#1 $$\frac{dc}{dt} = D\frac{d^2c}{dx^2}$$

$$\frac{dc}{\Delta t} = D\frac{C_{i+1}^n - 2C_i^n + C_{i-1}^n}{\Delta x^2}$$

using v. Neumann

$$\sigma = \frac{D\Delta t}{\Delta x^2} \le \frac{1}{2}$$ for RK4 prescribe $\sigma = \frac{1}{2}$

$$\sigma = \frac{1}{2} = \frac{D\Delta t}{\Delta x^2} \Rightarrow \Delta t = \frac{\Delta x^2}{2D}$$; $\Delta x = \frac{x}{N} = \frac{10}{200} = 0.05$

$$\Delta t = \frac{(0.05)^2}{2(0.1)} = 0.0125s$$

for Crank Nicolson, it was seen through experiments that even at $\Delta t = 1s$ which is only one solution between 2s & 4s it still produced an accurate and stable answer

$$\sigma = \frac{D\Delta t}{\Delta x^2} = \frac{(0.1)(1)}{0.05} = 2$$ so even for $\sigma > \frac{1}{2}$ & $\sigma > 1$

it is still good

Justine John "JJ" A. Serdoncillo
December 10, 2022

Homework 3

Number 1 a)

Below is the plot for the diffusion equation using Runge-Kutta 4th Order Method. The time step used is 0.0125s as shown in the written work. It can be seen that the numerical solution at t=4s matches exactly on top of the analytical solution.

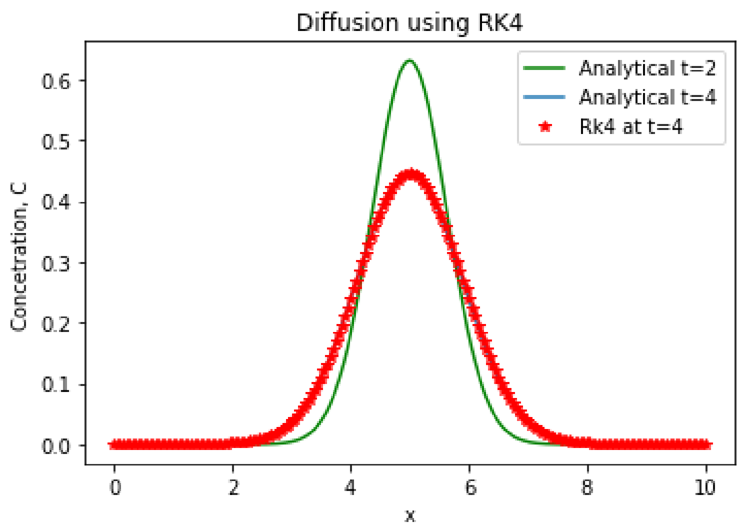

Figure 1: 1a) Diffusion Equation using Runge-Kutta 4

Number 1 b)

Below is the plot for the diffusion equation using Crank-Nicolson method. Upon experimenting with various time steps. It can be seen that it is stable even for high values of dt. The time step used below is t=1s which is the highest possible value that would make sense in this case. It can be seen that the numerical solution at t=4s still matches exactly on top of the analytical solution.

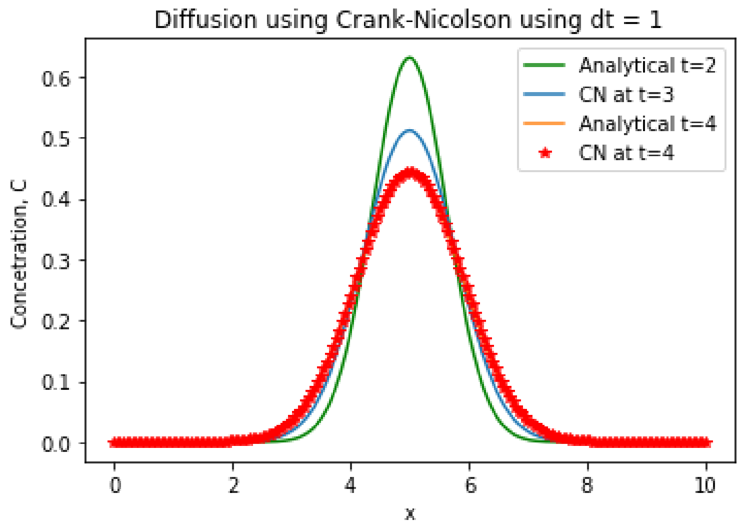

Figure 2: 1b) Diffusion Equation using Crank-Nicolson

Number 2)

Given the boundary conditions of the rectangular copper plate and the Laplace's equation, the analytical solution of the steady state was given and can be shown by the plot below.

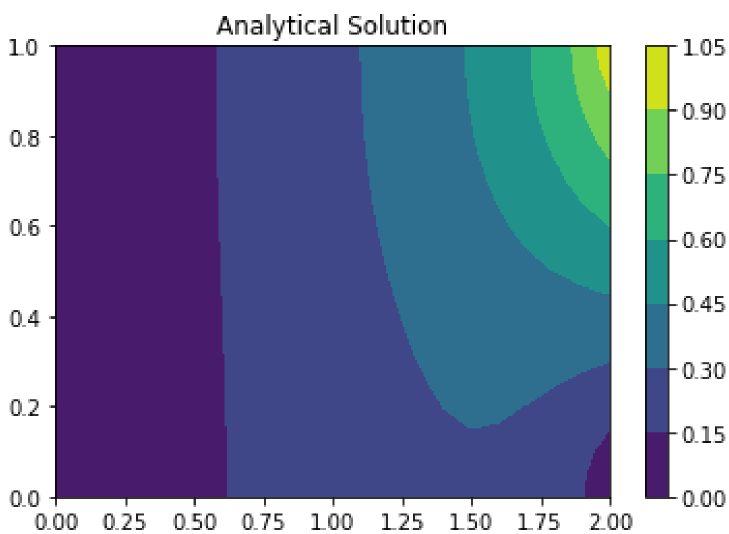

Figure 3: 2) Analytical Solution of the Laplace Equation

For this problem, this will be solved iteratively by using the given equation for the residual and prescribing a tolerance of 10^{-6}

Number 2 a)

The figure below shows the equation solved using Jacobi. The number of iterations can be seen at the summary at the end.

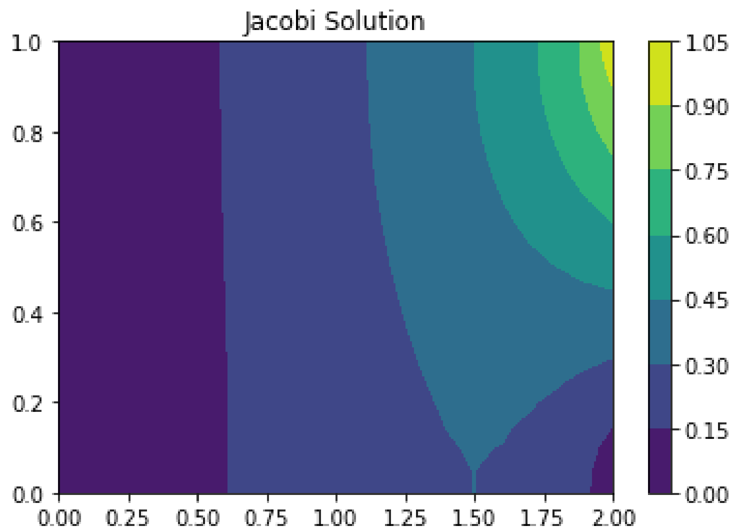

Figure 4: 2a) Laplace Equation using Jacobi

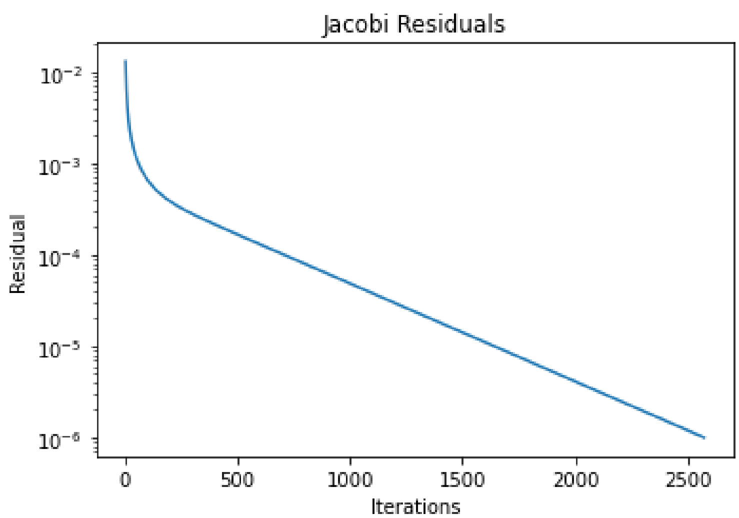

Figure 5: 2a) Residuals from Laplace Equation using Jacobi

Number 2 b)

The figure below shows the equation solved using Gauss-Seidel. It can be seen that it converges earlier than than the Jacobi solution. Using this method can have a bias when doing the i or j index first but when run both cases, did not pose any difference to the number of iterations.

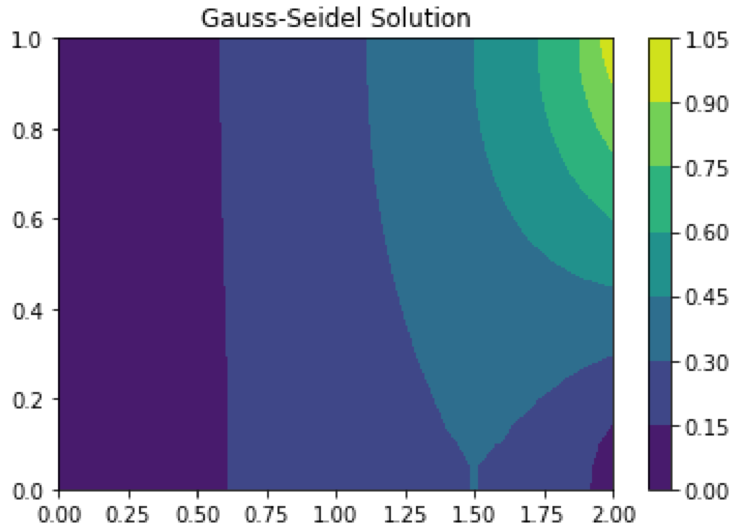

Figure 6: 2b) Laplace Equation using Gauss-Seidel

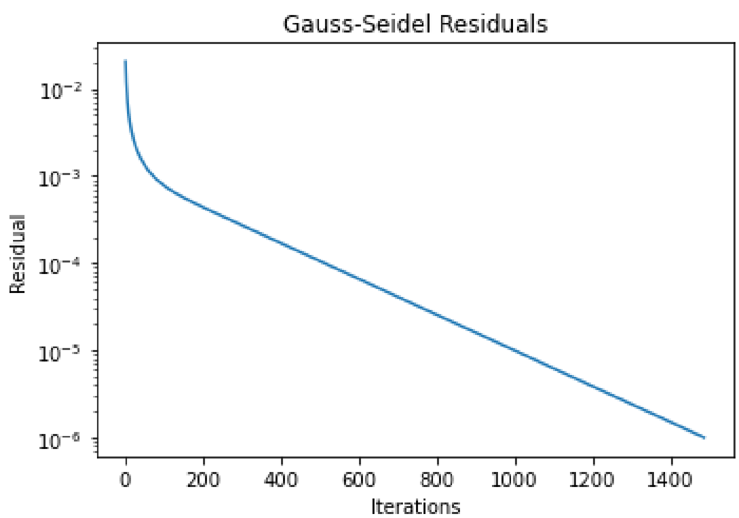

Figure 7: 2b) Residuals from Laplace Equation using Gauss-Seidel

Number 2 c)

The Successive-Over Relaxation (SOR) method was used across multiple values of w. The effect of w on iterations can be seen below. It can be seen that most of the iterations were higher than 2000 which was used as a ceiling. This shows that when using the wrong w, it can be more expensive to use.

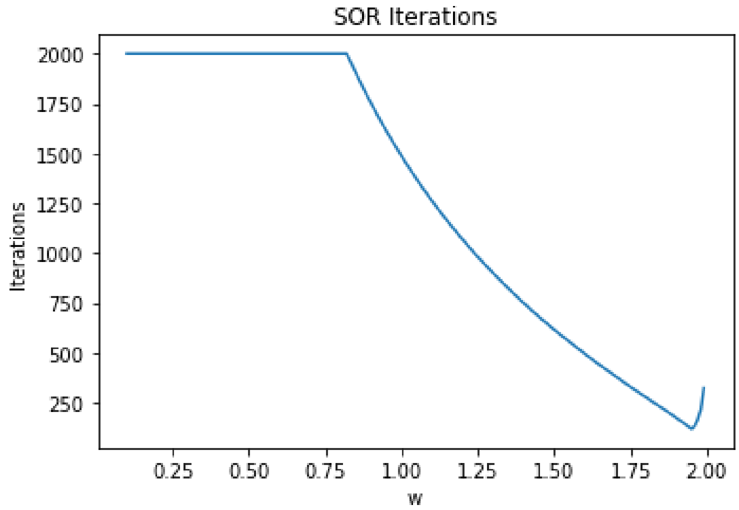

Figure 8: 2c) Effect of w on iterations using SOR

Using w=1.95 produced the lowest number of iterations and is used to produce the plot below.

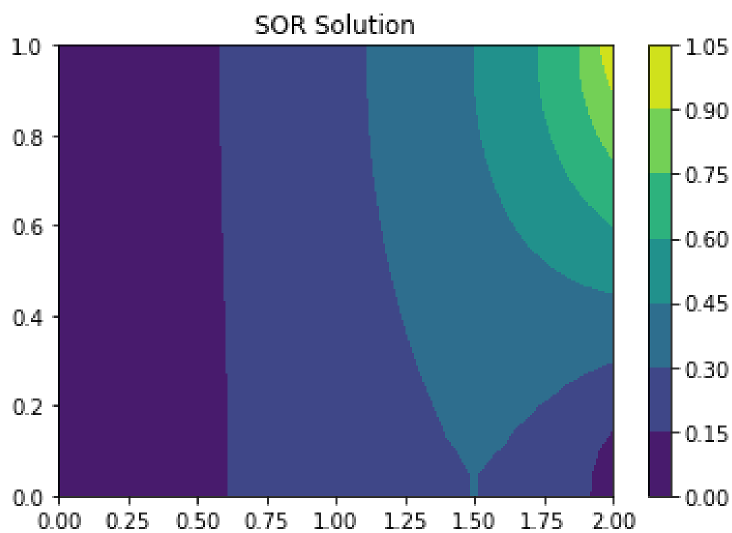

Figure 9: 2c) Laplace Equation using SOR

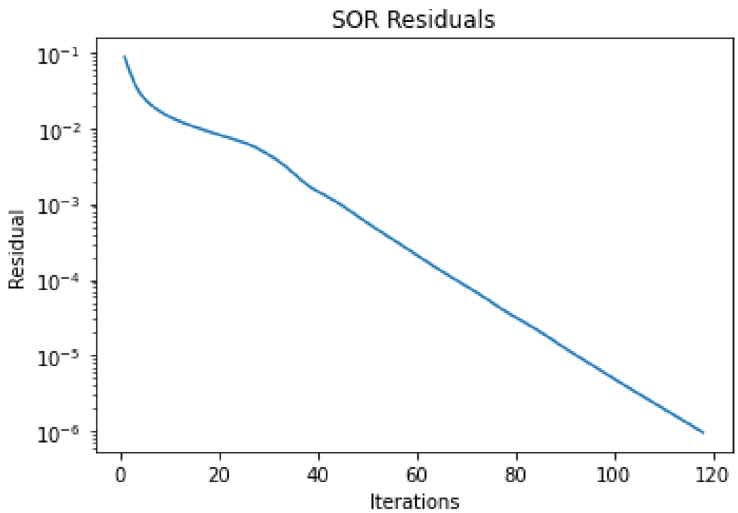

Figure 10: 2c) Residuals from Laplace Equation using SOR

The table below shows the number of iterations to convergence using the different methods.

	Jacobi	Gauss-Seidel	SOR
Iterations	2574	1486	118
Final Residual	9.978e-7	9.976e-7	9.643

Table 1: Validation Accuracy vs. Type of Activation Function with no Softmax

It can be seen from above that although all of the methods gave substantial results compared to the analytical solution, the number of iterations is drastically different. Gauss-Seidel is better than Jacobi. Using SOR drastically decreases the iterations given that the right value of w is used. If not, then it will be more computationally expensive to do so.

Appendix)

Python Code for 1

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as la

x0 = 5
D = 0.1

def Canal(x,t):
    return 1/(np.sqrt(4*np.pi*D*t)) * np.exp(-(x-x0)**2/(4*D*t))

N = 200

x = np.linspace(0,10,N+1)
dx = 10/N
#print(dx)
dt = np.round(dx**2/(2*D),10)
#dt = 0.0142 # hmm only stops working at 0.0142
print(dt)
numt = int(np.floor(2/dt))
#print(numt)
t = np.linspace(2,4,numt+1)
#print(t[0])
#print(t[1])
print('CFL is', str(4*dt/dx))
```

```python

def bound(f):
    f[0] = 0
    f[-1] = 0
    return f

cti = Canal(x,2)
bound(cti)
ctf = Canal(x,4)
bound(ctf)

''' Right Hand Side Functions '''
def CS(f, n, i):
    RHS = D * (f[n][i+1] - 2*f[n][i] + f[n][i-1] ) / (dx)**2
    return RHS
''' Left Hand Side Functions '''
def Rk4(f, n, i):
    k0 = CS(f,n,i)
    k1 = CS(f,n,i) + dt/2 * k0
    k2 = CS(f,n,i) + dt/2 * k1
    k3 = CS(f,n,i) + dt   * k2
    df  = (k0 + 2*k1 + 2*k2 + k3) / 6
    LHS = f[n][i] + dt * df
    return LHS


# Exact Solution at t=2 and t=4
fig, ax = plt.subplots()
ax.plot(x, cti)
ax.plot(x, ctf)
plt.show()

C = np.zeros((len(t),len(x)))
C[0] = cti
for n in range(len(t)-1):
    for i in np.arange(N):
        C[n+1][i] = Rk4(C, n, i)
    C[n+1] = bound(C[n+1])

# Analytical Solution
plt.plot(x, C[0],'g',label='Analytical t=2')
#ax.plot(x, C[40],'g')
#ax.plot(x, C[80],'g')
#ax.plot(x, C[120],'g')
plt.plot(x, ctf, label='Analytical t=4')
plt.plot(x, C[-1],'r*', label='Rk4 at t=4')
plt.legend()
plt.title('Diffusion using RK4')
plt.xlabel('x')
plt.ylabel('Concetration, C')
plt.show()


#### Part 2 Crank Nicolson
dt = 1
# dt works even as high as 2
numt = int(np.floor(2/dt))
print(numt)
t = np.linspace(2,4,numt+1)
print(t)
print('CFL is', str(4*dt/dx))

C = np.zeros((len(t),len(x)))
C[0] = cti
for n in np.arange(1,len(t)):#range(len(t)-1):
    A = np.zeros((len(x),len(x)))
    B = np.zeros(len(x))
    d = D*dt/(2*dx**2)
    B[0] = 0
    B[-1] = 0
    A[0][0] = 1
    A[-1][-1] = 1
```

```python
    #print(C[0])
    for i in np.arange(1,len(B)-1):
        A[i,i-1] = -d
        A[i,i] = 1+2*d
        A[i,i+1] = -d
        B[i] = d*C[n-1][i-1] + (1-2*d)*C[n-1][i] + d*C[n-1][i+1]
    C[n] = la.solve(A,B)



# Analytical Solution
plt.plot(x, C[0],'g',label='Analytical t=2')
plt.plot(x, C[1], label = 'CN at t=3')
plt.plot(x, ctf, label='Analytical t=4')
plt.plot(x, C[-1],'r*', label='CN at t=4')
plt.legend()
plt.title('Diffusion using Crank-Nicolson using dt = 1')
plt.xlabel('x')
plt.ylabel('Concetration, C')
plt.show()
```

Python Code for 2

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Dec  5 16:33:14 2022

@author: jjser
"""

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

# Initializing Everything
ni = 21 # 20
nj = 21 # 20
x = np.zeros((ni,nj))
y = np.zeros((ni,nj))
dx = 2/(ni-1)
dy = 1/(nj-1)

def bound(arr):
    ni,nj = arr.shape
    # Create top and bottom to be dy/dx = 0
    for i in range(ni):
        arr[i,0] = arr[i,1]
    for i in range(ni):
        arr[i,-1] = arr[i,-2]
    # Initialize boundary condition f(0) = 0
    arr[0,:] = 0
    # Initialize boundary condition f(y) = y
    for j in range(nj):
        arr[-1,j] = y[-1,j]
    return arr
def grid():
    # Initializing the Grid x and y
    for i in np.arange(0,ni): # goes from 1 to ni
        for j in np.arange(0,nj): # goes from 1 to nj
            x[i,j] = (i) * 2.0 / (ni-1)
            y[i,j] = (j) * 1.0 / (nj-1)

def anal():
    u = np.zeros((ni,nj))
    for i in np.arange(0,ni): # goes from 1 to ni
        for j in np.arange(0,nj): # goes from 1 to nj
            u[i,j] = x[i,j] / 4
            for n in (np.arange(10) * 2 + 1):
                u[i,j] -= 4 * (np.sinh(n*math.pi*x[i,j]) * np.cos(n*math.pi*y[i,j])) / ( (n*
    math.pi)**2 * np.sinh(2*n*math.pi ) )
    return u
def jaco():
```

```
    TOL = 1e-6
    res = 1
    MAXITE = 3000
    ite = 0
    resp = []
    u = np.zeros((ni,nj))
    u = bound(u)
    uk = u.copy()
    uk1 = u.copy()
    while res > TOL and ite < MAXITE:
        ite += 1
        res = 0
        for j in np.arange(1,nj-1): # goes from 2 to nj-1
            for i in np.arange(1,ni-1): # goes from 2 to ni-1
                uk1[i,j] = ( (uk[i-1,j] + uk[i+1,j])/dx**2 + (uk[i,j-1] + uk[i,j+1])/dy**2 ) * (1 / (2/dx**2 + 2/dy**2) )
        # Compute Residual
        for i in np.arange(1,ni-1): # goes from 2 to ni-1
            for j in np.arange(1,nj-1): # goes from 2 to nj-1
                res += (uk1[i,j]-uk[i,j])**2
        res = math.sqrt(res / ((ni-2)*(nj-2)) )
        resp.append(res)
        uk = bound(uk1).copy()
    ufin = bound(uk1)
    return ufin, ite, resp
def gase():
    TOL = 1e-6
    res = 1
    MAXITE = 2000
    ite = 0
    resp = []
    u = np.zeros((ni,nj))
    u = bound(u)
    uk = u.copy()
    uk1 = u.copy()
    while res > TOL and ite < MAXITE:
        ite += 1
        res = 0
        for j in np.arange(1,nj-1): # goes from 2 to nj-1
            for i in np.arange(1,ni-1): # goes from 2 to ni-1
                uk1[i,j] = ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])/dy**2 ) * (1 / (2/dx**2 + 2/dy**2) )
        # Compute Residual
        for i in np.arange(1,ni-1): # goes from 2 to ni-1
            for j in np.arange(1,nj-1): # goes from 2 to nj-1
                res += (uk1[i,j]-uk[i,j])**2
        res = math.sqrt(res / ((ni-2)*(nj-2)) )
        resp.append(res)
        uk = bound(uk1).copy()
    ufin = bound(uk1)
    return ufin, ite, resp
def gase2():
    TOL = 1e-6
    res = 1
    MAXITE = 2000
    ite = 0
    resp = []
    u = np.zeros((ni,nj))
    u = bound(u)
    uk = u.copy()
    uk1 = u.copy()
    while res > TOL and ite < MAXITE:
        ite += 1
        res = 0
        for i in np.arange(1,ni-1): # goes from 2 to nj-1
            for j in np.arange(1,nj-1): # goes from 2 to ni-1
                uk1[i,j] = ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])/dy**2 ) * (1 / (2/dx**2 + 2/dy**2) )
        # Compute Residual
        for i in np.arange(1,ni-1): # goes from 2 to ni-1
            for j in np.arange(1,nj-1): # goes from 2 to nj-1
                res += (uk1[i,j]-uk[i,j])**2
```

```
        res = math.sqrt(res / ((ni-2)*(nj-2)) )
        resp.append(res)
        uk = bound(uk1).copy()
    ufin = bound(uk1)
    return ufin, ite, resp
def sor(w):
    TOL = 1e-6
    res = 1
    MAXITE = 2000
    ite = 0
    resp = []
    u = np.zeros((ni,nj))
    #u = np.random.rand(ni,nj)
    u = bound(u)
    uk = u.copy()
    uk1 = u.copy()
    while res > TOL and ite < MAXITE:
        ite += 1
        res = 0
        for j in np.arange(1,nj-1): # goes from 2 to nj-1
            for i in np.arange(1,ni-1): # goes from 2 to ni-1
                uk1[i,j] = w * ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])
    /dy**2 ) * (1 / (2/dx**2 + 2/dy**2) ) + (1-w)*uk[i,j]
        # Compute Residual
        for i in np.arange(1,ni-1): # goes from 2 to ni-1
            for j in np.arange(1,nj-1): # goes from 2 to nj-1
                res += (uk1[i,j]-uk[i,j])**2
        res = math.sqrt(res / ((ni-2)*(nj-2)) )
        #print(res)
        resp.append(res)
        uk = bound(uk1).copy()
    ufin = bound(uk1)
    return ufin, ite, resp

grid()

#### Analytical Solution
u = anal()
cs = plt.contourf(x,y,u)
plt.colorbar(cs)
plt.title('Analytical Solution')
plt.show()


#### Jacobi ####
u, ite, resp = jaco()
print(' #### Jacobi ####')
print('Iterations to Convergence:', str(ite))
print('Final residual: ', str(np.round(resp[-1],10)) )
cs = plt.contourf(x,y,u)
plt.colorbar(cs)
plt.title('Jacobi Solution')
plt.show()
plt.plot(np.arange(1,ite+1), resp)
plt.title('Jacobi Residuals')
plt.xlabel('Iterations')
plt.ylabel('Residual')
plt.yscale('log')
plt.show()


#### Gauss-Seidel ####
u, ite, resp = gase()
print(' #### Gauss-Seidel ####')
print('Iterations to Convergence:', str(ite))
print('Final residual: ', str(np.round(resp[-1],10)) )
cs = plt.contourf(x,y,u)
plt.colorbar(cs)
plt.title('Gauss-Seidel Solution')
plt.show()

plt.plot(np.arange(1,ite+1), resp)
```

```python
plt.title('Gauss-Seidel Residuals')
plt.xlabel('Iterations')
plt.ylabel('Residual')
plt.yscale('log')
plt.show()

#### Gauss-Seidel ####
u, ite, resp = gase2()
print(' #### Gauss-Seidel ####')
print('Iterations to Convergence:', str(ite))
print('Final residual: ', str(np.round(resp[-1],10)) )
cs = plt.contourf(x,y,u)
plt.colorbar(cs)
plt.title('Gauss-Seidel Solution')
plt.show()

plt.plot(np.arange(1,ite+1), resp)
plt.title('Gauss-Seidel Residuals')
plt.xlabel('Iterations')
plt.ylabel('Residual')
plt.yscale('log')
plt.show()


#### Successive-Over Relaxation ###
wlist = np.arange(0.1,2,0.01)
itesor = np.zeros(len(wlist))
for i, w in enumerate(wlist):
    u, ite, resp = sor(w)
    itesor[i] = ite
plt.plot(wlist,itesor)
plt.title('SOR Iterations')
plt.xlabel('w')
plt.ylabel('Iterations')
plt.show()
m = np.argmin(itesor)
mini = np.min(itesor)
print(' #### SOR ####')
print('w at Minimum Iterations:', str(int(mini)) )
print('Minimum Iterations: ', str(np.round(wlist[m],5)) )

u, ite, resp = sor( np.round(wlist[m],5) )
print(' #### SOR ####')
print('Iterations to Convergence:', str(ite))
print('Final residual: ', str(np.round(resp[-1],10)) )
cs = plt.contourf(x,y,u)
plt.colorbar(cs)
plt.title('SOR Solution')
plt.show()
plt.plot(np.arange(1,ite+1), resp)
plt.title('SOR Residuals')
plt.xlabel('Iterations')
plt.ylabel('Residual')
plt.yscale('log')
plt.show()
```