

1. Suppose that the mesh points x_j are chosen to satisfy

$$0 = x_0 < x_1 < x_2 < \cdots < x_{J-1} < x_J = 1$$

but are otherwise arbitrary. The equation $u_t = u_{xx}$ is approximated over the interval $0 \leq t \leq t_F$ by

$$\frac{U_j^{n+1} - U_j^n}{\Delta t} = \frac{2}{\Delta x_{j-1} + \Delta x_j} \left(\frac{U_{j+1}^n - U_j^n}{\Delta x_j} - \frac{U_j^n - U_{j-1}^n}{\Delta x_{j-1}} \right)$$

where $\Delta x_j = x_{j+1} - x_j$. Calculate the truncation error at (x_j, t^n) .

Truncation Error: Begin by expanding each term. On the left hand side we have

$$\begin{aligned} \frac{U_j^{n+1} - U_j^n}{\Delta t} &= \frac{1}{\Delta t} \left[u_t \Delta t + \frac{1}{2} u_{tt} (\Delta t)^2 + \frac{1}{6} u_{ttt} (\Delta t)^3 + \frac{1}{24} u_{tttt} (\Delta t)^4 \cdots \right] \\ &= u_t + \frac{1}{2} u_{tt} \Delta t + \frac{1}{6} u_{ttt} (\Delta t)^2 + \frac{1}{24} u_{tttt} (\Delta t)^3 + \cdots \end{aligned}$$

On the right hand side we have a couple terms to be expanded

$$\begin{aligned} \frac{U_{j+1}^n - U_j^n}{\Delta x_j} &= \frac{1}{\Delta x_j} \left[u_x \Delta x_j + \frac{1}{2} u_{xx} (\Delta x_j)^2 + \frac{1}{6} u_{xxx} (\Delta x_j)^3 + \frac{1}{24} u_{xxxx} (\Delta x_j)^4 + \cdots \right] \\ &= u_x + \frac{1}{2} u_{xx} \Delta x_j + \frac{1}{6} u_{xxx} (\Delta x_j)^2 + \frac{1}{24} u_{xxxx} (\Delta x_j)^3 + \cdots \end{aligned}$$

and

$$\begin{aligned} \frac{U_j^n - U_{j-1}^n}{\Delta x_{j-1}} &= \frac{1}{\Delta x_{j-1}} \left[u_x \Delta x_{j-1} + \frac{1}{2} u_{xx} (\Delta x_{j-1})^2 + \frac{1}{6} u_{xxx} (\Delta x_{j-1})^3 + \frac{1}{24} u_{xxxx} (\Delta x_{j-1})^4 + \cdots \right] \\ &= u_x + \frac{1}{2} u_{xx} \Delta x_{j-1} + \frac{1}{6} u_{xxx} (\Delta x_{j-1})^2 + \frac{1}{24} u_{xxxx} (\Delta x_{j-1})^3 + \cdots \end{aligned}$$

Combining these two to get

$$\begin{aligned} \frac{U_{j+1}^n - U_j^n}{\Delta x_j} - \frac{U_j^n - U_{j-1}^n}{\Delta x_{j-1}} &= \frac{1}{2} u_{xx} (\Delta x_j + \Delta x_{j-1}) + \frac{1}{6} u_{xxx} ((\Delta x_j)^2 - (\Delta x_{j-1})^2) \\ &\quad + \frac{1}{24} u_{xxxx} ((\Delta x_j)^3 + (\Delta x_{j-1})^3) + \cdots \end{aligned}$$

Multiplying through by $\frac{2}{\Delta x_{j-1} + \Delta x_j}$ we get

$$\begin{aligned} \frac{2}{\Delta x_{j-1} + \Delta x_j} \left(\frac{U_{j+1}^n - U_j^n}{\Delta x_j} - \frac{U_j^n - U_{j-1}^n}{\Delta x_{j-1}} \right) &= \\ u_{xx} + \frac{1}{3} u_{xxx} \left[\frac{(\Delta x_j)^2 - (\Delta x_{j-1})^2}{\Delta x_{j-1} + \Delta x_j} \right] &+ \frac{1}{12} u_{xxxx} \left[\frac{(\Delta x_j)^3 + (\Delta x_{j-1})^3}{\Delta x_{j-1} + \Delta x_j} \right] + \cdots \end{aligned}$$

$$= u_{xx} + \frac{1}{3}u_{xxx}(\Delta x_j - \Delta x_{j-1}) + \frac{1}{12}u_{xxxx}((\Delta x_j - \Delta x_{j-1})^2 + \Delta x_j \Delta x_{j-1}) + \dots$$

The truncation error is the difference between the two sides of this equation

$$\begin{aligned} T(x, t) &= \left[u_t + \frac{1}{2}u_{tt}\Delta t + \frac{1}{6}u_{ttt}(\Delta t)^2 + \frac{1}{24}u_{tttt}(\Delta t)^3 + \dots \right] - \\ &\quad \left[u_{xx} + \frac{1}{3}u_{xxx}(\Delta x_j - \Delta x_{j-1}) + \frac{1}{12}u_{xxxx}((\Delta x_j - \Delta x_{j-1})^2 + \Delta x_j \Delta x_{j-1}) + \dots \right] \\ &= (u_t - u_{xx}) + u_{xxx} \left[\frac{\Delta t}{2} - \frac{1}{12}((\Delta x_j - \Delta x_{j-1})^2 + \Delta x_j \Delta x_{j-1}) \right] + \frac{1}{6}u_{ttt}(\Delta t)^2 \\ &\quad + \frac{1}{3}u_{xxx}(\Delta x_j - \Delta x_{j-1}) + \frac{1}{24}u_{tttt}(\Delta t)^3 + \dots \end{aligned}$$

The first term is zero since $u_t = u_{xx}$ and some terms can be combined since $u_{tt} = u_{xxxx}$

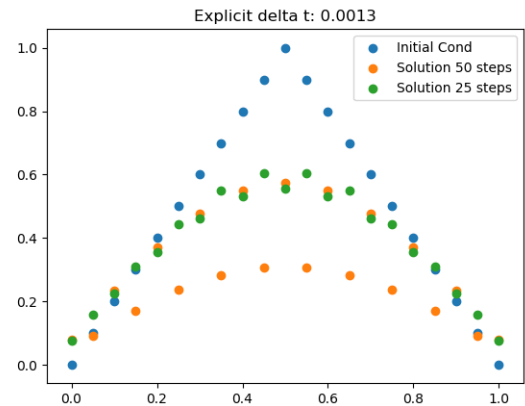
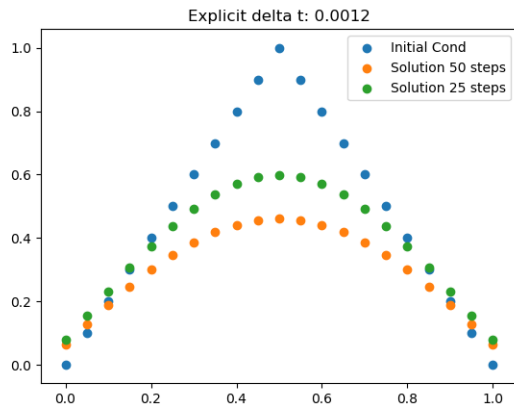
$$\begin{aligned} T(x, t) &= u_{xxx} \left[\frac{\Delta t}{2} - \frac{1}{12}((\Delta x_j - \Delta x_{j-1})^2 + \Delta x_j \Delta x_{j-1}) \right] + \frac{1}{3}u_{xxx}(\Delta x_j - \Delta x_{j-1}) + \text{h. o. t.} \\ &\implies O(\Delta t) \text{ and } O(\Delta x_j - \Delta x_{j-1}) \end{aligned}$$

2. Implement the explicit method, implicit method, CN method for the heat diffusion equation

$$\begin{cases} u_t = u_{xx}, \\ u(0, t) = u(1, t) = 0, \\ u(x, 0) = \begin{cases} 2x, & x \in [0, 0.5], \\ 2 - 2x, & x \in [0.5, 1]. \end{cases} \end{cases}$$

Use $\Delta x = 0.05$ and solve 50 time steps with a.) $\Delta t = 0.0012$, b.) $\Delta t = 0.0013$. Plot your solutions and explain what you observed.

Output Explicit Method:



For the time step $\Delta t = 0.0013$ the solution begins to oscillate, i.e. solution is unstable.

Code Explicit Method:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 f1 = lambda x: 2*x
5 f2 = lambda x: 2-2*x
6 step = 0.05
7 dt1 = 0.0012
8 dt2 = 0.0013
9 deltax = [step*i for i in range(0,21)]
10 iterates = 0
11
12 def control(f1,f2,x):
13     if 0<=x<=0.5:
14         s=f1(x)
15     else:
16         s=f2(x)
17     return(s)
18
19 def tridag(L,M,U,k1=-1,k2=0,k3=1):
20     return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))
21
22 def explicit(f1,f2,step,dt,deltax,iterates):
23     # define s
24     s=dt/(step**2)
25     # initial data
26     initial=[control(f1,f2,x) for x in deltax]
27     # create matrix
28     main=[1-2*s for i in range(0,len(initial))]
29     upper=[s for i in range(0,len(initial)-1)]
30     lower=[s for i in range(0,len(initial)-1)]
31     M =tridag(lower,main,upper)
32     solution =np.matmul(M,initial)
33     # iterate n times
34     for n in range (0,iterates + 1):
35         solution = np.matmul(M,solution)
36     return(initial, solution)
37
38
39 initial1, solution1 = explicit(f1,f2,step,dt1,deltax,iterates)
40 initial2, solution2 = explicit(f1,f2,step,dt2,deltax,iterates)
41 initial3, solution3 = explicit(f1,f2,step,dt1,deltax,25)
42 initial4, solution4 = explicit(f1,f2,step,dt2,deltax,25)
43
44 plt.figure(1)
45 plt.title('Explicit delta t: 0.0012')
46 plt.scatter(deltax,initial1,label="Initial Cond")

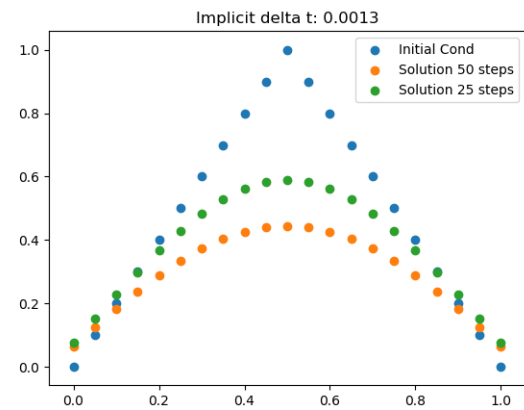
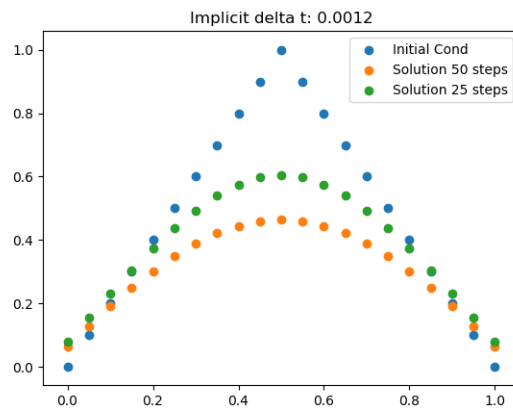
```

```

47 plt.scatter(deltax,solution1, label = "Solution 50 steps")
48 plt.scatter(deltax,solution3, label = "Solution 25 steps")
49 plt.legend()
50 plt.show()
51
52 plt.figure(1)
53 plt.title('Explicit delta t: 0.0013')
54 plt.scatter(deltax,initial2,label="Initial Cond")
55 plt.scatter(deltax,solution2, label = "Solution 50 steps")
56 plt.scatter(deltax,solution4, label = "Solution 25 steps")
57 plt.legend()
58 plt.show()

```

Output Implicit Method:



Code Implicit Method:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 f1 = lambda x: 2*x
5 f2 = lambda x: 2-2*x
6 step = 0.05
7 dt1 = 0.0012
8 dt2 = 0.0013
9 deltax = [step*i for i in range(0,21)]
10 iterates = 50
11
12 def control(f1,f2,x):
13     if 0<=x<=0.5:
14         s=f1(x)
15     else:
16         s=f2(x)
17     return(s)
18

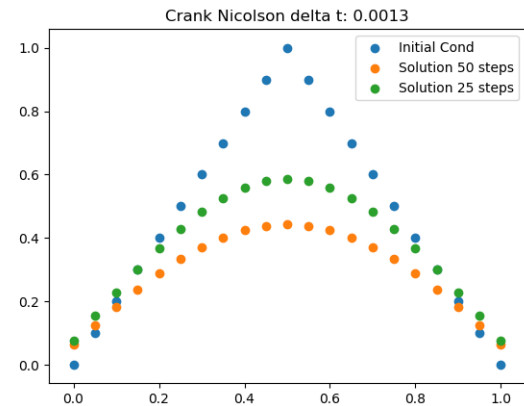
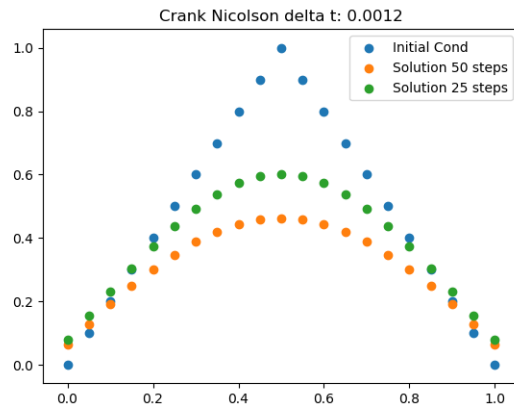
```

```

19 def tridag(L,M,U,k1=-1,k2=0,k3=1):
20     return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))
21
22 def implicit(f1,f2,step,dt,deltax,iterates):
23     # define s
24     s=dt/(step**2)
25     # initial data
26     initial=[control(f1,f2,x) for x in deltax]
27     # create matrix
28     main=[1+2*s for i in range(0,len(initial))]
29     upper=[-s for i in range(0,len(initial)-1)]
30     lower=[-s for i in range(0,len(initial)-1)]
31     M =tridag(lower,main,upper)
32     # calculate inverse of M
33     Minv=np.linalg.inv(M)
34     solution =np.matmul(Minv,initial)
35     # iterate n times
36     for n in range (0,iterates + 1):
37         solution = np.matmul(Minv,solution)
38     return(initial, solution)
39
40 initial1, solution1 = implicit(f1,f2,step,dt1,deltax,iterates)
41 initial2, solution2 = implicit(f1,f2,step,dt2,deltax,iterates)
42 initial3, solution3 = implicit(f1,f2,step,dt1,deltax,25)
43 initial4, solution4 = implicit(f1,f2,step,dt2,deltax,25)
44
45 plt.figure(1)
46 plt.title('Implicit delta t: 0.0012')
47 plt.scatter(deltax,initial1,label="Initial Cond")
48 plt.scatter(deltax,solution1, label = "Solution 50 steps")
49 plt.scatter(deltax,solution3, label = "Solution 25 steps")
50 plt.legend()
51 plt.show()
52
53 plt.figure(1)
54 plt.title('Implicit delta t: 0.0013')
55 plt.scatter(deltax,initial2,label="Initial Cond")
56 plt.scatter(deltax,solution2, label = "Solution 50 steps")
57 plt.scatter(deltax,solution4, label = "Solution 25 steps")
58 plt.legend()
59 plt.show()

```

Output Crank Nicolson Method:



Code Crank Nicolson Method:

```

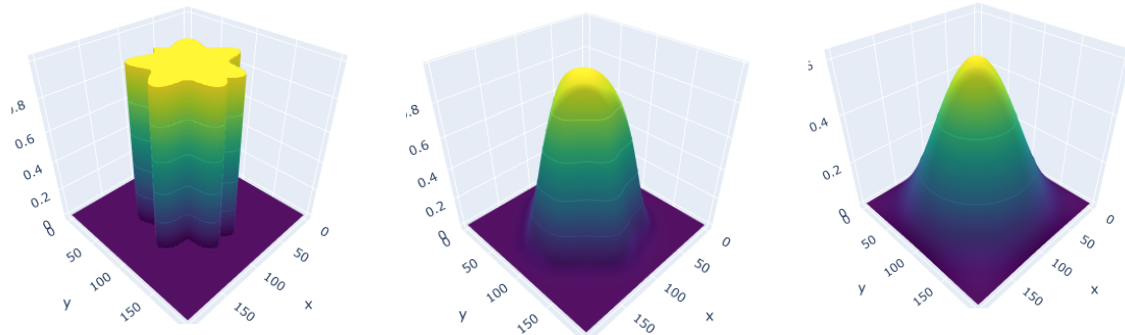
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 f1 = lambda x: 2*x
5 f2 = lambda x: 2-2*x
6 step = 0.05
7 dt1 = 0.0012
8 dt2 = 0.0013
9 deltax = [step*i for i in range(0,21)]
10 iterates = 50
11 theta = 1/2
12
13 def control(f1,f2,x):
14     if 0<=x<=0.5:
15         s=f1(x)
16     else:
17         s=f2(x)
18     return(s)
19
20 def tridag(L,M,U,k1=-1,k2=0,k3=1):
21     return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))
22
23 def crank(f1, f2, theta, step, dt, deltax, iterates):
24     s1=(dt/(step**2))*theta
25     s2=(dt/(step**2))*(1-theta)
26     initial=[control(f1,f2,x) for x in deltax]
27     mainI=[1+2*s1 for i in range(0,len(initial))]
28     upperI=[-s1 for i in range(0,len(initial)-1)]
29     lowerI=[-s1 for i in range(0,len(initial)-1)]
30     mainE=[1-2*s2 for i in range(0,len(initial))]
31     upperE=[s2 for i in range(0,len(initial)-1)]

```

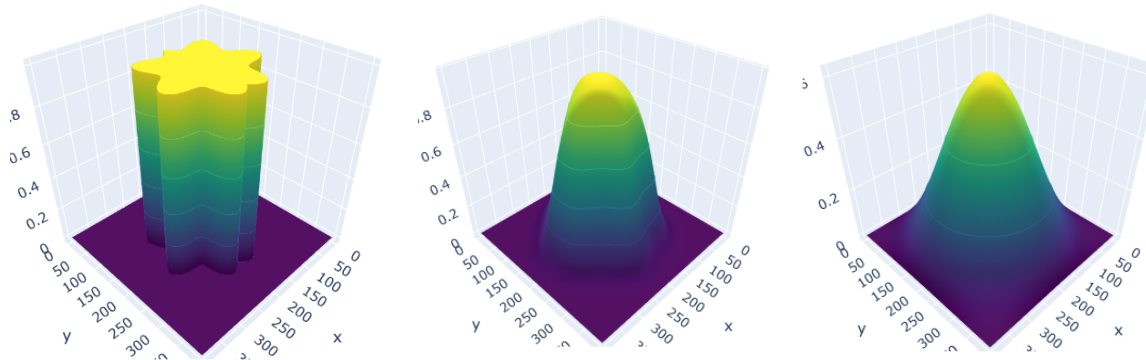
```
32     lowerE=[s2 for i in range(0,len(initial)-1)]
33     MI=tridag(lowerI,mainI,upperI)
34     ME=tridag(lowerE,mainE,upperE)
35     MIinv=np.linalg.inv(MI)
36     Mm=np.matmul(ME,MIinv)
37     solution = np.matmul(Mm,initial)
38     for n in range(0,iterates + 1):
39         solution = np.matmul(Mm,solution)
40     return(initial,solution)
41
42 initial1,solution1=crank(f1,f2,theta,step,dt1,deltax,iterates)
43 initial2,solution2=crank(f1,f2,theta,step,dt2,deltax,iterates)
44 initial3,solution3=crank(f1,f2,theta,step,dt1,deltax,25)
45 initial4,solution4=crank(f1,f2,theta,step,dt2,deltax,25)
46
47 plt.figure(1)
48 plt.title('Crank Nicolson delta t: 0.0012')
49 plt.scatter(deltax,initial1,label="Initial Cond")
50 plt.scatter(deltax,solution1, label = "Solution 50 steps")
51 plt.scatter(deltax,solution3, label = "Solution 25 steps")
52 plt.legend()
53 plt.show()
54
55 plt.figure(1)
56 plt.title('Crank Nicolson delta t: 0.0013')
57 plt.scatter(deltax,initial2,label="Initial Cond")
58 plt.scatter(deltax,solution2, label = "Solution 50 steps")
59 plt.scatter(deltax,solution4, label = "Solution 25 steps")
60 plt.legend()
61 plt.show()
```

3. Implement the ADI method for $u_t = u_{xx} + u_{yy}$ on the square $-1 < x < 1, -1 < y < 1$, with homogeneous Dirichlet boundary boundary condition. The initial condition is $u(x, y, 0) = f(x, y)$. $f(x, y) = 1$ inside a “flower” region parametrized by $x = r \cos(\theta)$ and $y = r \sin(\theta)$, where $r = 0.5 + 0.1 \sin(6\theta)$ and $\theta \in [0, 2\pi]$. $f(x, y) = 0$ in the rest of the square. Using $\Delta x = \Delta y = 1/100, 1/200$, and $1/400$ and proper Δt to indicate your solution is convincing by plotting several snapshots in your iterations.

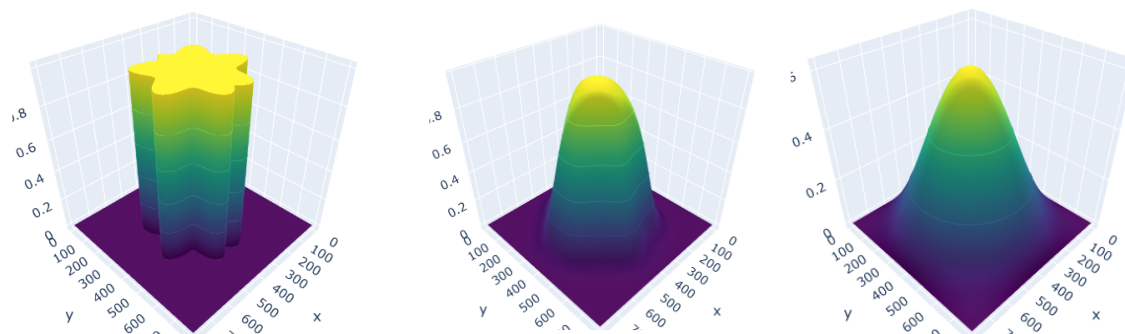
For $\Delta x = \Delta y = 1/100, \Delta t = 0.012$. From left to right initial, after 1 iteration, after 10 iterations.



For $\Delta x = \Delta y = 1/200, \Delta t = 0.012$. From left to right initial, after 1 iteration, after 10 iterations.



For $\Delta x = \Delta y = 1/400, \Delta t = 0.012$. From left to right initial, after 1 iteration, after 10 iterations.



Code for ADI

```

1 import numpy as np
2 import pandas as pd
3 import plotly.graph_objects as go
4
5 # parameters
6 deltaxy1 = 1/100 # delta x = delta y
7 deltaxy2 = 1/200
8 deltaxy3 = 1/400
9 step1 = 200
10 step2 = 400
11 step3 = 800
12 deltat = 0.0012
13 iterates1 = 1
14 iterates2 = 10
15 s1 = (deltat/(deltaxy1**2)) # mu_x = mu_y
16 s2 = (deltat/(deltaxy2**2))
17 s3 = (deltat/(deltaxy3**2))
18
19 # initial condition
20
21 x = lambda theta: (0.5+ 0.1*np.sin(6*theta))*np.cos(theta)
22
23 y = lambda theta: (0.5+ 0.1*np.sin(6*theta))*np.sin(theta)
24
25
26 def initial_mat(x,y,n):
27     initial=np.zeros(shape=(n,n))
28     inter=np.linspace(-1,1,n)
29     for i in range(0,len(inter)):
30         for j in range(0,len(inter)):
31             ri=np.sqrt(inter[i]**2 +inter[j]**2)
32             r=np.sqrt(x(np.arccos(inter[i]/ri))**2
33                     + y(np.arccos(inter[j]/ri))**2)
34             if abs(ri)>abs(r):
35                 initial[i,j]=0
36             else:
37                 initial[i,j]=1
38     return(initial)
39
40 # tridiagonal matrix
41 def tridag(L,M,U,k1=-1,k2=0,k3=1):
42     return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))
43
44 # one adi xy sweep
45 def adi_method(s, deltaxy, step, initial):
46     # diagonals

```

```

47     lower=[-s/2 for i in range(0,len(initial)-1)]
48     main=[1+s for i in range(0,len(initial))]
49     upper=[-s/2 for i in range(0,len(initial)-1)]
50
51     # tridiagonal matrix
52     M = tridag(lower,main,upper)
53     # calculate inverse
54     Minv = np.linalg.inv(M)
55
56     initial = np.transpose(initial)
57     # x sweep
58     for i in range(0, step):
59         initial[i] = np.matmul(Minv,np.transpose(initial[i]))
60         i = i + 1
61     # y sweep
62     initial = np.transpose(initial)
63     for j in range(0, step):
64         initial[j] = np.matmul(Minv,np.transpose(initial[j]))
65         j = j + 1
66
67     initial = np.transpose(initial)
68
69     return initial
70
71 def adi_loop(s, deltaxy, step, initial, iterates):
72     solution = adi_method(s, deltaxy, step, initial)
73     for n in range(0,iterates):
74         adi_method(s, deltaxy, step, solution)
75         n = n + 1
76     return solution
77
78 def plot_surface(solution):
79     surface_df = pd.DataFrame(solution)
80     surface_df.to_csv('surface_data.csv', index = False)
81     surface_data = pd.read_csv('surface_data.csv')
82     fig = go.Figure(data=[go.Surface(z=surface_data.values,
83                                     colorscale='Viridis')])
84     fig.update_traces(contours_z=dict(show=True,
85                                     usecolormap=True, highlightcolor="limegreen",
86                                     project_z=True))
87     fig.update_layout(title='Solution', autosize=False,
88                       width=500, height=500,margin=dict(l=65, r=50,
89                                                         b=65, t=90), xaxis = dict(visible = False))
90     fig.show()
91
92
93 # deltax = deltaxy = 1/100

```

```
94 ## initial
95 plot_surface(initial_mat(x,y,step1))
96 ## after 1 iterations
97 plot_surface(adi_loop(s1, deltaxy1, step1,
98     initial_mat(x,y,step1), iterates1))
99 ## after 10 iterations
100 plot_surface(adi_loop(s1, deltaxy1, step1,
101     initial_mat(x,y,step1), iterates2))
102
103 # deltax = deltay = 1/200
104 ## initial
105 plot_surface(initial_mat(x,y,step2))
106 ## after 1 iterations
107 plot_surface(adi_loop(s2, deltaxy2, step2,
108     initial_mat(x,y,step2), iterates1))
109 # after 10 iterations
110 plot_surface(adi_loop(s2, deltaxy2, step2,
111     initial_mat(x,y,step2), iterates2))
112
113 # deltax = deltay = 1/400
114 ## initial
115 plot_surface(initial_mat(x,y,step3))
116 ## after 1 iterations
117 plot_surface(adi_loop(s3, deltaxy3, step3,
118     initial_mat(x,y,step3), iterates1))
119 ## after 10 iterations
120 plot_surface(adi_loop(s3, deltaxy3, step3,
121     initial_mat(x,y,step3), iterates2))
```