

## 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.

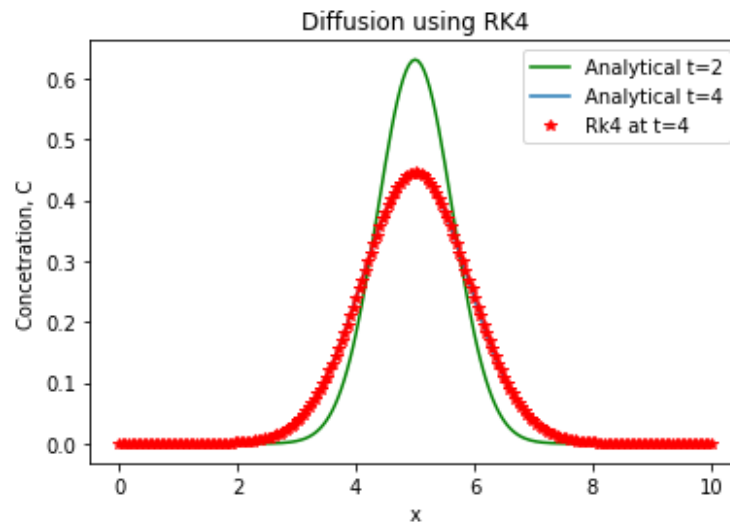


Figure 1: 1a) Diffusion Equation using Runge-Kutta 4

### 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  $\Delta t$ . The time step used below is  $\Delta t=1s$  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.

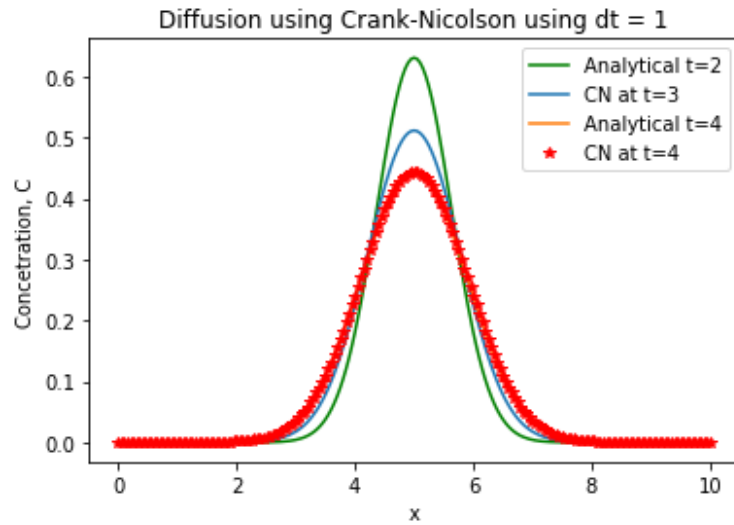


Figure 2: 1b) Diffusion Equation using Crank-Nicolson

## 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.

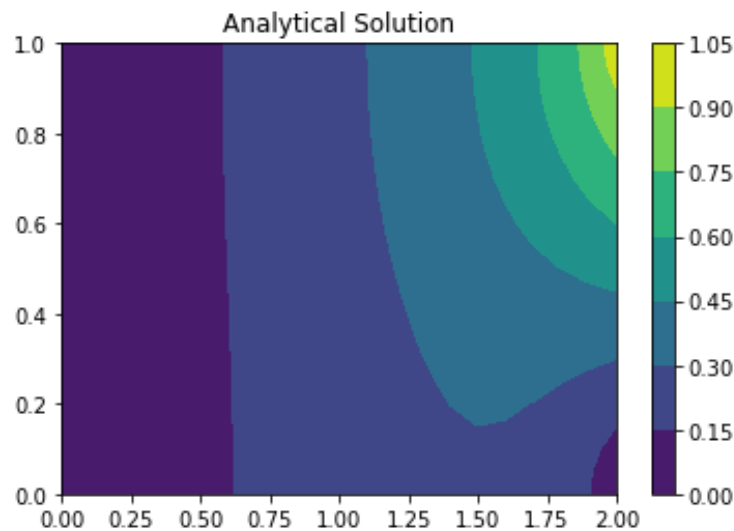


Figure 3: 2) Analytical Solution of the Laplace Equation

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.

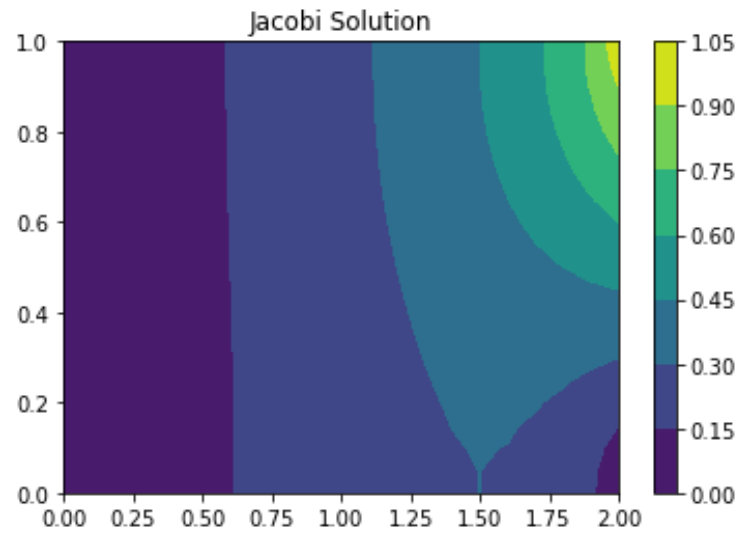


Figure 4: 2a) Laplace Equation using Jacobi

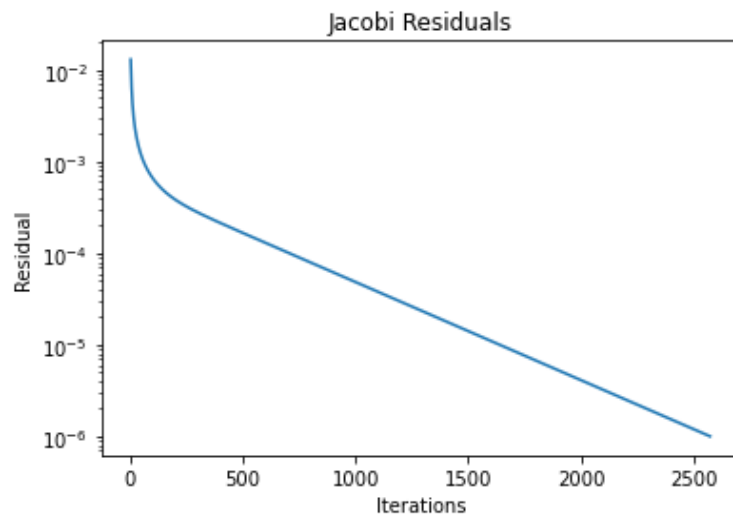


Figure 5: 2a) Residuals from Laplace Equation using Jacobi

## 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.

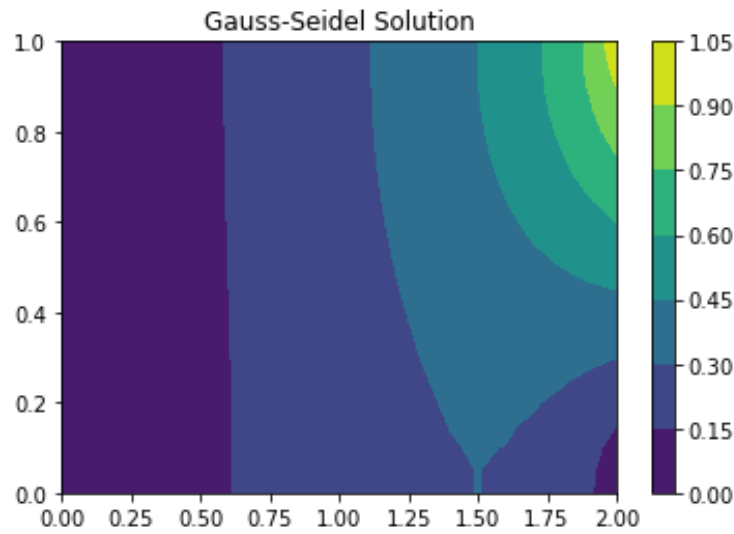


Figure 6: 2b) Laplace Equation using Gauss-Seidel

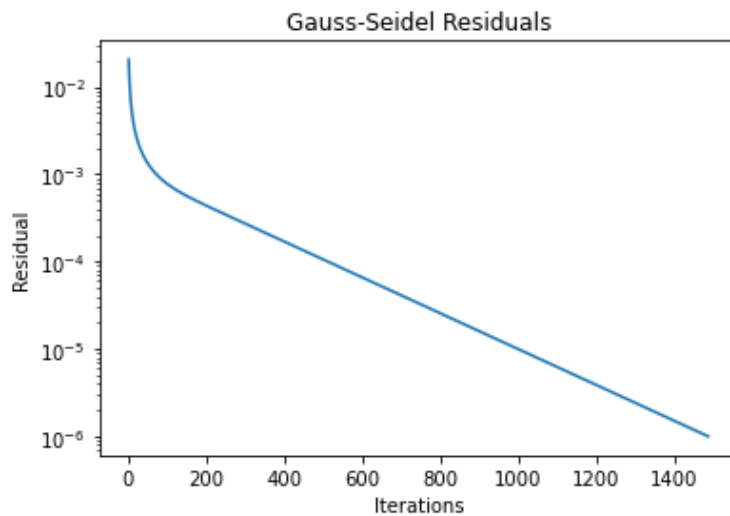


Figure 7: 2b) Residuals from Laplace Equation using Gauss-Seidel

## 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.

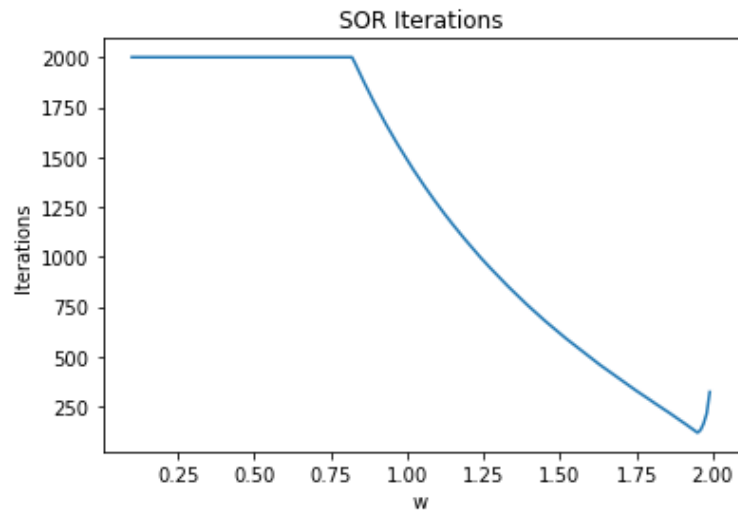


Figure 8: 2c) Effect of  $w$  on iterations using SOR

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

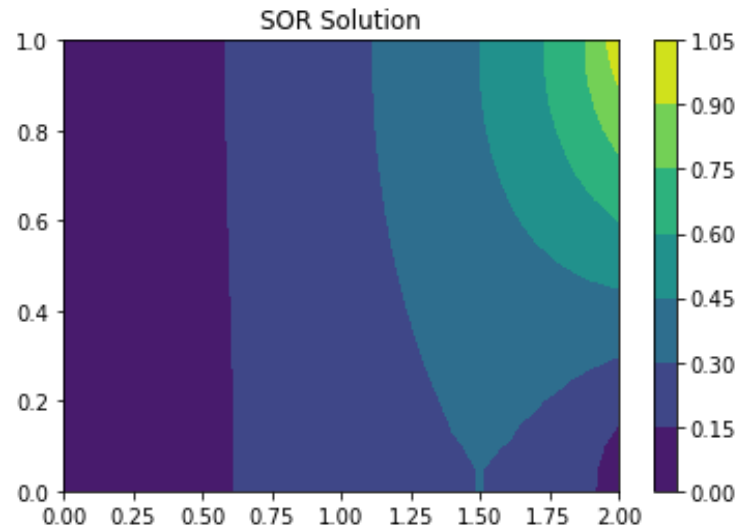


Figure 9: 2c) Laplace Equation using SOR

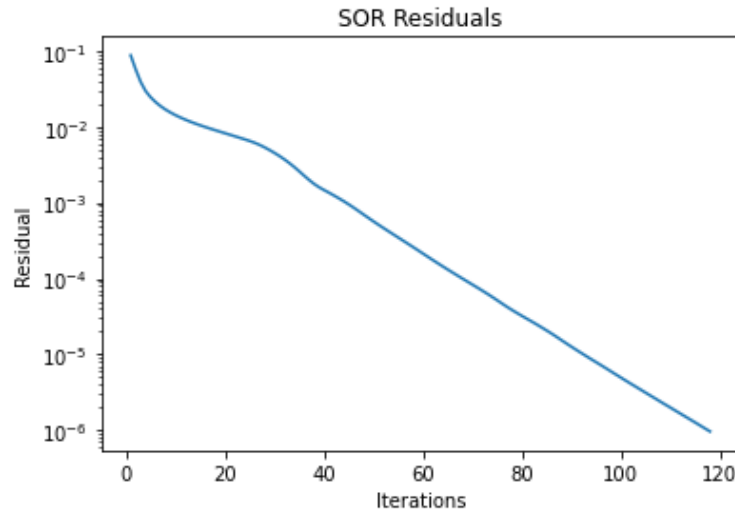


Figure 10: 2c) Residuals from Laplace Equation using SOR

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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numpy import linalg as la
4
5 x0 = 5
6 D = 0.1
7
8 def Canal(x,t):
9     return 1/(np.sqrt(4*np.pi*D*t)) * np.exp(-(x-x0)**2/(4*D*t))
10
11 N = 200
12
13 x = np.linspace(0,10,N+1)
14 dx = 10/N
15 #print(dx)
16 dt = np.round(dx**2/(2*D),10)
17 #dt = 0.0142 # hmm only stops working at 0.0142
18 print(dt)
19 numt = int(np.floor(2/dt))
20 #print(numt)
21 t = np.linspace(2,4,numt+1)
22 #print(t[0])
23 #print(t[1])
24 print('CFL is', str(4*dt/dx))

```

```

25
26 def bound(f):
27     f[0] = 0
28     f[-1] = 0
29     return f
30
31 cti = Canal(x,2)
32 bound(cti)
33 ctf = Canal(x,4)
34 bound(ctf)
35
36 ''' Right Hand Side Functions '''
37 def CS(f, n, i):
38     RHS = D * (f[n][i+1] - 2*f[n][i] + f[n][i-1]) / (dx)**2
39     return RHS
40 ''' Left Hand Side Functions '''
41 def Rk4(f, n, i):
42     k0 = CS(f,n,i)
43     k1 = CS(f,n,i) + dt/2 * k0
44     k2 = CS(f,n,i) + dt/2 * k1
45     k3 = CS(f,n,i) + dt * k2
46     df = (k0 + 2*k1 + 2*k2 + k3) / 6
47     LHS = f[n][i] + dt * df
48     return LHS
49
50
51 # Exact Solution at t=2 and t=4
52 fig, ax = plt.subplots()
53 ax.plot(x, cti)
54 ax.plot(x, ctf)
55 plt.show()
56
57 C = np.zeros((len(t),len(x)))
58 C[0] = cti
59 for n in range(len(t)-1):
60     for i in np.arange(N):
61         C[n+1][i] = Rk4(C, n, i)
62     C[n+1] = bound(C[n+1])
63
64 # Analytical Solution
65 plt.plot(x, C[0], 'g', label='Analytical t=2')
66 #ax.plot(x, C[40], 'g')
67 #ax.plot(x, C[80], 'g')
68 #ax.plot(x, C[120], 'g')
69 plt.plot(x, ctf, label='Analytical t=4')
70 plt.plot(x, C[-1], 'r*', label='Rk4 at t=4')
71 plt.legend()
72 plt.title('Diffusion using RK4')
73 plt.xlabel('x')
74 plt.ylabel('Concentration, C')
75 plt.show()
76
77
78 ##### Part 2 Crank Nicolson
79 dt = 1
80 # dt works even as high as 2
81 numt = int(np.floor(2/dt))
82 print(numt)
83 t = np.linspace(2,4,numt+1)
84 print(t)
85 print('CFL is', str(4*dt/dx))
86
87 C = np.zeros((len(t),len(x)))
88 C[0] = cti
89 for n in np.arange(1,len(t)):#range(len(t)-1):
90     A = np.zeros((len(x),len(x)))
91     B = np.zeros(len(x))
92     d = D*dt/(2*dx**2)
93     B[0] = 0
94     B[-1] = 0
95     A[0][0] = 1
96     A[-1][-1] = 1

```

```

97     #print(C[0])
98     for i in np.arange(1,len(B)-1):
99         A[i,i-1] = -d
100        A[i,i] = 1+2*d
101        A[i,i+1] = -d
102        B[i] = d*C[n-1][i-1] + (1-2*d)*C[n-1][i] + d*C[n-1][i+1]
103    C[n] = la.solve(A,B)
104
105
106
107 # Analytical Solution
108 plt.plot(x, C[0], 'g', label='Analytical t=2')
109 plt.plot(x, C[1], label='CN at t=3')
110 plt.plot(x, ctf, label='Analytical t=4')
111 plt.plot(x, C[-1], 'r*', label='CN at t=4')
112 plt.legend()
113 plt.title('Diffusion using Crank-Nicolson using dt = 1')
114 plt.xlabel('x')
115 plt.ylabel('Concentration, C')
116 plt.show()

```

## Python Code for 2

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Mon Dec 5 16:33:14 2022
4
5  @author: jjser
6  """
7
8  import numpy as np
9  import matplotlib.pyplot as plt
10 import math
11
12 # Initializing Everything
13 ni = 21 # 20
14 nj = 21 # 20
15 x = np.zeros((ni,nj))
16 y = np.zeros((ni,nj))
17 dx = 2/(ni-1)
18 dy = 1/(nj-1)
19
20 def bound(arr):
21     ni,nj = arr.shape
22     # Create top and bottom to be dy/dx = 0
23     for i in range(ni):
24         arr[i,0] = arr[i,1]
25     for i in range(ni):
26         arr[i,-1] = arr[i,-2]
27     # Initialize boundary condition f(0) = 0
28     arr[0,:] = 0
29     # Initialize boundary condition f(y) = y
30     for j in range(nj):
31         arr[-1,j] = y[-1,j]
32     return arr
33
34 def grid():
35     # Initializing the Grid x and y
36     for i in np.arange(0,ni): # goes from 1 to ni
37         for j in np.arange(0,nj): # goes from 1 to nj
38             x[i,j] = (i) * 2.0 / (ni-1)
39             y[i,j] = (j) * 1.0 / (nj-1)
40
41 def anal():
42     u = np.zeros((ni,nj))
43     for i in np.arange(0,ni): # goes from 1 to ni
44         for j in np.arange(0,nj): # goes from 1 to nj
45             u[i,j] = x[i,j] / 4
46             for n in (np.arange(10) * 2 + 1):
47                 u[i,j] -= 4 * (np.sinh(n*math.pi*x[i,j]) * np.cos(n*math.pi*y[i,j])) / ( (n*
48                     math.pi)**2 * np.sinh(2*n*math.pi) )
49     return u
50
51 def jaco():

```



```

49 TOL = 1e-6
50 res = 1
51 MAXITE = 3000
52 ite = 0
53 resp = []
54 u = np.zeros((ni,nj))
55 u = bound(u)
56 uk = u.copy()
57 uk1 = u.copy()
58 while res > TOL and ite < MAXITE:
59     ite += 1
60     res = 0
61     for j in np.arange(1,nj-1): # goes from 2 to nj-1
62         for i in np.arange(1,ni-1): # goes from 2 to ni-1
63             uk1[i,j] = ( (uk[i-1,j] + uk[i+1,j])/dx**2 + (uk[i,j-1] + uk[i,j+1])/dy**2 )
64             * (1 / (2/dx**2 + 2/dy**2))
65             # Compute Residual
66             for i in np.arange(1,ni-1): # goes from 2 to ni-1
67                 for j in np.arange(1,nj-1): # goes from 2 to nj-1
68                     res += (uk1[i,j]-uk[i,j])**2
69             res = math.sqrt(res / ((ni-2)*(nj-2)))
70             resp.append(res)
71             uk = bound(uk1).copy()
72 ufin = bound(uk1)
73 return ufin, ite, resp
74 def gase():
75     TOL = 1e-6
76     res = 1
77     MAXITE = 2000
78     ite = 0
79     resp = []
80     u = np.zeros((ni,nj))
81     u = bound(u)
82     uk = u.copy()
83     uk1 = u.copy()
84     while res > TOL and ite < MAXITE:
85         ite += 1
86         res = 0
87         for j in np.arange(1,nj-1): # goes from 2 to nj-1
88             for i in np.arange(1,ni-1): # goes from 2 to ni-1
89                 uk1[i,j] = ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])/dy
90                 **2 ) * (1 / (2/dx**2 + 2/dy**2))
91                 # Compute Residual
92                 for i in np.arange(1,ni-1): # goes from 2 to ni-1
93                     for j in np.arange(1,nj-1): # goes from 2 to nj-1
94                         res += (uk1[i,j]-uk[i,j])**2
95                 res = math.sqrt(res / ((ni-2)*(nj-2)))
96                 resp.append(res)
97                 uk = bound(uk1).copy()
98 ufin = bound(uk1)
99 return ufin, ite, resp
100 def gase2():
101     TOL = 1e-6
102     res = 1
103     MAXITE = 2000
104     ite = 0
105     resp = []
106     u = np.zeros((ni,nj))
107     u = bound(u)
108     uk = u.copy()
109     uk1 = u.copy()
110     while res > TOL and ite < MAXITE:
111         ite += 1
112         res = 0
113         for i in np.arange(1,ni-1): # goes from 2 to nj-1
114             for j in np.arange(1,nj-1): # goes from 2 to ni-1
115                 uk1[i,j] = ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])/dy
116                 **2 ) * (1 / (2/dx**2 + 2/dy**2))
117                 # Compute Residual
118                 for i in np.arange(1,ni-1): # goes from 2 to ni-1
119                     for j in np.arange(1,nj-1): # goes from 2 to nj-1
120                         res += (uk1[i,j]-uk[i,j])**2

```

```

118         res = math.sqrt(res / ((ni-2)*(nj-2)) )
119         resp.append(res)
120         uk = bound(uk1).copy()
121     ufin = bound(uk1)
122     return ufin, ite, resp
123 def sor(w):
124     TOL = 1e-6
125     res = 1
126     MAXITE = 2000
127     ite = 0
128     resp = []
129     u = np.zeros((ni,nj))
130     #u = np.random.rand(ni,nj)
131     u = bound(u)
132     uk = u.copy()
133     uk1 = u.copy()
134     while res > TOL and ite < MAXITE:
135         ite += 1
136         res = 0
137         for j in np.arange(1,nj-1): # goes from 2 to nj-1
138             for i in np.arange(1,ni-1): # goes from 2 to ni-1
139                 uk1[i,j] = w * ( (uk1[i-1,j] + uk1[i+1,j])/dx**2 + (uk1[i,j-1] + uk1[i,j+1])
140                 /dy**2 ) * (1 / (2/dx**2 + 2/dy**2) ) + (1-w)*uk[i,j]
141             # Compute Residual
142             for i in np.arange(1,ni-1): # goes from 2 to ni-1
143                 for j in np.arange(1,nj-1): # goes from 2 to nj-1
144                     res += (uk1[i,j]-uk[i,j])**2
145             res = math.sqrt(res / ((ni-2)*(nj-2)) )
146             #print(res)
147             resp.append(res)
148             uk = bound(uk1).copy()
149         ufin = bound(uk1)
150         return ufin, ite, resp
151 grid()
152
153 ##### Analytical Solution
154 u = anal()
155 cs = plt.contourf(x,y,u)
156 plt.colorbar(cs)
157 plt.title('Analytical Solution')
158 plt.show()
159
160
161 ##### Jacobi #####
162 u, ite, resp = jaco()
163 print(' ##### Jacobi #####')
164 print('Iterations to Convergence:', str(ite))
165 print('Final residual: ', str(np.round(resp[-1],10)) )
166 cs = plt.contourf(x,y,u)
167 plt.colorbar(cs)
168 plt.title('Jacobi Solution')
169 plt.show()
170 plt.plot(np.arange(1,ite+1), resp)
171 plt.title('Jacobi Residuals')
172 plt.xlabel('Iterations')
173 plt.ylabel('Residual')
174 plt.yscale('log')
175 plt.show()
176
177
178 ##### Gauss-Seidel #####
179 u, ite, resp = gase()
180 print(' ##### Gauss-Seidel #####')
181 print('Iterations to Convergence:', str(ite))
182 print('Final residual: ', str(np.round(resp[-1],10)) )
183 cs = plt.contourf(x,y,u)
184 plt.colorbar(cs)
185 plt.title('Gauss-Seidel Solution')
186 plt.show()
187
188 plt.plot(np.arange(1,ite+1), resp)

```

```

189 plt.title('Gauss-Seidel Residuals')
190 plt.xlabel('Iterations')
191 plt.ylabel('Residual')
192 plt.yscale('log')
193 plt.show()
194
195 ##### Gauss-Seidel #####
196 u, ite, resp = gase2()
197 print(' ##### Gauss-Seidel #####')
198 print('Iterations to Convergence:', str(ite))
199 print('Final residual: ', str(np.round(resp[-1],10)) )
200 cs = plt.contourf(x,y,u)
201 plt.colorbar(cs)
202 plt.title('Gauss-Seidel Solution')
203 plt.show()
204
205 plt.plot(np.arange(1,ite+1), resp)
206 plt.title('Gauss-Seidel Residuals')
207 plt.xlabel('Iterations')
208 plt.ylabel('Residual')
209 plt.yscale('log')
210 plt.show()
211
212
213 ##### Successive-Over Relaxation ###
214 wlist = np.arange(0.1,2,0.01)
215 itesor = np.zeros(len(wlist))
216 for i, w in enumerate(wlist):
217     u, ite, resp = sor(w)
218     itesor[i] = ite
219 plt.plot(wlist, itesor)
220 plt.title('SOR Iterations')
221 plt.xlabel('w')
222 plt.ylabel('Iterations')
223 plt.show()
224 m = np.argmin(itesor)
225 mini = np.min(itesor)
226 print(' ##### SOR #####')
227 print('w at Minimum Iterations:', str(int(mini)) )
228 print('Minimum Iterations: ', str(np.round(wlist[m],5)) )
229
230 u, ite, resp = sor( np.round(wlist[m],5) )
231 print(' ##### SOR #####')
232 print('Iterations to Convergence:', str(ite))
233 print('Final residual: ', str(np.round(resp[-1],10)) )
234 cs = plt.contourf(x,y,u)
235 plt.colorbar(cs)
236 plt.title('SOR Solution')
237 plt.show()
238 plt.plot(np.arange(1,ite+1), resp)
239 plt.title('SOR Residuals')
240 plt.xlabel('Iterations')
241 plt.ylabel('Residual')
242 plt.yscale('log')
243 plt.show()

```