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 $0 \le t \le t_F$ 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

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

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

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

$$u_{xx} + \frac{1}{3}u_{xxx}\left[\frac{(\Delta x_j)^2 - (\Delta x_{j-1})^2}{\Delta x_{j-1} + \Delta x_j}\right] + \frac{1}{12}u_{xxxx}\left[\frac{(\Delta x_j)^3 + (\Delta x_{j-1})^3}{\Delta x_{j-1} + \Delta x_j}\right] + \cdots$$

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

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

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

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

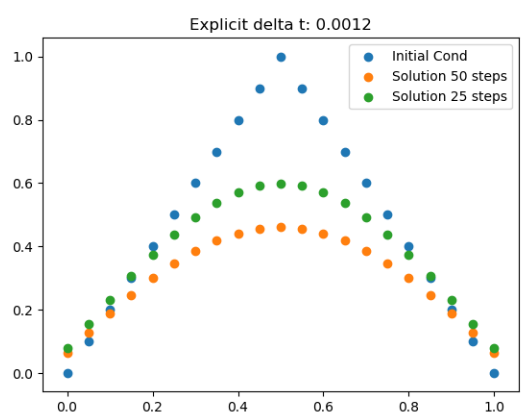
$$T(x,t) = u_{xxxx}\left[\frac{\Delta t}{2} - \frac{1}{12}((\Delta x_j - \Delta x_{j-1})^2 + \Delta x_j \Delta x_{j-1})\right] + \frac{1}{3}u_{xxx}(\Delta x_j - \Delta x_{j-1}) + \text{ h. o. t.}$$
$$\implies O(\Delta t) \text{ and } O(\Delta x_j - \Delta x_{j-1}))$$

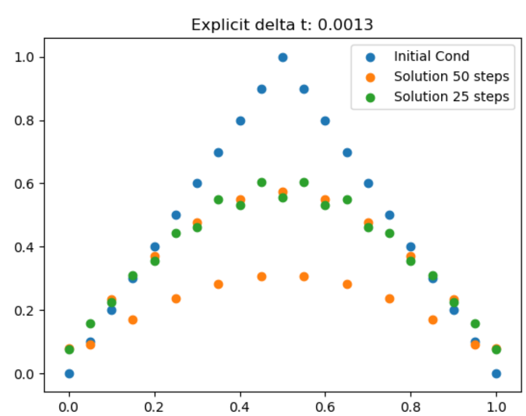
2. Implement the explicit method, implicit method, CN method for the heat diffusion equation

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

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:





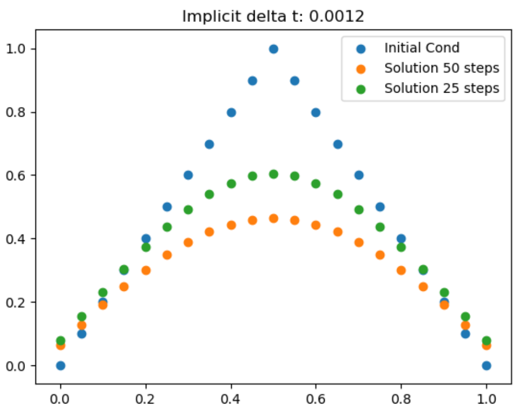
For the time step $\Delta t = 0.0013$ the solution begins to oscillate, i.e. solution is unstable.

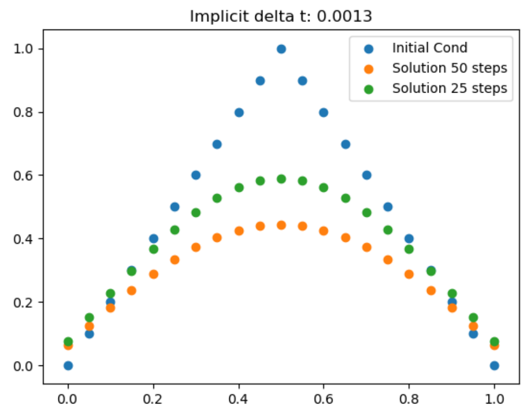
Code Explicit Method:

```
import numpy as np
import matplotlib.pyplot as plt

f1 = lambda x: 2*x
f2 = lambda x: 2-2*x
step = 0.05
dt1 = 0.0012
dt2 = 0.0013
deltax = [step*i for i in range(0,21)]
iterates = 0

def control(f1,f2,x):
    if 0<=x<=0.5:
        s=f1(x)
    else:
        s=f2(x)
    return(s)

def tridag(L,M,U,k1=-1,k2=0,k3=1):
    return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))

def explicit(f1,f2,step,dt,deltax,iterates):
    # define s
    s=dt/(step**2)
    # initial data
    initial=[control(f1,f2,x) for x in deltax]
    # create matrix
    main=[1-2*s for i in range(0,len(initial))]
    upper=[s for i in range(0,len(initial)-1)]
    lower=[s for i in range(0,len(initial)-1)]
    M =tridag(lower,main,upper)
    solution =np.matmul(M,initial)
    # iterate n times
    for n in range (0,iterates + 1):
        solution = np.matmul(M,solution)
    return(initial, solution)


initial1, solution1 = explicit(f1,f2,step,dt1,deltax,iterates)
initial2, solution2 = explicit(f1,f2,step,dt2,deltax,iterates)
initial3, solution3 = explicit(f1,f2,step,dt1,deltax,25)
initial4, solution4 = explicit(f1,f2,step,dt2,deltax,25)

plt.figure(1)
plt.title('Explicit delta t: 0.0012')
plt.scatter(deltax,initial1,label="Initial Cond")
```

```
plt.scatter(deltax,solution1, label = "Solution 50 steps")
plt.scatter(deltax,solution3, label = "Solution 25 steps")
plt.legend()
plt.show()

plt.figure(1)
plt.title('Explicit delta t: 0.0013')
plt.scatter(deltax,initial2,label="Initial Cond")
plt.scatter(deltax,solution2, label = "Solution 50 steps")
plt.scatter(deltax,solution4, label = "Solution 25 steps")
plt.legend()
plt.show()
```

Output Implicit Method:



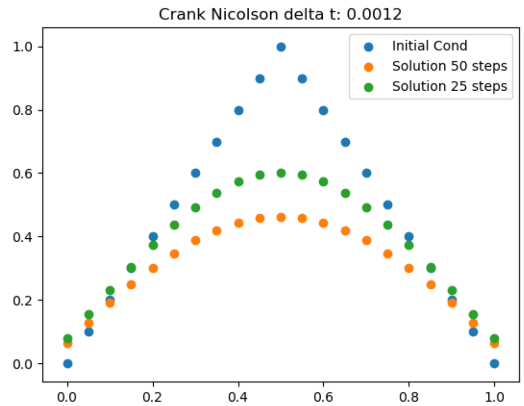


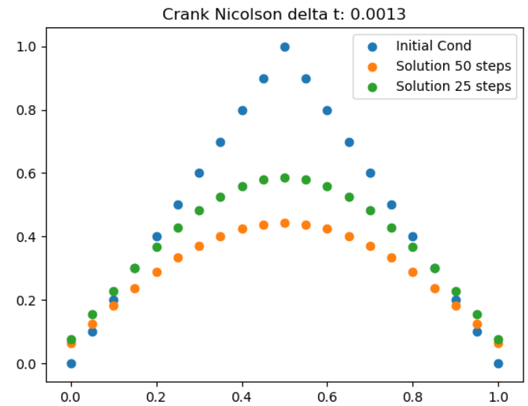
Code Implicit Method:

```
import numpy as np
import matplotlib.pyplot as plt

f1 = lambda x: 2*x
f2 = lambda x: 2-2*x
step = 0.05
dt1 = 0.0012
dt2 = 0.0013
deltax = [step*i for i in range(0,21)]
iterates = 50

def control(f1,f2,x):
    if 0<=x<=0.5:
        s=f1(x)
    else:
        s=f2(x)
    return(s)

```

```
def tridag(L,M,U,k1=-1,k2=0,k3=1):
    return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))

def implicit(f1,f2,step,dt,deltax,iterates):
    # define s
    s=dt/(step**2)
    # initial data
    initial=[control(f1,f2,x) for x in deltax]
    # create matrix
    main=[1+2*s for i in range(0,len(initial))]
    upper=[-s for i in range(0,len(initial)-1)]
    lower=[-s for i in range(0,len(initial)-1)]
    M =tridag(lower,main,upper)
    # calculate inverse of M
    Minv=np.linalg.inv(M)
    solution =np.matmul(Minv,initial)
    # iterate n times
    for n in range (0,iterates + 1):
        solution = np.matmul(Minv,solution)
    return(initial, solution)

initial1, solution1 = implicit(f1,f2,step,dt1,deltax,iterates)
initial2, solution2 = implicit(f1,f2,step,dt2,deltax,iterates)
initial3, solution3 = implicit(f1,f2,step,dt1,deltax,25)
initial4, solution4 = implicit(f1,f2,step,dt2,deltax,25)

plt.figure(1)
plt.title('Implicit delta t: 0.0012')
plt.scatter(deltax,initial1,label="Initial Cond")
plt.scatter(deltax,solution1, label = "Solution 50 steps")
plt.scatter(deltax,solution3, label = "Solution 25 steps")
plt.legend()
plt.show()

plt.figure(1)
plt.title('Implicit delta t: 0.0013')
plt.scatter(deltax,initial2,label="Initial Cond")
plt.scatter(deltax,solution2, label = "Solution 50 steps")
plt.scatter(deltax,solution4, label = "Solution 25 steps")
plt.legend()
plt.show()
```

Output Crank Nicolson Method:





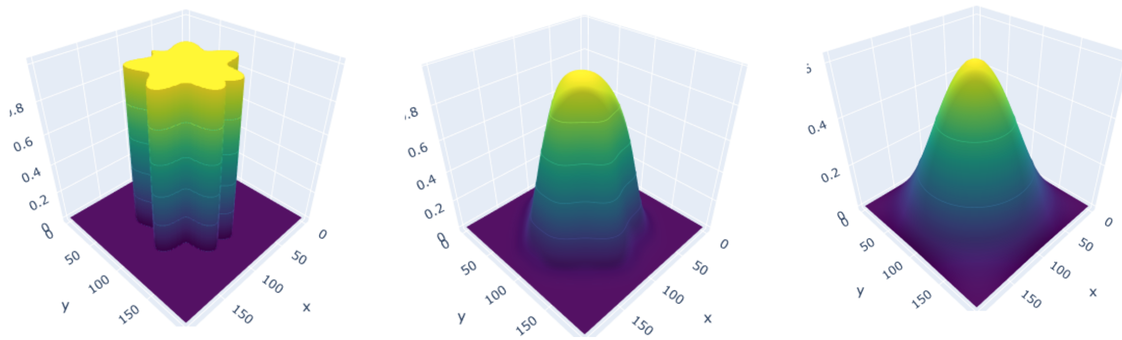
Code Crank Nicolson Method:

```
import numpy as np
import matplotlib.pyplot as plt

f1 = lambda x: 2*x
f2 = lambda x: 2-2*x
step = 0.05
dt1 = 0.0012
dt2 = 0.0013
deltax = [step*i for i in range(0,21)]
iterates = 50
theta = 1/2

def control(f1,f2,x):
    if 0<=x<=0.5:
        s=f1(x)
    else:
        s=f2(x)
    return(s)

def tridag(L,M,U,k1=-1,k2=0,k3=1):
    return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))

def crank(f1, f2, theta, step, dt, deltax, iterates):
    s1=(dt/(step**2))*theta
    s2=(dt/(step**2))*(1-theta)
    initial=[control(f1,f2,x) for x in deltax]
    mainI=[1+2*s1 for i in range(0,len(initial))]
    upperI=[-s1 for i in range(0,len(initial)-1)]
    lowerI=[-s1 for i in range(0,len(initial)-1)]
    mainE=[1-2*s2 for i in range(0,len(initial))]
    upperE=[s2 for i in range(0,len(initial)-1)]
```

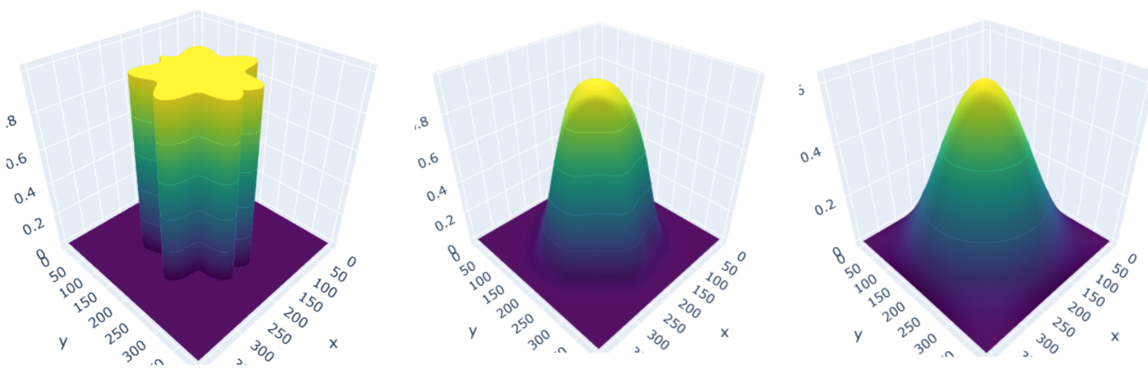
```
    lowerE=[s2 for i in range(0,len(initial)-1)]
    MI=tridag(lowerI,mainI,upperI)
    ME=tridag(lowerE,mainE,upperE)
    MIinv=np.linalg.inv(MI)
    Mm=np.matmul(ME,MIinv)
    solution = np.matmul(Mm,initial)
    for n in range(0,iterates + 1):
        solution = np.matmul(Mm,solution)
    return(initial,solution)

initial1,solution1=crank(f1,f2,theta,step,dt1,deltax,iterates)
initial2,solution2=crank(f1,f2,theta,step,dt2,deltax,iterates)
initial3,solution3=crank(f1,f2,theta,step,dt1,deltax,25)
initial4,solution4=crank(f1,f2,theta,step,dt2,deltax,25)

plt.figure(1)
plt.title('Crank Nicolson delta t: 0.0012')
plt.scatter(deltax,initial1,label="Initial Cond")
plt.scatter(deltax,solution1, label = "Solution 50 steps")
plt.scatter(deltax,solution3, label = "Solution 25 steps")
plt.legend()
plt.show()

plt.figure(1)
plt.title('Crank Nicolson delta t: 0.0013')
plt.scatter(deltax,initial2,label="Initial Cond")
plt.scatter(deltax,solution2, label = "Solution 50 steps")
plt.scatter(deltax,solution4, label = "Solution 25 steps")
plt.legend()
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.

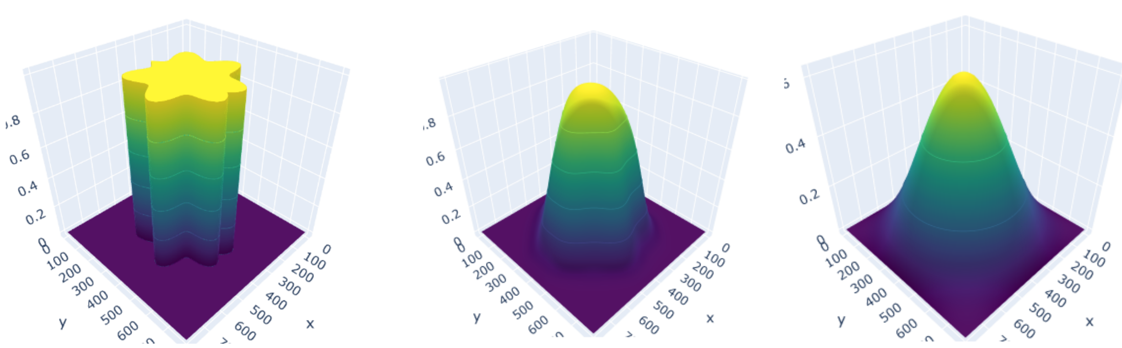
   For $\Delta x = \Delta y = 1/100$, $\Delta t = 0.012$. From left to right initial, after 1 iteration, after 10 iterations.

   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

```
import numpy as np
import pandas as pd
import plotly.graph_objects as go

# parameters
deltaxy1 = 1/100 # delta x = delta y
deltaxy2 = 1/200
deltaxy3 = 1/400
step1 = 200
step2 = 400
step3 = 800
deltat = 0.0012
iterates1 = 1
iterates2 = 10
s1 = (deltat/(deltaxy1**2)) # mu_x = mu_y
s2 = (deltat/(deltaxy2**2))
s3 = (deltat/(deltaxy3**2))

# initial condition

x = lambda theta: (0.5+ 0.1*np.sin(6*theta))*np.cos(theta)

y = lambda theta: (0.5+ 0.1*np.sin(6*theta))*np.sin(theta)


def initial_mat(x,y,n):
    initial=np.zeros(shape=(n,n))
    inter=np.linspace(-1,1,n)
    for i in range(0,len(inter)):
        for j in range(0,len(inter)):
            ri=np.sqrt(inter[i]**2 +inter[j]**2)
            r=np.sqrt(x(np.arccos(inter[i]/ri))**2
                + y(np.arccos(inter[i]/ri))**2)
            if abs(ri)>abs(r):
                initial[i,j]=0
            else:
                initial[i,j]=1
    return(initial)

# tridiagonal matrix
def tridag(L,M,U,k1=-1,k2=0,k3=1):
    return(np.diag(L,k1)+np.diag(M,k2)+np.diag(U,k3))

# one adi xy sweep
def adi_method(s, deltaxy, step, initial):
    # diagonals
```

```
    lower=[-s/2 for i in range(0,len(initial)-1)]
    main=[1+s for i in range(0,len(initial))]
    upper=[-s/2 for i in range(0,len(initial)-1)]

    # tridiagonal matrix
    M = tridag(lower,main,upper)
    # calculate inverse
    Minv = np.linalg.inv(M)

    initial = np.transpose(initial)
    # x sweep
    for i in range(0, step):
        initial[i] = np.matmul(Minv,np.transpose(initial[i]))
        i = i + 1
    # y sweep
    initial = np.transpose(initial)
    for j in range(0, step):
        initial[j] = np.matmul(Minv,np.transpose(initial[j]))
        j = j + 1

    initial = np.transpose(initial)

    return initial

def adi_loop(s, deltaxy, step, initial, iterates):
    solution = adi_method(s, deltaxy, step, initial)
    for n in range(0,iterates):
        adi_method(s, deltaxy, step, solution)
        n = n + 1
    return solution

def plot_surface(solution):
    surface_df = pd.DataFrame(solution)
    surface_df.to_csv('surface_data.csv', index = False)
    surface_data = pd.read_csv('surface_data.csv')
    fig = go.Figure(data=[go.Surface(z=surface_data.values,
        colorscale='Viridis')])
    fig.update_traces(contours_z=dict(show=True,
        usecolormap=True, highlightcolor="limegreen",
        project_z=True))
    fig.update_layout(title='Solution', autosize=False,
        width=500, height=500,margin=dict(l=65, r=50,
            b=65, t=90), xaxis = dict(visible = False))
    fig.show()


# deltax = deltay = 1/100
```

```
## initial
plot_surface(initial_mat(x,y,step1))
## after 1 iterations
plot_surface(adi_loop(s1, deltaxy1, step1,
    initial_mat(x,y,step1), iterates1))
## after 10 iterations
plot_surface(adi_loop(s1, deltaxy1, step1,
    initial_mat(x,y,step1), iterates2))

# deltax = deltay = 1/200
## initial
plot_surface(initial_mat(x,y,step2))
## after 1 iterations
plot_surface(adi_loop(s2, deltaxy2, step2,
    initial_mat(x,y,step2), iterates1))
# after 10 iterations
plot_surface(adi_loop(s2, deltaxy2, step2,
    initial_mat(x,y,step2), iterates2))

# deltax = deltay = 1/400
## initial
plot_surface(initial_mat(x,y,step3))
## after 1 iterations
plot_surface(adi_loop(s3, deltaxy3, step3,
    initial_mat(x,y,step3), iterates1))
## after 10 iterations
plot_surface(adi_loop(s3, deltaxy3, step3,
    initial_mat(x,y,step3), iterates2))
```