...  ...  |
215         # | 0   0    0    ...    b(n-2) c(n-2)|
216         # | 0   0    0    ...    0    1    |
217
218         # print("TDMA")

```

```

215         # print(u_UPDATE)
216
217         return u_UPDATE
218
219
220
221
222
223
224 def Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu):
225
226     Try_1 = ADI_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
227     Try_1.Grid_Generate()
228     Try_1.Initialize()
229     u, v =Try_1.Iterative()
230
231     return u,v
232
233
234 def Vorticity(u, v, t):
235     # u,v = np.transpose(u), np.transpose(v)
236
237     Ni, Nj = u.shape
238
239     x = np.linspace(0, 2, Ni)
240     y = np.linspace(0, 1, Nj)
241
242     X, Y = np.meshgrid(x, y)
243
244     omega = np.zeros((Ni, Nj))
245     # omega[1:-1, 1:-1] = (v[1:-1, 2:] - v[1:-1, :-2]) / (x[2] - x[1])
246     - (u[2:, 1:-1] - u[:-2, 1:-1]) / (y[2] - y[1])
247
248     for i in range(1,Ni-1):
249         for j in range(1,Nj-1):
250             omega[i,j] = (v[i+1,j] - v[i-1,j]) / (x[2]-x[0]) - (u[i,j+1]
251             - u[i,j-1]) / (y[2] - y[0])
252
253     omega = np.transpose(omega)
254
255     d = x[2]-x[1]
256
257     print(d)
258
259     plt.figure(figsize=(12, 6))
260     plt.contourf(X, Y, omega, levels=50, cmap='jet')
261     plt.colorbar()
262     plt.title('d={:.3 f}, t={:.2 f}'.format(d, t))
263     plt.xlabel('x')
264     plt.ylabel('y')
265
266
267     plt.tight_layout()
268     plt.savefig('3d {:.3 f} t {:.2 f}.png'.format(d, t))
269     # plt.show()
270

```

```

271
272
273
274 def main():
275     x_max = 2
276     y_max = 1
277     t_max = 2
278
279     # dx = 0.02
280     # dy = 0.01
281     # dt = 0.005
282
283
284     dx = 0.005
285     dy = 0.005
286     dt = 0.001
287
288
289     mu = 0.01
290     # mu = 1
291
292
293     t_max = 0
294     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
295     # print(u)
296     Vorticity(u, v, t_max)
297
298     t_max = 0.5
299     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
300     # print(u)
301     Vorticity(u, v, t_max)
302
303     t_max = 1
304     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
305     # print(u)
306     Vorticity(u, v, t_max)
307
308     t_max = 1.5
309     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
310     # print(u)
311     Vorticity(u, v, t_max)
312
313     t_max = 2
314     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
315     # print(u)
316     Vorticity(u, v, t_max)
317
318
319
320 if __name__ == '__main__':
321     main()
322     # arg1 = [2,2,2,2,2]
323     # arg2 = [1,1,1,1,1]
324     # arg3 = arg1
325     # arg4 = [1,2,3,4,5]
326     # print(TDMA(arg1 ,arg2 ,arg3 ,arg4))

```

Listing 1: ADI Solver

```

1
2
3 import numpy as np #use it in all class u,v
4
5 import math # use it in Init to get initial value
6 import copy # use it in TDMA and Update to avoid influence original
  data
7
8 import matplotlib.pyplot as plt # use it to draw vorticity contour
9
10
11
12 def TDMA(a, b, c, d):
13     # TDMA solver
14     # b: main diagonal a: down diagonal c: up diagonal
15     # lenth of a , b, c, d is as same as D, with no use of a[0],
    a[max], c[0], c[max]
16
17     # | b0  c0  0  ...  0  |
18     # | a1  b1  c1  ...  0  |
19     # | 0  a2  b2  ...  0  |
20     # | ...  ...  ...  ...  ...  |
21     # | 0  ...  an-2  bn-2  cn-2|
22     # | 0  ...  0  an-1  bn-1|
23
24     import copy
25     do = copy.deepcopy(d)
26     ao = copy.deepcopy(a)
27     bo = copy.deepcopy(b)
28     co = copy.deepcopy(c)
29     N = len(d)
30     xo = np.zeros(N)
31
32     for rowi in range(1,N):
33         k = ao[rowi]/bo[rowi-1]
34         bo[rowi] -= co[rowi-1]*k
35         do[rowi] -= do[rowi-1]*k
36
37     xo[N-1] = do[N-1]/bo[N-1]
38     for rowi in range(N-2,-1,-1):
39         xo[rowi] = (do[rowi]-co[rowi]*xo[rowi+1])/bo[rowi]
40
41     return xo
42
43
44
45 class ADI_Solver:
46     def __init__(self, x_max, y_max, t_max, dx, dy, dt, mu):
47         self.x_max = x_max
48         self.y_max = y_max
49         self.t_max = t_max
50         self.dx = dx
51         self.dy = dy
52         self.dt = dt
53         self.mu = mu
54
55     def Grid_Generate(self):
56         self.i_max = int(self.x_max/self.dx)

```

```

57         self.j_max = int(self.y_max/self.dy)
58         self.n_max = int(self.t_max/self.dt)
59
60         self.u = np.zeros([ self.i_max+1 , self.j_max+1 ], dtype =
float)
61         self.v = np.zeros([ self.i_max+1 , self.j_max+1 ], dtype =
float)
62
63
64     def Initialize(self):
65         for i in range(0, self.i_max+1):
66             for j in range(0, self.j_max+1):
67                 self.u[i][j], self.v[i][j] = self.initial_formula(i, j)
68
69         # print(self.u) ## TESTING
70
71     def initial_formula(self, i, j):
72         V_t = 0.25
73         x_0 = 0.5
74         y_0 = 0.5
75         r_0 = 0.1
76
77         r2 = (i*self.dx-x_0)**2+(j*self.dy-y_0)**2
78
79         u_init = 1 - V_t * (j*self.dy-y_0) * math.exp( (1-( r2 )/(r_0
**2)) / (2) )
80         v_init = V_t * (i*self.dx-x_0) * math.exp( (1-( r2 )/(r_0**2))
/ (2) )
81
82         return u_init , v_init
83
84
85     def BC(self, u, v):
86
87         for i in range(0, self.i_max+1):
88             u[i][0] = 1
89             v[i][0] = 0
90             u[i][-1] = 1
91             v[i][-1] = 0
92
93         for j in range(0, self.j_max+1):
94             u[0][j] = 1
95             v[0][j] = 0
96             u[-1][j] = 1
97             v[-1][j] = 0
98             # u[self.i_max-1][j], v[self.i_max-1][j] = u[self.i_max][j
], v[self.i_max][j]
99         return u, v
100
101
102     def Iterative(self):
103         self.u, self.v = self.BC(self.u, self.v)
104
105         for n in range(0, self.n_max):
106             self.u, self.v = self.BC(self.u, self.v)
107
108             u_new, v_new = copy.deepcopy(self.u), copy.deepcopy(self.v)
109

```



```

110         u_new, v_new = self.K_iteration(self.u, self.v)
111
112         self.u, self.v = u_new, v_new
113         print(n)
114
115     return self.u, self.v
116
117
118
119
120
121
122
123     def K_iteration(self, u, v):
124
125         u_half, v_half = copy.deepcopy(u), copy.deepcopy(v)
126         u_new, v_new = copy.deepcopy(u), copy.deepcopy(v)
127
128         # update u:
129         # u{n} to u{n+1/2} line update:
130         for j in range(1, self.j_max): # flag=flag(0/1,0/1), 1:donig
131             0:other
132                 flag = (0,1)
133                 u_half[:,j] = self.UPDATE( u , v, u, j, self.i_max, flag)
134                 # Update function: Update(u, v, doing(u/v), doing(row=j
135                 /column=i), doingAnother, flag)
136
137
138         u_new, v_new = self.BC(u_new, v_new)
139         u_half, v_half = self.BC(u_half, v_half )
140
141
142         # update v:
143         for j in range(1, self.j_max): # flag=flag(0/1,0/1), 1:donig
144             0:other
145                 flag = (0,1)
146                 v_half[:,j] = self.UPDATE( u_half , v, v, j, self.i_max,
147                 flag)
148
149         u_new, v_new = self.BC(u_new, v_new)
150         u_half, v_half = self.BC(u_half, v_half )
151
152         # u{n+1/2} to u{n+1} column update:
153         for i in range(1, self.i_max-1):
154             flag = (1,0)
155             u_new[i,:] = self.UPDATE( u_half , v_half, u_half, i, self.
156             j_max, flag)
157
158         u_new, v_new = self.BC(u_new, v_new)
159         u_half, v_half = self.BC(u_half, v_half )
160
161
162

```

```

163         for i in range(1, self.i_max):
164             flag = (1, 0)
165             v_new[i, :] = self.UPDATE( u_new , v_half , v_half , i , self.
j_max, flag)
166
167
168             u_new, v_new = self.BC(u_new, v_new)
169             u_half, v_half = self.BC(u_half, v_half )
170
171         return u_new , v_new
172
173
174     def UPDATE(self, u, v, U, Idoing, Ianother_max, flag):
175
176         delta = self.dx * flag[1] + self.dy*flag[0]
177
178
179         r = (self.mu*self.dt)/(2*delta**2)
180
181
182         # Creating a, b, c, D
183         a = np.full(Ianother_max-1, -r, dtype = float)
184         b = np.full(Ianother_max-1, 1 + 4*r, dtype = float)
185         c = np.full(Ianother_max-1, -r, dtype = float)
186         d = np.full(Ianother_max+1, 0, dtype = float)
187
188         for ij in range(1, Ianother_max):
189             i, j = Idoing*flag[0] + ij* flag[1] , Idoing*flag[1] + ij*
flag[0]
190
191             cox = (u[i, j]*self.dt/self.dx /2)
192             coy = (v[i, j]*self.dt/self.dy /2)
193             rx = (self.mu*self.dt/(2*self.dx**2))
194             ry = (self.mu*self.dt/(2*self.dy**2))
195             rxy = rx * flag[1] + ry*flag[0]
196             d[ij] = U[i, j]-cox*((U[i+1, j]-U[i-1, j])/(2))-coy*((U[i, j
+1]-U[i, j-1])/(2))+rxy*(U[i+flag[0], j+flag[1]]+U[i-flag[0], j-flag[1]])
197             d[0], d[-1] = U[i*flag[0], j*flag[1]], U[i*flag[0]-1, j*flag
[1]-1]
198
199         # added boundary get full matrix:
200         a = np.concatenate((np.array([0]), a, np.array([0])), axis=0)
201         b = np.concatenate((np.array([1]), b, np.array([1])), axis=0)
202         c = np.concatenate((np.array([0]), c, np.array([0])), axis=0)
203
204
205
206         u_UPDATE = TDMA(a, b, c, d)
207
208         # | 1    0    0    ...    0    0    |
209         # | a1  b1  c1    ...    0    0    |
210         # | 0   a2  b2    ...    0    0    |
211         # | ...  ...  ...    ...    ...  ...  |
212         # | 0   0    0    ...  b(n-2) c(n-2) |
213         # | 0   0    0    ...    0    1    |
214
215         # print("TDMA")
216         # print(u_UPDATE)

```

```

217         return u_UPDATE
218
219
220
221
222
223
224
225 def Do_Solver(x_max, y_max, t_max, dx, dy, dt , mu):
226
227     Try_1 = ADI_Solver(x_max, y_max, t_max, dx, dy, dt , mu)
228     Try_1.Grid_Generate()
229     Try_1.Initialize()
230     u, v =Try_1.Iterative()
231
232     return u,v
233
234
235 def Vorticity(u, v, t):
236     # u,v = np.transpose(u), np.transpose(v)
237
238     Ni, Nj = u.shape
239
240     x = np.linspace(0, 3, Ni)
241     y = np.linspace(0, 1, Nj)
242
243     X, Y = np.meshgrid(x, y)
244
245     omega = np.zeros((Ni, Nj))
246     # omega[1:-1, 1:-1] = (v[1:-1, 2:] - v[1:-1, :-2]) / (x[2] - x[1])
247     - (u[2:, 1:-1] - u[:-2, 1:-1]) / (y[2] - y[1])
248
249     for i in range(1,Ni-1):
250         for j in range(1,Nj-1):
251             omega[i,j] = (v[i+1,j] - v[i-1,j])/(x[2]-x[0])-(u[i,j+1]
252             - u[i,j-1])/(y[2] - y[0])
253
254     omega = np.transpose(omega)
255
256     d = x[2]-x[1]
257
258     print(d)
259
260     plt.figure(figsize=(18, 6))
261     plt.contourf(X, Y, omega, levels=50, cmap='jet')
262     plt.colorbar()
263     plt.title('d={:.3 f}, t={:.2 f}'.format(d, t))
264     plt.xlabel('x')
265     plt.ylabel('y')
266
267
268     plt.tight_layout()
269     plt.savefig('6d {:.3 f} t {:.2 f}.png'.format(d, t))
270     plt.show()
271
272

```

```

273
274
275 def main():
276     x_max = 3
277     y_max = 1
278     t_max = 1.5
279
280     # dx = 0.02
281     # dy = 0.01
282     # dt = 0.005
283
284
285     dx = 0.01
286     dy = 0.01
287     dt = 0.001
288
289
290     mu = 0.01
291     # mu = 1
292
293
294     u,v = Do_Solver(x_max, y_max, t_max, dx, dy, dt, mu)
295     # print(u)
296     Vorticity(u, v, t_max)
297
298
299
300 if __name__ == '__main__':
301     main()
302     # arg1 = [2,2,2,2,2]
303     # arg2 = [1,1,1,1,1]
304     # arg3 = arg1
305     # arg4 = [1,2,3,4,5]
306     # print(TDMA(arg1, arg2, arg3, arg4))

```

Listing 2: Extended Region using ADI Solver