

## Computational Physics Lab

# Homework 4

108000204

Yuan-Yen Peng

*Dept. of Physics, NTHU*

*Hsinchu, Taiwan*

December 29, 2022

## 1 Writing Assignments

### 1.1 Finite difference method for solving 2D Poisson equation

The  $3 \times 3$  grids with grid space  $h$  and the source term  $\rho(x, y)$  can be described as:

$$u_{xx} + u_{yy} = \rho(x, y)$$

where the subscripts mean second partial derivatives of the  $u$  with  $x$  and  $y$ , respectively. Besides, we annotate  $(x, y)$  with  $(i, j)$  where  $i$  and  $j$  are from  $0 \sim (3 + 1)$  (including boundaries, which are  $(i, j) = (i, 0), (i, 4), (0, j),$  and  $(4, j)$ ), and implement the Euler method for derivative; then we can get:

$$\begin{aligned} & \frac{\partial}{\partial x} \left( \frac{u_{i,j} - u_{i-1,j}}{h} \right) + \frac{\partial}{\partial y} \left( \frac{u_{i,j} - u_{i,j-1}}{h} \right) = \rho(x, y) \\ \Rightarrow & \frac{1}{h} \left( \frac{u_{i+1,j} - u_{i,j}}{h} - \frac{u_{i,j} - u_{i-1,j}}{h} \right) + \frac{1}{h} \left( \frac{u_{i,j+1} - u_{i,j}}{h} - \frac{u_{i,j} - u_{i,j-1}}{h} \right) = \rho(x, y) \\ \Rightarrow & (u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + (u_{i,j+1} - 2u_{i,j} + u_{i,j-1}) = \rho(x, y)h^2 \\ \Rightarrow & 4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} = \rho(x, y)h^2 \end{aligned} \quad (I)$$

In the wake of knowing the general form of the solution, we apply the boundary conditions; here boundaries are all zeros (i.e.,  $u_{0,j}, u_{i,0}, u_{4,j},$  and  $u_{i,4} = 0$ ). Exploiting all  $i$  and  $j$ , we can get LHS of eq.(I) in the matrix form  $\mathbf{A}$  and RHS in the vector form  $\mathbf{b}$  with source term  $g_{ij}$  which represents the splitting of the source function  $\rho(x, y)$ :

$$\mathbf{A} = \begin{bmatrix} \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} & \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} & \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} & \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} & \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix} \end{bmatrix}$$

and

$$\mathbf{b} = -h^2 \begin{bmatrix} g_{11} \\ g_{12} \\ g_{13} \\ - - - \\ g_{21} \\ g_{22} \\ g_{23} \\ - - - \\ g_{31} \\ g_{32} \\ g_{33} \end{bmatrix}$$

## 2 Programming Assignments

### 2.1 $\rho_{22} = 1$ and else are zeros

Homework's figure in the last problem is  $4 \times 4$ , but the question asks the  $3 \times 3$  matrix. Thus, I generate the two results in Figure 1, the left one uses the definition of the left bottom corner, and the other (right) implements the center of the grid as a benchmark.

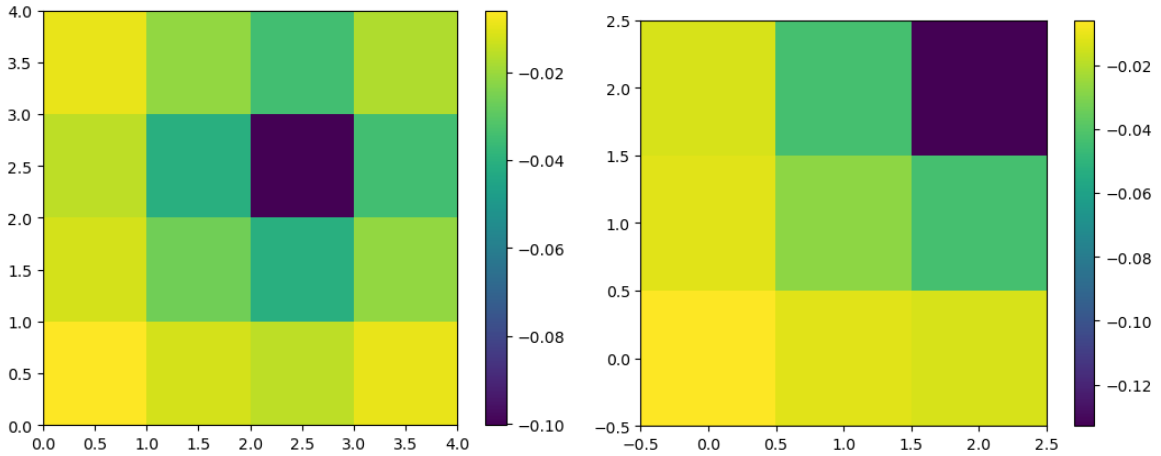


Figure 1: These figures are the solution of the potential  $u$ . The left is the  $4 \times 4$  grid; the right is the  $3 \times 3$  grid. Both of them are with the source  $\rho_{22} = 1$  and the others are zeros.

### 2.2 2D Poisson's equation with a given source with periodic boindary

In this subsection (also the below's subsections), we exploit the finite difference method with the sparse matrix to solve the equation:

$$\rho(x, y) = e^{-\frac{5}{4}r_1^2} + \frac{3}{2} \times e^{-r_2^2}$$

with the domain  $\mathcal{D}\{[-5 < x < 5] \times [-5 < y < 5]\}$  in the  $128 \times 128$  grid.

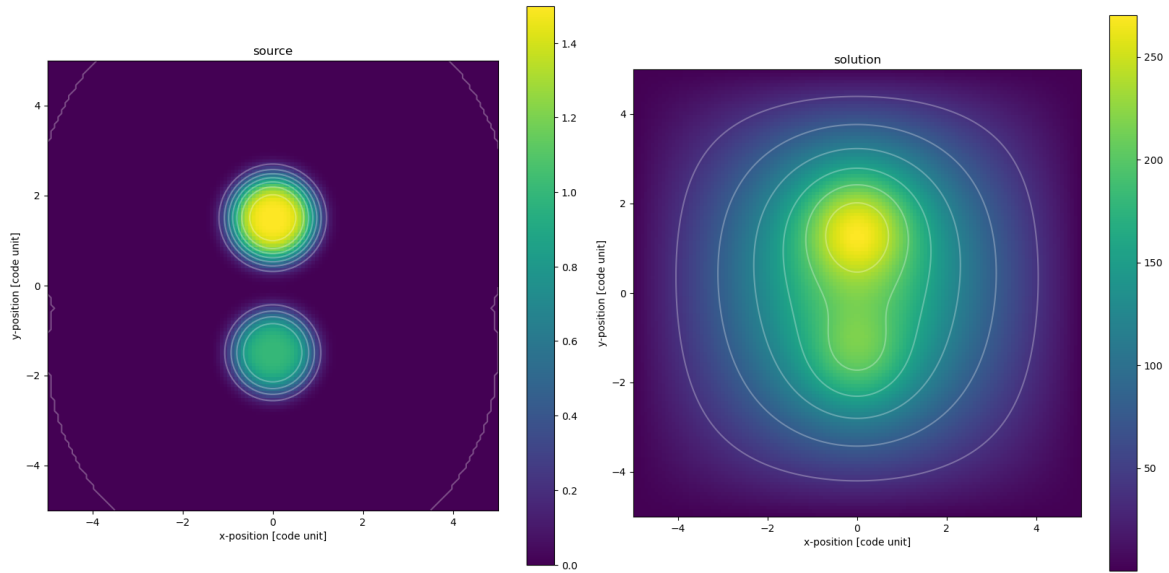


Figure 2: The left is the source function with the scheme with periodic boundary; the right is the solution of the corresponding potential for this Poisson's equation also with the periodic boundary exploiting the sparse matrix method. Moreover, this “periodic boundary” will be updated in each run during finite difference algorithm execution.

### 2.3 Error convergence comparison between different algorithms

We utilize three different methods learned in the lecture to investigate the error convergence of this Poisson's equation. The first method we used is the Jacobi method and the second is the Gauss-Seidel method, and the last method is the successive over-relaxation method with  $w = 1.2, 1.5$ , and  $2.0$ . However, in SOR (successive over-relaxation method), it might be “diverge”, so we plot two schemes to research the convergence rate, one is all converge and the other is one of them diverge, please see in Figure3.

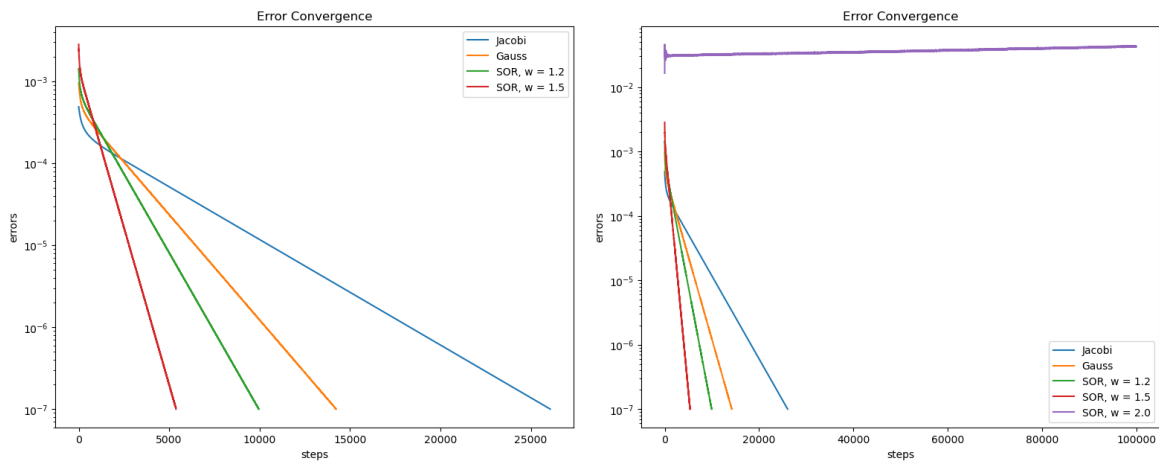


Figure 3: These two figures have the log-scale y-axis; normal scale x-axis so as to show errors as functions of iterations. The left is all of the methods are converged; the right scheme is the one diverge situation ( $w = 2.0$ , purple line).

It is manifestly that the Jacobi method is the slowest one and then is the Gauss-Seidel method, and the fastest is the SOR method. In this scenario, although SOR is the fastest when  $w = 1.5$ , it will diverge with  $w = 2.0$  (Figure3-right)! The fastest (SOR with  $w = 1.5$ ) is approximately 4.5 times faster than the slowest (Jacobi) as the error tolerance is  $\sim 10^{-6}$ .

## 2.4 Resolution vs time with different methodologies

In this subsection, we will discuss different grid sizes; that is, different resolutions ( $32 \times 32$ ,  $64 \times 64$ , and  $128 \times 128$ ) with the time. Additionally, we exploit the methods of the sparse matrix, Jacobi, Gauss-Seidel, and SOR with  $w = 1.2$ ,  $1.5$  as well. Setting tolerance  $\epsilon = 10^{-7}$ , the resolutions vs time show in Figure4.

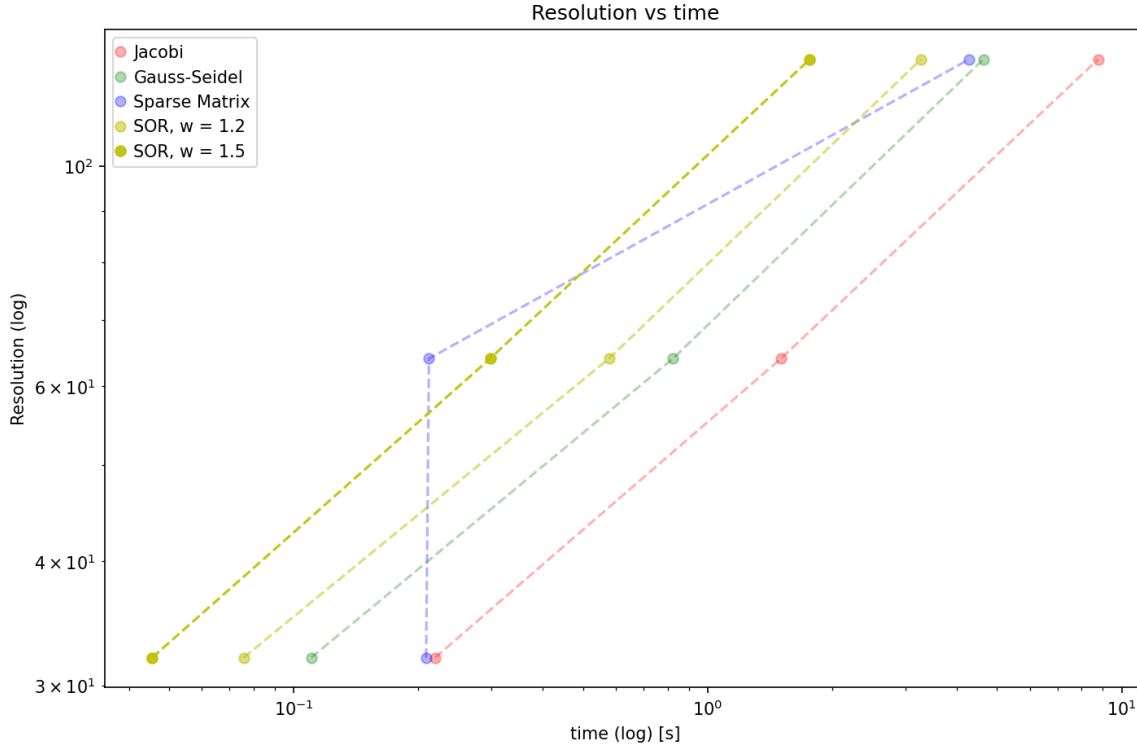


Figure 4: This figure shows different algorithms to solve Poisson’s equation in (subsection 2.2). The tendency of these methods, excluding sparse matrix, require more time to iterate to the tolerance ( $\epsilon = 10^{-7}$ ). The outcomes are the same consequent analysis in the aforementioned subsections; i.e., Jacobi is the slowest, Gauss-Seidel is the middle and the SOR is the fastest (besides,  $w = 1.5$  is more efficient than  $w = 1.3$ ). On the other hand, the behavior of the sparse matrix method does not act the same as others. It takes the least time among others when the resolution is  $64 \times 64$ . Likewise, in low resolution ( $32 \times 32$ ), the sparse matrix method is only faster than Jacobi, but in the largest resolution ( $128 \times 128$ ), it drills faster than Jacobi and also the Gauss-Seidel. Here, we can briefly sum up, if the size (resolution) is in “some proper range”, such as  $64 \times 64$ , we can adopt scipy’s sparse solver! It is more well organized than others.

## 3 Codes

All the codes are transferred from jupyterlab or python codes; hence, if you want to re-run them, please see the source code in the attached files or my GitHub repository:

<https://github.com/gary20000915/Comphyslab-HW4.git>

### 3.1 solvers.py

```
1 import numpy as np
2 from numba import njit, prange
3
```

```

4      from .mesh import Mesh2D
5
6      """
7      Solver to solve for Laplace/Poisson's equation
8
9      """
10
11     @njit(parallel=True)
12     def generate_g(g, buff_size, nx, ny, x, y):
13         for i in prange(nx+2*buff_size):
14             for j in prange(ny+2*buff_size):
15                 r1 = np.square(x[i] + 1.5) + np.square(y[j])
16                 r2 = np.square(x[i] - 1.5) + np.square(y[j])
17                 g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                     np.exp(-np.square(r2))
18
19         return g
20
21     def set_boundary(g, x, y, nx, ny, buff_size, mesh: Mesh2D):
22         generate_g(g, buff_size, nx, ny, x, y)
23         UL = g[-1,:]
24         LR = g[:, -1]
25         boundary = np.array([[UL], [LR], [UL], [LR]])
26         mesh[0, :] = boundary[0]
27         mesh[ny+buff_size, :] = boundary[1]
28         mesh[:, 0] = boundary[2]
29         mesh[:, nx+buff_size] = boundary[3]
30
31
32     @njit(parallel = True)
33     def j_kernel(u, u_temp, x, y, nx, ny, g, buff_size):
34         for i in prange(1, nx + 2*buff_size - 1, 1):
35             for j in prange(1, ny + 2*buff_size - 1, 1):
36                 r1 = np.square(x[i] + 1.5) + np.square(y[j])
37                 r2 = np.square(x[i] - 1.5) + np.square(y[j])
38                 g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                     np.exp(-np.square(r2))
39                 u[i, j] = 0.25 * (u_temp[i+1, j] + u_temp[i, j+1] +
                     u_temp[i-1, j] + u_temp[i, j-1] + g[i, j])
40         return u
41
42     @njit(parallel = True)
43     def gs_kernel(u, x, y, nx, ny, g, buff_size):
44         for i in prange(1, nx + 2*buff_size - 1, 1):
45             for j in prange(1, ny + 2*buff_size - 1, 1):
46                 r1 = np.square(x[i] + 1.5) + np.square(y[j])
47                 r2 = np.square(x[i] - 1.5) + np.square(y[j])
48                 g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                     np.exp(-np.square(r2))
49                 u[i, j] = 0.25 * (u[i+1, j] + u[i, j+1] + u[i-1, j] + u[i,
                     j-1] + g[i, j])
50         return u
51

```

```

52 @njit(parallel = True)
53 def SOR_kernel(u, u_temp, x, y, nx, ny, g, buff_size, w):
54     for i in prange(1, int(nx + 2*buff_size - 1), 1):
55         for j in prange(1, int(ny + 2*buff_size - 1), 1):
56             r1 = np.square(x[i] + 1.5) + np.square(y[j])
57             r2 = np.square(x[i] - 1.5) + np.square(y[j])
58             g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                    np.exp(-np.square(r2))
59             u[i, j] = 0.25 * (u[i+1, j] + u[i, j+1] + u[i-1, j] + u[i,
                    j-1] + g[i, j])
60             u[i, j] = (1-w) * u_temp[i, j] + w * u[i, j]
61     return u
62
63 def solve(name, tor, mesh: Mesh2D, **kwargs):
64     u = mesh.get_mesh()
65     x = mesh.get_x()
66     y = mesh.get_y()
67     nx = mesh.get_nx()
68     ny = mesh.get_ny()
69     g = mesh.get_xx()
70     buff_size = mesh.get_buff_size()
71
72     err = 10
73     err_arr = np.array([])
74     n = 0
75
76     while err > tor:
77         u_temp = np.copy(u)
78         set_boundary(g, x, y, nx, ny, buff_size, u)
79
80         if name == "Jacobi":
81             u = j_kernel(u, u_temp, x, y, nx, ny, g, buff_size)
82         elif name == "Gauss":
83             u = gs_kernel(u, x, y, nx, ny, g, buff_size)
84         elif name == "SOR":
85             u = SOR_kernel(u, u_temp, x, y, nx, ny, g, buff_size,
                    kwargs['w'])
86         else:
87             print("Error: unknown kernel!")
88             break
89
90     err = np.sqrt(np.sum(np.square(u - u_temp))) / (nx * ny)
91     err_arr = np.append(err_arr, err)
92     n += 1
93     # check
94     # if n % 100 == 0:
95     #     print(err, tor)
96     #     print(u)
97     #     plt.imshow(u.reshape(nx+2*buff_size, ny+2*buff_size), origin
98     #                 = 'lower', extent=[-1, 1, -1, 1])
99     #     plt.colorbar()
100    #     plt.contour(u, colors = 'white', extent=[-1, 1, -1, 1])
101    if n == 1e5:

```

```

101         break
102
103     return u.reshape(nx+2*buff_size, ny+2*buff_size), err_arr, n
104
105
106
107 if __name__=='__main__':
108
109     nx, ny = 4, 4
110     buff_size=1
111     tor = 1e-10
112     # boundary = np.zeros((4, nx + 2*buff_size))
113     # boundary[0] =np.ones(nx + 2*buff_size)
114     mesh = Mesh2D(nx = nx, ny = ny, buff_size=buff_size)
115
116     u = solve("Jacobi", tor, mesh)[1]
117     print(u)
118     print("TEST")

```

---

## 3.2 mesh.py

---

```

1     """
2     This file define classes for generating 2D meshes.
3
4     """
5     import numpy as np
6     from numba import jit, int32, float64
7     from numba.experimental import jitclass
8
9     class Mesh2D:
10         def __init__(self, nx:int = 10, ny:int = 10, buff_size = 1, xmin = 0,
11             xmax = 1, ymin = 0, ymax = 1):
12             self._nx = nx
13             self._ny = ny
14             self._nbuff = buff_size
15             self._xmin = xmin
16             self._xmax = xmax
17             self._ymin = ymin
18             self._ymax = ymax
19
20             self._setup()
21
22             return
23
24         def _setup(self):
25             self._dx = (self._xmax - self._xmin) / (self._nx + 1)
26             self._dy = (self._ymax - self._ymin) / (self._ny + 1)
27
28             self._istart = self._nbuff
29             self._istartGC = 0
30             self._iend = self._nbuff + self._nx - 1
31             self._iendGC = 2 * self._nbuff + self._nx - 1

```

```

31         self._nxGC = 2 * self._nbuff + self._nx
32
33         self._jstart = self._nbuff
34         self._jstartGC = 0
35         self._jend = self._nbuff + self._ny - 1
36         self._jendGC = 2 * self._nbuff + self._ny - 1
37         self._nyGC = 2 * self._nbuff + self._ny
38
39         x = np.linspace(self._xmin, self._xmax, self._nxGC)
40         y = np.linspace(self._ymin, self._ymax, self._nyGC)
41         xx, yy = np.meshgrid(x, y, indexing='ij')
42         self._mesh = xx * 0
43         self._xx = xx
44         self._yy = yy
45         self._x = x
46         self._y = y
47
48         return
49
50     def get_nx(self):
51         return int(self._nx)
52     def set_nx(self, nx):
53         self._nx = nx
54         self._setup()
55         return
56
57     def get_ny(self):
58         return int(self._ny)
59     def set_ny(self, ny):
60         self._ny = ny
61         self._setup()
62         return
63
64     def get_buff_size(self):
65         return int(self._nbuff)
66     def set_buff_size(self, buff_size):
67         self._nbuff = buff_size
68         self._setup()
69         return
70
71     def get_xmin(self):
72         return self._xmin
73     def set_xmin(self, xmin):
74         self._xmin = xmin
75         self._setup()
76         return
77
78     def get_xmax(self):
79         return self._xmax
80     def set_xmax(self, xmax):
81         self._xmax = xmax
82         self._setup()
83         return

```



```

84
85     def get_ymin(self):
86         return self._ymin
87     def set_ymin(self, ymin):
88         self._ymin = ymin
89         self._setup()
90         return
91
92     def get_ymax(self):
93         return self._ymax
94     def set_ymax(self, ymax):
95         self._ymax = ymax
96         self._setup()
97         return
98
99     def get_istart(self):
100         return self._istart
101     def set_istart(self, istart):
102         self._istart = istart
103         self._setup()
104         return
105
106     def get_istartGC(self):
107         return self._istartGC
108     def set_istartGC(self, istartGC):
109         self._istartGC = istartGC
110         self._setup()
111         return
112
113     def get_iend(self):
114         return self._iend
115     def set_iend(self, iend):
116         self._iend = iend
117         self._setup()
118         return
119
120     def get_iendGC(self):
121         return self._iendGC
122     def set_iendGC(self, iendGC):
123         self._iendGC = iendGC
124         self._setup()
125         return
126
127     def get_jstart(self):
128         return self._jstart
129     def set_jstart(self, jstart):
130         self._jstart = jstart
131         self._setup()
132         return
133
134     def get_jstartGC(self):
135         return self._jstartGC
136     def set_jstartGC(self, jstartGC):

```

```

137         self._jstartGC = jstartGC
138         self._setup()
139         return
140
141     def get_jend(self):
142         return self._jend
143     def set_jend(self, jend):
144         self._jend = jend
145         self._setup()
146         return
147
148     def get_jendGC(self):
149         return self._jendGC
150     def set_jendGC(self, jendGC):
151         self._jendGC = jendGC
152         self._setup()
153         return
154
155     def get_mesh(self):
156         return self._mesh
157     def set_mesh(self, mesh):
158         self._mesh = mesh
159         if (mesh.size != self._mesh.size):
160             print("error! size conflict!")
161         return
162
163     def get_xx(self):
164         return self._xx
165     def set_xx(self, xx):
166         self._xx = xx
167         self._setup()
168         return
169
170     def get_yy(self):
171         return self._yy
172     def set_yy(self, yy):
173         self._yy = yy
174         self._setup()
175         return
176
177     def get_x(self):
178         return self._x
179     def set_x(self, x):
180         self._x = x
181         self._setup()
182         return
183
184     def get_y(self):
185         return self._y
186     def set_y(self, y):
187         self._y = y
188         self._setup()
189         return

```

```

190
191     def get_nx(self):
192         return self._nx
193     def set_nx(self, nx):
194         self._nx = nx
195         self._setup()
196         return
197
198     def get_ny(self):
199         return self._ny
200     def set_ny(self, ny):
201         self._ny = ny
202         self._setup()
203         return
204
205
206 if __name__ == '__main__':
207     mesh = Mesh2D(nx = 3, ny = 3, buff_size=1)
208     # mesh.set_nx(32)
209     # mesh.set_ny(32)
210
211     u = mesh.get_mesh()
212     nx = mesh.get_nx()
213     ny = mesh.get_ny()
214     buff = mesh.get_buff_size()
215
216     print(u)
217     print(f"Testing ... nx={nx}, ny={ny}, buff = {buff}")
218     print('Done')

```

---

### 3.3 1\_finite-difference.ipynb

---

```

1     # %% [markdown]
2     # ### 1. Finite Difference method for solving discrete Laplace Equation
3
4     # %%
5     %reset -f
6
7     import numpy as np
8     import matplotlib.pyplot as plt
9     from scipy import linalg
10    from scipy.sparse import csc_matrix
11    from scipy.sparse import dia_array
12    from scipy.sparse import dia_matrix
13    import scipy.sparse.linalg as splinalg
14    from numba import njit, prange
15
16    # %%
17    N = 4
18    min, max = -1, 1
19    dx = (max - min) / N
20

```

```

21 # %%
22 def generate_D(n):
23     ex = np.ones(n)
24     data = np.array([-1 * ex, 4 * ex, -1 * ex])
25     offsets = np.array([-1, 0, 1])
26
27     return dia_matrix((data, offsets), shape=(n, n)).toarray()
28
29 # %%
30 @njit(parallel = True)
31 def kernel(N, A, D, I):
32     for i in prange(N):
33         for j in prange(N):
34             if i == j:
35                 for ii in prange(N):
36                     for jj in prange(N):
37                         A[ii+N*i, jj+N*j] = D[ii, jj]
38
39             if np.abs(i - j) == 1:
40                 for ii in prange(N):
41                     for jj in prange(N):
42                         A[ii+N*i, jj+N*j] = I[ii, jj]
43
44     return A
45
46 # %%
47 def generate_A(N):
48     A = np.zeros((N ** 2, N ** 2))
49     D = generate_D(N)
50     I = -np.identity(N)
51     A = kernel(N, A, D, I)
52     return A
53
54 # %%
55 def generate_b(N, dx):
56     b = np.zeros(N ** 2).reshape(N, N)
57     b[2, 2] = -1 * np.square(dx)
58     plt.imshow(b, origin = 'lower')
59     plt.colorbar()
60     return b.reshape(N ** 2)
61
62 # %%
63 # solve
64 def solve(N, dx):
65     a = generate_A(N)
66     a = csc_matrix(a) # transform to the fitting matrix
67     b = generate_b(N, dx)
68     x = splinalg.spsolve(a, b).reshape(N, N)
69     return x
70
71 # %%
72 u = solve(N, dx)
73 print(u)

```

```

74
75     # %%
76     plt.imshow(u, origin = 'lower', extent = [0,4,0,4])
77     plt.colorbar()

```

---

### 3.4 2\_finite\_difference.ipynb

---

```

1     # %% [markdown]
2     # ### Finite Difference method for solving discrete Laplace Equation
3
4     # %%
5     %reset -f
6
7     import numpy as np
8     import matplotlib.pyplot as plt
9     from matplotlib.pyplot import figure
10    from scipy import linalg
11    from scipy.sparse import csc_matrix
12    from scipy.sparse import dia_array
13    from scipy.sparse import dia_matrix
14    import scipy.sparse.linalg as splinalg
15    from numba import njit, prange
16
17    # %%
18    N = 128
19    min, max = -5, 5
20    dx = (max - min) / N
21
22    x = np.linspace(-5, 5, N)
23    y = np.linspace(-5, 5, N)
24    g = np.zeros([N, N])
25
26    # %%
27    def generate_D(n):
28        ex = np.ones(n)
29        data = np.array([-1 * ex, 4 * ex, -1 * ex])
30        offsets = np.array([-1, 0, 1])
31
32        return dia_matrix((data, offsets), shape=(n, n)).toarray()
33
34    # %%
35    @njit
36    def kernel(N, A, D, I):
37        for i in prange(N):
38            for j in prange(N):
39                if i == j:
40                    for ii in prange(N):
41                        for jj in prange(N):
42                            A[ii+N*i, jj+N*j] = D[ii, jj]
43
44                if np.abs(i - j) == 1:
45                    for ii in prange(N):

```

```

46         for jj in prange(N):
47             A[ii+N*i, jj+N*j] = I[ii, jj]
48
49     return A
50
51 # %%
52 def generate_A(N):
53     A = np.zeros((N ** 2, N ** 2))
54     D = generate_D(N)
55     I = -np.identity(N)
56     A = kernel(N, A, D, I)
57     return A
58
59 # %%
60 @njit
61 def generate_g(g, N, x, y):
62     for i in prange(N):
63         for j in prange(N):
64             r1 = np.square(x[i] + 1.5) + np.square(y[j])
65             r2 = np.square(x[i] - 1.5) + np.square(y[j])
66             g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                np.exp(-np.square(r2))
67
68     return g
69
70 # %%
71 ans = generate_g(g, N, x, y)
72
73 figure(figsize=(10, 10), dpi=100)
74 plt.imshow(ans, origin = 'lower', extent=[-5, 5, -5, 5])
75 plt.colorbar()
76 plt.contour(ans, colors = 'w', alpha = .3, extent=[-5, 5, -5, 5])
77 plt.title('source')
78 plt.xlabel('x-position [code unit]')
79 plt.ylabel('y-position [code unit]')
80 plt.show()
81 plt.close()
82
83 # %%
84 def generate_b(N, g, dx):
85     # boundaries
86     # b = np.zeros(N ** 2)
87     b = g.reshape(N, N)
88     b[-1,:] = b[1,:]
89     b[:, -1] = b[:, -1]
90     b = b.reshape(N ** 2)
91     # source
92     b += - g * np.square(dx)
93     return b
94
95 # %%
96 # solve
97 def solve(N, g, dx, x, y):

```

```

98     g = generate_g(g, N, x, y).reshape(N ** 2)
99     a = generate_A(N)
100    a = csc_matrix(a) # transform to the fitting matrix
101    b = generate_b(N, g, dx)
102    x = splinalg.spsolve(a, b).reshape(N, N)
103    return x
104
105    # %%
106    u = solve(N, g, dx, x, y)
107    # %timeit solve(N, g, dx, x, y)
108
109    # %% [markdown]
110    # parallel: 6.31 s ± 29.3 ms per loop (mean ± std. dev. of 7 runs, 1
111    # loop each)
112    # nopython: 6.08 s ± 136 ms per loop (mean ± std. dev. of 7 runs, 1 loop
113    # each)
114
115    # %%
116    figure(figsize=(10, 10), dpi=100)
117    plt.imshow(u, origin = 'lower', extent=[-5, 5, -5, 5])
118    plt.colorbar()
119    plt.contour(u, colors = 'w', alpha = .3, extent=[-5, 5, -5, 5])
120    plt.title('solution')
121    plt.xlabel('x-position [code unit]')
122    plt.ylabel('y-position [code unit]')
123    plt.show()
124    plt.close()

```

---

### 3.5 3\_interactive\_methods.ipynb

---

```

1    # %% [markdown]
2    # # Laplace Equation & Poisson Equations
3    #
4    # In this Lab, we will learn how to numerically solve Laplace and
5    # Poisson equations, which are common equations in electromagnetism
6    # and gravitational problems.
7
8    # %% [markdown]
9    # There should be two files under `./poisson_solver`.
10   # 1. `mesh.py` handles the mesh grids we will used in this Lab.
11   # 2. `solvers.py` handles all corresponding iterative solvers for
12   # Laplace/Poisson Equation.
13
14   # %%
15   %reset -f
16
17   import numpy as np
18   import numba as na
19   import matplotlib.pyplot as plt
20   from matplotlib.pyplot import figure
21   import time as time
22   from poisson_solver.mesh import Mesh2D

```

```

20 from poisson_solver.solvers import *
21
22 # %% [markdown]
23 # ## Exercise 4: Jacobi method
24 #
25 # 1. Test your Mesh2D class to see if you could generate the grids we
    need for this calculation
26 # 2. Implement the Jacobi meothd in `./poisson_solver/solver.py`.
27 # 3. Write a function called `update_boundary()` to update the boundary
    conditions.\
28 # Where to put this `update_boundary()` function is up to you.\
29 # You could put it either inside the `Mesh2D` class, in `solvers.py`,
    or here.
30
31 # %% [markdown]
32 # ## Exercise 5: Gauss-Seidel Meothd.
33 #
34 # 1. Implement the Gauss-Seidel meothd in your solver.
35 # 2. Repeat exercise 4. for the Gauss-Seidel meothd.
36 # 3. Compare the error convergence between Jacobi and Gauss-Seidel
37
38 # %%
39 def setup():
40     tor = 1e-7
41     xmin, xmax = -5, 5
42     ymin, ymax = -5, 5
43     nx, ny = 128, 128
44     buff_size = 1
45
46     mesh = Mesh2D(nx = nx, ny = ny, buff_size = buff_size)
47     mesh.set_xmin(xmin)
48     mesh.set_xmax(xmax)
49     mesh.set_ymin(ymin)
50     mesh.set_ymax(ymax)
51
52     return tor, mesh
53
54 # %%
55 '''
56 Jacobi
57 '''
58 ini = setup()
59 t1 = time.time()
60 # name == Gauss or Jacobi or SOR
61 j_arr = solve("Jacobi", ini[0], ini[1])
62 j_u = j_arr[0]
63 j_err = j_arr[1]
64 j_n = j_arr[2]
65 j_N = np.linspace(0, int(j_n), int(j_n))
66 t2 = time.time()
67
68 '''
69 Gauss-Seidel

```



```

70     '''
71     ini = setup()
72     t3 = time.time()
73     # name == Gauss or Jacobi or SOR
74     g_arr = solve("Gauss", ini[0], ini[1])
75     g_u = g_arr[0]
76     g_err = g_arr[1]
77     g_n = g_arr[2]
78     g_N = np.linspace(0, int(g_n), int(g_n))
79     t4 = time.time()
80
81     '''
82     SOR
83     '''
84     ini = setup()
85     t5 = time.time()
86     # name == Gauss or Jacobi or SOR
87     w1, w2, w3 = 1.2, 1.5, 2.0
88
89     s1_arr = solve("SOR", ini[0], ini[1], w = w1)
90     s1_u = s1_arr[0]
91     s1_err = s1_arr[1]
92     s1_n = s1_arr[2]
93     s1_N = np.linspace(0, int(s1_n), int(s1_n))
94     # prune the margins
95     s1_u = np.delete(s1_u, (0, -1), 0)
96     s1_u = np.delete(s1_u, (0, -1), 1)
97
98     ini = setup()
99     s2_arr = solve("SOR", ini[0], ini[1], w = w2)
100    s2_u = s2_arr[0]
101    s2_err = s2_arr[1]
102    s2_n = s2_arr[2]
103    s2_N = np.linspace(0, int(s2_n), int(s2_n))
104    # prune the margins
105    s2_u = np.delete(s2_u, (0, -1), 0)
106    s2_u = np.delete(s2_u, (0, -1), 1)
107
108    ini = setup()
109    s3_arr = solve("SOR", ini[0], ini[1], w = w3)
110    s3_u = s3_arr[0]
111    s3_err = s3_arr[1]
112    s3_n = s3_arr[2]
113    s3_N = np.linspace(0, int(s3_n), int(s3_n))
114    # prune the margins
115    s3_u = np.delete(s3_u, (0, -1), 0)
116    s3_u = np.delete(s3_u, (0, -1), 1)
117    t6 = time.time()
118
119    print("Jacobi -> Time = ", np.round((t2-t1), 2))
120    print("Gauss -> Time = ", np.round((t4-t3), 2))
121    print("SOR -> Time = ", np.round((t6-t5), 2))
122    print("Done!")

```

```

123
124 # %% [markdown]
125 # ### Visualize your results
126
127 # %%
128 '''
129 Jacobi
130 '''
131 # prune the margins
132 j_u = np.delete(j_u, (0, -1), 0)
133 j_u = np.delete(j_u, (0, -1), 1)
134
135 figure(figsize=(8, 8), dpi=100)
136 plt.imshow(j_u, origin = 'lower', extent=[-5, 5, -5, 5])
137 plt.colorbar()
138 plt.contour(j_u, colors = 'w', alpha = .3, extent=[-5, 5, -5, 5])
139 plt.title("Jacobi")
140 plt.show()
141
142 '''
143 Gauss-Seidel
144 '''
145 # prune the margins
146 g_u = np.delete(g_u, (0, -1), 0)
147 g_u = np.delete(g_u, (0, -1), 1)
148
149 figure(figsize=(8, 8), dpi=100)
150 plt.imshow(g_u, origin = 'lower', extent=[-5, 5, -5, 5])
151 plt.colorbar()
152 plt.contour(g_u, colors = 'w', alpha = .3, extent=[-5, 5, -5, 5])
153 plt.title("Gauss-Seidel")
154 plt.show()
155
156 '''
157 SOR
158 '''
159 figure(figsize=(8, 8), dpi=100)
160 plt.imshow(s1_u, origin = 'lower', extent=[-5, 5, -5, 5])
161 plt.colorbar()
162 plt.contour(s1_u, colors = 'w', alpha = .3, extent=[-5, 5, -5, 5])
163 plt.title(f"SOR, with w = {w1}")
164 plt.show()
165
166 # %% [markdown]
167 # ### Error convergence.
168 #
169 # To see how it converge, we could make a of Error vs. Iteration times
    to see how it converges.
170
171 # %%
172 figure(figsize=(9, 7), dpi=100)
173 plt.yscale('log')
174 plt.plot(j_N, j_err, label = "Jacobi")

```

```

175 plt.plot(g_N, g_err, label = "Gauss")
176 plt.plot(s1_N, s1_err, label = f"SOR, w = {w1}")
177 plt.plot(s2_N, s2_err, label = f"SOR, w = {w2}")
178 plt.plot(s3_N, s3_err, label = f"SOR, w = {w3}")
179 plt.xlabel("steps")
180 plt.ylabel("errors")
181 plt.title("Error Convergence")
182 plt.legend(loc = 'best')

```

---

### 3.6 4\_interactive\_methods.ipynb

---

```

1 # %%
2 %reset -f
3
4 from scipy import linalg
5 from scipy.sparse import csc_matrix
6 from scipy.sparse import dia_array
7 from scipy.sparse import dia_matrix
8 import scipy.sparse.linalg as splinalg
9 from numba import njit, prange
10
11 import numpy as np
12 import matplotlib.pyplot as plt
13 import matplotlib
14 from matplotlib.pyplot import figure
15 import time as time
16 from poisson_solver.mesh import Mesh2D
17 from poisson_solver.solvers import *
18
19 # %%
20 def generate_D(n):
21     ex = np.ones(n)
22     data = np.array([-1 * ex, 4 * ex, -1 * ex])
23     offsets = np.array([-1, 0, 1])
24
25     return dia_matrix((data, offsets), shape=(n, n)).toarray()
26
27 # %%
28 @njit
29 def kernel(N, A, D, I):
30     for i in prange(N):
31         for j in prange(N):
32             if i == j:
33                 for ii in prange(N):
34                     for jj in prange(N):
35                         A[ii+N*i, jj+N*j] = D[ii, jj]
36
37             if np.abs(i - j) == 1:
38                 for ii in prange(N):
39                     for jj in prange(N):
40                         A[ii+N*i, jj+N*j] = I[ii, jj]
41

```

```

42         return A
43
44     # %%
45     def generate_A(N):
46         A = np.zeros((N ** 2, N ** 2))
47         D = generate_D(N)
48         I = -np.identity(N)
49         A = kernel(N, A, D, I)
50         return A
51
52     # %%
53     @njit
54     def generate_g(g, N, x, y):
55         for i in prange(N):
56             for j in prange(N):
57                 r1 = np.square(x[i] + 1.5) + np.square(y[j])
58                 r2 = np.square(x[i] - 1.5) + np.square(y[j])
59                 g[i, j] = np.exp(-5 * 0.25 * np.square(r1)) + 3 * 0.5 *
                    np.exp(-np.square(r2))
60
61         return g
62
63     # %%
64     def generate_b(N, g, dx):
65         # boundaries
66         # b = np.zeros(N ** 2)
67         b = g.reshape(N, N)
68         b[-1,:] = b[1,:]
69         b[:, -1] = b[:, -1]
70         b = b.reshape(N ** 2)
71         # source
72         b += - g * np.square(dx)
73         return b
74
75     # %%
76     # solve
77     def psolve(N, g, dx, x, y):
78         g = generate_g(g, N, x, y).reshape(N ** 2)
79         a = generate_A(N)
80         a = csc_matrix(a) # transform to the fitting matrix
81         b = generate_b(N, g, dx)
82         x = splinalg.spsolve(a, b).reshape(N, N)
83         return x
84
85     # %%
86     def setup(N):
87         min, max = -5, 5
88         dx = (max - min) / N
89
90         x = np.linspace(-5, 5, N)
91         y = np.linspace(-5, 5, N)
92         g = np.zeros([N, N])
93

```

```

94         return N, g, dx, x, y
95
96     # %%
97     t0_32 = time.time()
98     N, g, dx, x, y = setup(32)
99     u = psolve(N, g, dx, x, y)
100     t00_32 = time.time()
101     t32 = np.round(t00_32 - t0_32, 5)
102
103     t0_64 = time.time()
104     N, g, dx, x, y = setup(64)
105     u = psolve(N, g, dx, x, y)
106     t00_64 = time.time()
107     t64 = np.round(t00_64 - t0_64, 5)
108
109     t0_128 = time.time()
110     N, g, dx, x, y = setup(128)
111     u = psolve(N, g, dx, x, y)
112     t00_128 = time.time()
113     t128 = np.round(t00_128 - t0_128, 5)
114
115     m = [[t32, t64, t128], [32, 64, 128]]
116
117     # %%
118     def setup32():
119         tor = 1e-7
120         xmin, xmax = -5, 5
121         ymin, ymax = -5, 5
122         nx, ny = 32, 32
123         buff_size = 1
124
125         mesh = Mesh2D(nx = nx, ny = ny, buff_size = buff_size)
126         mesh.set_xmin(xmin)
127         mesh.set_xmax(xmax)
128         mesh.set_ymin(ymin)
129         mesh.set_ymax(ymax)
130
131         return tor, mesh
132
133     # %%
134     def setup64():
135         tor = 1e-7
136         xmin, xmax = -5, 5
137         ymin, ymax = -5, 5
138         nx, ny = 64, 64
139         buff_size = 1
140
141         mesh = Mesh2D(nx = nx, ny = ny, buff_size = buff_size)
142         mesh.set_xmin(xmin)
143         mesh.set_xmax(xmax)
144         mesh.set_ymin(ymin)
145         mesh.set_ymax(ymax)
146

```

```

147         return tor, mesh
148
149     # %%
150     def setup128():
151         tor = 1e-7
152         xmin, xmax = -5, 5
153         ymin, ymax = -5, 5
154         nx, ny = 128, 128
155         buff_size = 1
156
157         mesh = Mesh2D(nx = nx, ny = ny, buff_size = buff_size)
158         mesh.set_xmin(xmin)
159         mesh.set_xmax(xmax)
160         mesh.set_ymin(ymin)
161         mesh.set_ymax(ymax)
162
163         return tor, mesh
164
165     # %%
166     '''
167     Jacobi
168     '''
169     def j(name):
170         if name == 32:
171             ini = setup32()
172         elif name == 64:
173             ini = setup64()
174         elif name == 128:
175             ini = setup128()
176         else:
177             print("No such Grid!")
178             quit()
179
180         t1 = time.time()
181         # name == Gauss or Jacobi or SOR
182         j_arr = solve("Jacobi", ini[0], ini[1])
183         t2 = time.time()
184
185         # print(f"Jacobi{name} -> Time = ", np.round((t2-t1), 2))
186         return name, np.round((t2-t1), 5)
187
188     # %%
189     '''
190     Gauss-Seidel
191     '''
192     def g(name):
193         if name == 32:
194             ini = setup32()
195         elif name == 64:
196             ini = setup64()
197         elif name == 128:
198             ini = setup128()
199         else:

```

```

200     print("No such Grid!")
201     quit()
202
203     t3 = time.time()
204     # name == Gauss or Jacobi or SOR
205     g_arr = solve("Gauss", ini[0], ini[1])
206     t4 = time.time()
207
208     # print(f"Gauss{name} -> Time = ", np.round((t4-t3), 2))
209     return name, np.round((t4-t3), 5)
210
211 # %%
212 '''
213 SOR
214 '''
215 def s(name):
216     if name == 32:
217         ini = setup32()
218     elif name == 64:
219         ini = setup64()
220     elif name == 128:
221         ini = setup128()
222     else:
223         print("(s1) No such Grid!")
224         quit()
225
226     t5 = time.time()
227     # name == Gauss or Jacobi or SOR
228     w1, w2 = 1.2, 1.5
229
230     s1_arr = solve("SOR", ini[0], ini[1], w = w1)
231     t6 = time.time()
232
233     if name == 32:
234         ini = setup32()
235     elif name == 64:
236         ini = setup64()
237     elif name == 128:
238         ini = setup128()
239     else:
240         print("(s2) No such Grid!")
241         quit()
242     t7 = time.time()
243     s2_arr = solve("SOR", ini[0], ini[1], w = w2)
244     t8 = time.time()
245
246     # print(f"SOR{name} -> Time = ", np.round((t6-t5), 2))
247     # print(f"SOR{name} -> Time = ", np.round((t8-t7), 2))
248     return name, w1, w2, np.round((t6-t5), 5), np.round((t8-t7), 5)
249
250 # %% [markdown]
251 # ### Error convergence.
252 #

```

```

253     # To see how it converge, we could make a of Error vs. Iteration times
      to see how it converges.
254
255     # %%
256     import matplotlib
257     figure(figsize=(11, 7), dpi=150)
258     matplotlib.rcParams['legend.handlelength'] = 0
259     matplotlib.rcParams['legend.numpoints'] = 1
260     plt.yscale('log')
261     plt.xscale('log')
262
263     j = [[j(32)[1], j(64)[1], j(128)[1]], [[j(32)[0], j(64)[0], j(128)[0]]]]
264     g = [[g(32)[1], g(64)[1], g(128)[1]], [[g(32)[0], g(64)[0], g(128)[0]]]]
265     s1 = [[s(32)[3], s(64)[3], s(128)[3]], [s(32)[0], s(64)[0], s(128)[0]]]
266     s2 = [[s(32)[4], s(64)[4], s(128)[4]], [s(32)[0], s(64)[0], s(128)[0]]]
267
268     plt.plot(j[0], j[1][0], '--o', alpha = .3, color = 'r', label = "Jacobi")
269     plt.plot(g[0], g[1][0], '--o', alpha = .3, color = 'g', label =
      "Gauss-Seidel")
270     plt.plot(m[0], m[1], '--o', alpha = .3, color = 'b', label = "Sparse
      Matrix")
271     plt.plot(s1[0], s1[1], '--o', alpha = .5, color = 'y', label = f"SOR, w
      = {s(32)[1]}")
272     plt.plot(s2[0], s2[1], '--o', alpha = .9, color = 'y', label = f"SOR, w
      = {s(32)[2]}")
273
274     plt.xlabel("time (log) [s]")
275     plt.ylabel("Resolution (log)")
276     plt.title("Resolution vs time")
277     plt.legend(loc = 'best')

```

---