ES_APPM 445 Relaxation Project

Northwestern University

Nicolas Guerra
January 29, 2023

1 Discretization

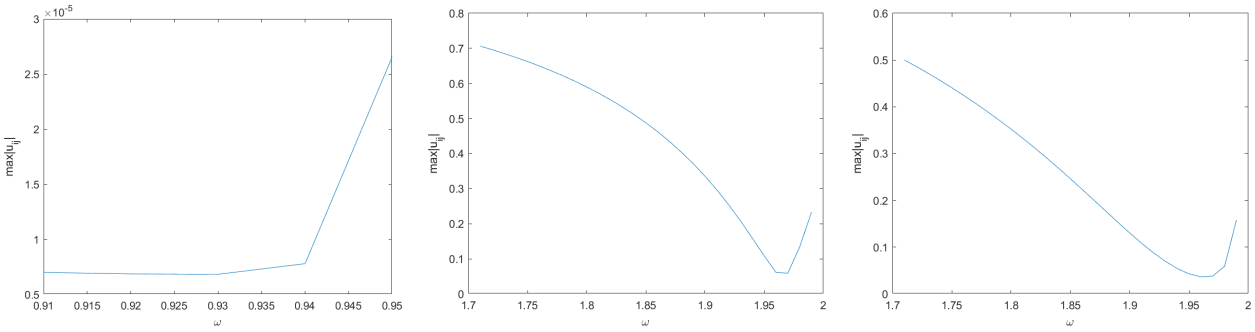
In order to solve numerically $-(\epsilon u_{xx} + u_{yy}) = 0$, $0 \leq x \leq 1$, $0 \leq y \leq 1$ with $u = 0$ on all boundaries, we must first discretize the problem. We'll take the second-order centered difference approximation of the second derivatives: $u_{xx} \approx \frac{u_{i+1,j}+u_{i-1,j}-2u_{ij}}{h^2}$ and $u_{yy} \approx \frac{u_{i,j+1}+u_{i,j-1}-2u_{ij}}{h^2}$. In our discretization, the total number of points including the boundaries will be $(N+2)^2$ meaning the interior grid will have N^2 points. In our approximation of the second derivatives, $h = 1/(N+1)$. Plugging both second derivative approximations into our original problem and simplifying gives us the following equation:
$\epsilon u_{i+1,j} + \epsilon u_{i-1,j} - (2\epsilon + 2) * u_{ij} + u_{i,j+1} + u_{i,j-1} = 0.$

2 Methods

Below are the equations to solve the PDE with point Jacobi, weighted Jacobi, Gauss-Seidel, red-black Gauss-Seidel, SOR, SSOR, and Kaczmarz relaxation. With the equation from Section 1, we can represent the problem in the form $A\vec{x} = \vec{0}$ where x is u reshaped to a vector. However, since A is sparse, it is more efficient to work with the following equations.
Point Jacobi: $u_{i,j\ new} = \frac{-\epsilon u_{i+1,j} - \epsilon u_{i-1,j} - u_{i,j+1} - u_{i,j-1}}{-(2\epsilon+2)}$
Weighted Jacobi: $u_{i,j\ new} = \frac{-(2\epsilon+2)u_{i,j} - \omega*(\epsilon u_{i+1,j} + \epsilon u_{i-1,j} - (2\epsilon+2)*u_{ij} + u_{i,j+1} + u_{i,j-1})}{-(2\epsilon+2)}$
Gauss-Seidel: $u_{i,j} = \frac{\epsilon u_{i+1,j} + \epsilon u_{i-1,j} + u_{i,j+1} + u_{i,j-1}}{2\epsilon+2}$
Red-Black Gauss-Seidel; Perform Gauss-Seidel in parallel updating a red and black point simultaneously.
SOR: $u_{i,j} = \omega \frac{\epsilon u_{i+1,j} + \epsilon u_{i-1,j} - (2\epsilon+2)*u_{ij} + u_{i,j+1} + u_{i,j-1}}{2\epsilon+2} + u_{i,j}$
SSOR: Use SOR from bottom to top and left to right, and then again but from top to bottom and right to left.
Kaczmarz: $\vec{x} = \vec{x} + \delta_i * \vec{a}_i$ where $\delta = \frac{(\vec{a}_i, \vec{x})}{(\vec{a}_i, \vec{a}_i)}$ and $\vec{a}_i$ is the $i-th$ row of A

3 Sensitivity of ω

Depending on N, ϵ, and the initial data itself, there is an optimal ω that must be chosen to achieve faster convergence.



(a) Weighted Jacobi, Data (a) (b) SOR, Data (c) (c) SSOR, Data (c)

Figure 1: $max|u_{ij}|$ after 100 iterations as a function of ω with $N = 128$ and $\epsilon = 1$

One can see in Figure 1 how the convergence rate strongly depends on ω in Weighted Jacobi, SOR, and SSOR. The minimum in each of these subplots is the optimal ω that should be used. In each case of N and ϵ, the optimal ω will be found by seeing which ω gives the smallest $max|u_{jk}|$ in 100 iterations. We will experiment around $\omega = 2/3$ for Weighted Jacobi and $\omega = 1.8$ for SOR and SSOR.

4 Convergence Results and Conclusion

After letting $N = 16, 32, 64, 128$ and $\epsilon = 1, .1, .01, .001$ and after adjusting to the optimal ω value, below are tables giving the number of iterations each method took to reach $max|u_{jk}| < tol$ where $tol = 10^{-7}$.

Table 1: Number of iterations to reach desired tolerance

Numerical Aspects			**Data (a)**		**Data (b)**		**Data (c)**	
Method	**N**	ϵ	ω	#	ω	#	ω	#
Point Jacobi	16	1	N/A	966	N/A	505	N/A	937
Weighted Jacobi	16	1	0.93	131	1	505	1	937
Gauss-Seidel	16	1	N/A	174	N/A	259	N/A	468
Red-Black Gauss-Seidel	16	1	N/A	484	N/A	253	N/A	469
SOR	16	1	1.68	57	1.69	54	1.69	52
SSOR	16	1	1.71	49	1.7	48	1.73	64
Point Jacobi	16	0.1	N/A	966	N/A	505	N/A	937
Weighted Jacobi	16	0.1	0.93	132	1	505	1	937
Gauss-Seidel	16	0.1	N/A	201	N/A	259	N/A	468
Red-Black Gauss-Seidel	16	0.1	N/A	484	N/A	253	N/A	469
SOR	16	0.1	1.69	58	1.69	54	1.69	52
SSOR	16	0.1	1.63	45	1.69	50	1.71	78
Point Jacobi	16	0.01	N/A	961	N/A	566	N/A	937
Weighted Jacobi	16	0.01	0.94	159	1	566	1	937
Gauss-Seidel	16	0.01	N/A	258	N/A	290	N/A	468
Red-Black Gauss-Seidel	16	0.01	N/A	481	N/A	284	N/A	469
SOR	16	0.01	1.69	51	1.69	49	1.7	50
SSOR	16	0.01	1.67	59	1.68	57	1.69	84
Point Jacobi	16	0.001	N/A	954	N/A	689	N/A	938
Weighted Jacobi	16	0.001	0.94	184	1	689	1	938
Gauss-Seidel	16	0.001	N/A	334	N/A	351	N/A	468
Red-Black Gauss-Seidel	16	0.001	N/A	478	N/A	345	N/A	470
SOR	16	0.001	1.69	50	1.69	49	1.7	49
SSOR	16	0.001	1.67	73	1.68	69	1.69	84
Point Jacobi	32	1	N/A	3658	N/A	1690	N/A	3550
Weighted Jacobi	32	1	0.93	343	1	1690	1	3550
Gauss-Seidel	32	1	N/A	300	N/A	910	N/A	1774
Red-Black Gauss-Seidel	32	1	N/A	1830	N/A	846	N/A	1776
SOR	32	1	1.79	87	1.81	101	1.83	98
SSOR	32	1	1.81	81	1.71	62	1.85	127
Point Jacobi	32	0.1	N/A	3658	N/A	1687	N/A	3550
Weighted Jacobi	32	0.1	0.93	354	1	1687	1	3550
Gauss-Seidel	32	0.1	N/A	506	N/A	908	N/A	1774
Red-Black Gauss-Seidel	32	0.1	N/A	1830	N/A	844	N/A	1776
SOR	32	0.1	1.79	87	1.81	102	1.83	97
SSOR	32	0.1	1.77	79	1.81	103	1.84	153
Point Jacobi	32	0.01	N/A	3638	N/A	1945	N/A	3550
Weighted Jacobi	32	0.01	0.95	438	1	1945	1	3550
Gauss-Seidel	32	0.01	N/A	701	N/A	1030	N/A	1775
Red-Black Gauss-Seidel	32	0.01	N/A	1820	N/A	973	N/A	1776
SOR	32	0.01	1.81	98	1.82	96	1.83	94
SSOR	32	0.01	1.8	103	1.81	120	1.83	165
Point Jacobi	32	0.001	N/A	3608	N/A	2487	N/A	3551

Weighted Jacobi	32	0.001	0.95	588	0.99	2512	1	3551
Gauss-Seidel	32	0.001	N/A	961	N/A	1304	N/A	1775
Red-Black Gauss-Seidel	32	0.001	N/A	1805	N/A	1244	N/A	1776
SOR	32	0.001	1.82	98	1.83	96	1.83	94
SSOR	32	0.001	1.81	126	1.81	148	1.83	167
Point Jacobi	64	1	N/A	14208	N/A	5468	N/A	13792
Weighted Jacobi	64	1	0.93	713	1	5468	1	13792
Gauss-Seidel	64	1	N/A	538	N/A	3026	N/A	6896
Red-Black Gauss-Seidel	64	1	N/A	7105	N/A	2735	N/A	6897
SOR	64	1	1.73	195	1.83	346	1.92	199
SSOR	64	1	1.85	144	1.88	192	1.92	251
Point Jacobi	64	0.1	N/A	14208	N/A	5438	N/A	13792
Weighted Jacobi	64	0.1	0.93	823	1	5438	1	13792
Gauss-Seidel	64	0.1	N/A	1040	N/A	3017	N/A	6896
Red-Black Gauss-Seidel	64	0.1	N/A	7105	N/A	2720	N/A	6897
SOR	64	0.1	1.83	178	1.87	259	1.92	210
SSOR	64	0.1	1.85	139	1.81	168	1.92	304
Point Jacobi	64	0.01	N/A	14131	N/A	6708	N/A	13793
Weighted Jacobi	64	0.01	0.94	1179	1	6708	1	13793
Gauss-Seidel	64	0.01	N/A	1728	N/A	3592	N/A	6896
Red-Black Gauss-Seidel	64	0.01	N/A	7066	N/A	3355	N/A	6897
SOR	64	0.01	1.85	206	1.89	222	1.91	190
SSOR	64	0.01	1.86	186	1.88	221	1.91	328
Point Jacobi	64	0.001	N/A	14015	N/A	9114	N/A	13793
Weighted Jacobi	64	0.001	0.95	1706	0.99	9206	1	13793
Gauss-Seidel	64	0.001	N/A	2560	N/A	4829	N/A	6896
Red-Black Gauss-Seidel	64	0.001	N/A	7008	N/A	4558	N/A	6897
SOR	64	0.001	1.87	225	1.91	190	1.91	189
SSOR	64	0.001	1.88	223	1.88	291	1.91	331
Point Jacobi	128	1	N/A	55976	N/A	16985	N/A	54346
Weighted Jacobi	128	1	0.93	823	1	16985	1	54346
Gauss-Seidel	128	1	N/A	688	N/A	9686	N/A	27172
Red-Black Gauss-Seidel	128	1	N/A	27989	N/A	8493	N/A	27174
SOR	128	1	1.73	247	1.81	1327	1.97	560
SSOR	128	1	1.64	498	1.54	1640	1.96	500
Point Jacobi	128	0.1	N/A	55976	N/A	16650	N/A	54346
Weighted Jacobi	128	0.1	0.93	1371	1	16650	1	54346
Gauss-Seidel	128	0.1	N/A	1260	N/A	9589	N/A	27172
Red-Black Gauss-Seidel	128	0.1	N/A	27989	N/A	8326	N/A	27174
SOR	128	0.1	1.75	458	1.86	989	1.96	437
SSOR	128	0.1	1.5	609	1.86	557	1.96	608
Point Jacobi	128	0.01	N/A	55672	N/A	23537	N/A	54347
Weighted Jacobi	128	0.01	0.94	2656	1	23537	1	54347
Gauss-Seidel	128	0.01	N/A	3325	N/A	12637	N/A	27173
Red-Black Gauss-Seidel	128	0.01	N/A	27837	N/A	11769	N/A	27174
SOR	128	0.01	1.78	734	1.91	728	1.96	415
SSOR	128	0.01	1.54	953	1.91	432	1.95	654
Point Jacobi	128	0.001	N/A	55217	N/A	33673	N/A	54347
Weighted Jacobi	128	0.001	0.95	4271	1	33673	1	54347
Gauss-Seidel	128	0.001	N/A	6159	N/A	17942	N/A	27173
Red-Black Gauss-Seidel	128	0.001	N/A	27609	N/A	16837	N/A	27174

SOR	128	0.001	1.83	872	1.64	4287	1.96	417
SSOR	128	0.001	1.89	481	1.52	2754	1.95	660

Before discussing the results, it's important to point out that initial data (a) represents high frequency modes while data (c) represents smooth modes. Data (b) goes from representing smoother modes to representing high frequency modes as N increases from 16 to 128. Looking at Table 1, we can see that Point Jacobi seems to consistently be the slowest out of all the methods. It seems that the best-performing method is SOR especially when the problem becomes more singular (i.e., $\epsilon \to 0$). Seeing that the optimal ω for Weighted Jacobi is less than 1 for data (a) confirms that data (a) is representing high frequency modes that are not well resolved by the grid. Looking at Gauss-Seidel and Red-Black Gauss-Seidel, a common trend is that Gauss-Seidel performs better with data (a), Red-Black Gauss-Seidel performs better with data (b), and they perform approximately the same with data(c). This could suggest that Red-Black Gauss-Seidel performs poorly with data with high-frequency modes. Note that in order to determine that SSOR is faster than SOR, SSOR has to converge in less than half the amount of iterations SOR converges in since one iteration of SSOR includes two sweeps of SOR. This does not seem to happen which indicates that for the three starting data, SOR performs better than SSOR.

By setting $\omega = 1$ for SSOR, we get a symmetrized version of the Gauss-Seidel method.

Table 2: SSOR results with $\omega = 1$

N	ϵ	**# for Data (a)**	**# for Data (b)**	**# for Data (c)**
16	1	116	131	240
16	0.1	107	133	242
16	0.01	132	149	244
16	0.001	172	181	244
32	1	284	422	894
32	0.1	266	438	896
32	0.01	352	507	897
32	0.001	483	647	898
64	1	515	1272	3454
64	0.1	552	1357	3457
64	0.01	865	1702	3458
64	0.001	1279	2311	3459
128	1	332	3713	13593
128	0.1	692	4037	13595
128	0.01	1669	5904	13597
128	0.001	3075	8454	13597

After comparing Table 2 with the Gauss-Seidel results in Table 1, we can see that the performance is roughly the same. Recall that SSOR includes two sweeps in one iteration, so two times the iterations it takes to reach convergence is roughly the same amount for Gauss-Seidel as expected.

Since the Kaczmarz method takes long to achieve the desired tolerance, we will instead run the method 10 times and give the max residual.

Table 3: Maximum residual after 10 iterations of Kaczmarz method

N	ϵ	$max\|u_{j,k}\|$ **for Data (a)**	$max\|u_{j,k}\|$ **for Data (b)**	$max\|u_{j,k}\|$ **for Data (c)**
16	1	0.005140445	0.031692826	0.9870994
16	0.1	0.029028549	0.086123316	0.991363488
16	0.01	0.045946569	0.11382129	0.992411159

16	0.001	0.047594879	0.116609536	0.992522988
32	1	0.005140396	0.391016426	0.997410816
32	0.1	0.02901825	0.459473032	0.997455182
32	0.01	0.045937377	0.485191056	0.997461956
32	0.001	0.047598341	0.487984988	0.997462818
64	1	0.005140396	0.925074786	0.999394348
64	0.1	0.02901825	0.930051062	0.999396914
64	0.01	0.045937378	0.933154782	0.999397855
64	0.001	0.047598342	0.933617055	0.999397963
128	1	0.005140396	0.994529888	0.999850329
128	0.1	0.02901825	0.999958195	0.999850495
128	0.01	0.045937378	1.001136729	0.999850556
128	0.001	0.047598342	1.001257683	0.999850562

Looking at Table 3, we can that after 10 iterations the max residuals with initial data (a) are a lot smaller than the max residuals with initial data (c). For the max residuals that had initial data (b), they are small when N is small and ϵ is large, but then are quite big when N increases and ϵ decreases. It is as if the 10 iterations barely made a dent, for lack of better words. The reason for this is likely due to the characteristics of the initial data itself such as the frequency mode. Lastly, while Kaczmarz works for any nonsingular matrix, as shown, convergence is slow.

Another note that should be made is that the role of ϵ is to make the problem more singular as $\epsilon \to 0$. When the problem is singular with $\epsilon = 0$, this means that the problem has a nontrivial solution. Now, since we are only looking at $\epsilon > 0$, what does this mean for us? This means that the problem has a tendency to want to become nontrivial, however, in the end, it converges to zero. Thus, when ϵ gets closer to zero, we expect the number of iterations to converge to increase which is consistent with Table 1.

As a bottom line, one should use SOR as $\epsilon \to 0$. One can see that in many instances in Table 1, the number of iterations for SOR actually decreases as ϵ. I would also say to use SOR as N increases as well. The number of iterations it takes to reach the desired tolerance is quite impressive. While Point Jacobi takes 55, 217 iterations with $N = 128$ and $\epsilon = 0.001$ with Data (a), SOR achieves the same tolerance in 872 iterations.

5 MATLAB Script: Storing Number of Iterations

```
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This program goes through every combination of N and epsilon for each
% starting data and solves the PDE of interest using six different methods.
% It then stores the number of iterations it takes to reach the
% desired tolerance

N = [16, 32, 64, 128]; % Number of interior points
eps = [1, .1, .01, .001];
tol = 10^-7;

%% Starting Data (a)
sz = [112 5];
var_types = ["string","double","double","double","double"];
var_names = ["Method","N","Epsilon","Omega","Number of Iterations"];
```

```
output_a = table('Size',sz,'VariableTypes',var_types,'VariableNames',var_names
    );
fill_row = 1;
for i = 1:length(N)
    % (a) initial u
    u_init = starting_data_a(N(i));

    for j = 1:length(eps)
        % Point Jacobi Loop
        [~, num_iter] = point_jacobi_loop(u_init, tol, N(i), eps(j));
        % Save number of iterations
        output_a(fill_row,:)={'Point Jacobi',N(i),eps(j),NaN,num_iter};
        fill_row = fill_row + 1;


        % Weighted Jacobi Loop
        w = get_optimal_w('Weighted Jacobi', u_init, N(i), eps(j));
        [~, num_iter] = weighted_jacobi_loop(u_init, tol, N(i), eps(j), w);
        % Save number of iterations
        output_a(fill_row,:)={'Weighted Jacobi',N(i),eps(j),w,num_iter};
        fill_row = fill_row + 1;

        % Gauss-Seidel Loop
        [~, num_iter] = gauss_seidel_loop(u_init, tol, N(i), eps(j));
        % Save number of iterations
        output_a(fill_row,:)={'Gauss-Seidel',N(i),eps(j),NaN,num_iter};
        fill_row = fill_row + 1;

        % Red-Black Gauss-Seidel Loop
        [~, num_iter] = rb_gauss_seidel_loop(u_init, tol, N(i), eps(j));
        % Save number of iterations
        output_a(fill_row,:)={'Red-Black Gauss-Seidel',N(i),eps(j),NaN,
            num_iter};
        fill_row = fill_row + 1;

        % SOR Loop
        w = get_optimal_w('SOR', u_init, N(i), eps(j));
        [~, num_iter] = SOR_loop(u_init, tol, N(i), eps(j), w);
        % Save number of iterations
        output_a(fill_row,:)={'SOR',N(i),eps(j),w,num_iter};
        fill_row = fill_row + 1;

        % SSOR Loop
        w = get_optimal_w('SSOR', u_init, N(i), eps(j));
        [~, num_iter] = SSOR_loop(u_init, tol, N(i), eps(j), w);
        % Save number of iterations
        output_a(fill_row,:)={'SSOR',N(i),eps(j),w,num_iter};
        fill_row = fill_row + 1;

        % Kaczmarz Loop
        [~, num_iter] = kaczmarz_loop(u_init, tol, N(i), eps(j));
```

```
66          % Save number of iterations
67          output_a(fill_row,:)={'Kaczmarz',N(i),eps(j),NaN,num_iter};
68          fill_row = fill_row + 1;
69      end
70  end
71
72  %% Starting Data (b)
73  output_b = table('Size',sz,'VariableTypes',var_types,'VariableNames',var_names
        );
74  fill_row = 1;
75  for i = 1:length(N)
76      % (b) initial u
77      u_init = starting_data_b(N(i));
78
79      for j = 1:length(eps)
80          % Point Jacobi Loop
81          [~, num_iter] = point_jacobi_loop(u_init, tol, N(i), eps(j));
82          % Save number of iterations
83          output_b(fill_row,:)={'Point Jacobi',N(i),eps(j),NaN,num_iter};
84          fill_row = fill_row + 1;
85
86
87          % Weighted Jacobi Loop
88          w = get_optimal_w('Weighted Jacobi', u_init, N(i), eps(j));
89          [~, num_iter] = weighted_jacobi_loop(u_init, tol, N(i), eps(j), w);
90          % Save number of iterations
91          output_b(fill_row,:)={'Weighted Jacobi',N(i),eps(j),w,num_iter};
92          fill_row = fill_row + 1;
93
94          % Gauss-Seidel Loop
95          [~, num_iter] = gauss_seidel_loop(u_init, tol, N(i), eps(j));
96          % Save number of iterations
97          output_b(fill_row,:)={'Gauss-Seidel',N(i),eps(j),NaN,num_iter};
98          fill_row = fill_row + 1;
99
100         % Red-Black Gauss-Seidel Loop
101         [~, num_iter] = rb_gauss_seidel_loop(u_init, tol, N(i), eps(j));
102         % Save number of iterations
103         output_b(fill_row,:)={'Red-Black Gauss-Seidel',N(i),eps(j),NaN,
                num_iter};
104         fill_row = fill_row + 1;
105
106         % SOR Loop
107         w = get_optimal_w('SOR', u_init, N(i), eps(j));
108         [~, num_iter] = SOR_loop(u_init, tol, N(i), eps(j), w);
109         % Save number of iterations
110         output_b(fill_row,:)={'SOR',N(i),eps(j),w,num_iter};
111         fill_row = fill_row + 1;
112
113         % SSOR Loop
114         w = get_optimal_w('SSOR', u_init, N(i), eps(j));
```

```
115         [~, num_iter] = SSOR_loop(u_init, tol, N(i), eps(j), w);
116         % Save number of iterations
117         output_b(fill_row,:)={'SSOR',N(i),eps(j),w,num_iter};
118         fill_row = fill_row + 1;
119
120         % Kaczmarz Loop
121         [~, num_iter] = kaczmarz_loop(u_init, tol, N(i), eps(j));
122         % Save number of iterations
123         output_b(fill_row,:)={'Kaczmarz',N(i),eps(j),NaN,num_iter};
124         fill_row = fill_row + 1;
125     end
126 end
127
128 %% Starting Data (c)
129 output_c = table('Size',sz,'VariableTypes',var_types,'VariableNames',var_names
        );
130 fill_row = 1;
131 for i = 1:length(N)
132     % (c) initial u
133     u_init = starting_data_c(N(i));
134
135     for j = 1:length(eps)
136         % Point Jacobi Loop
137         [~, num_iter] = point_jacobi_loop(u_init, tol, N(i), eps(j));
138         % Save number of iterations
139         output_c(fill_row,:)={'Point Jacobi',N(i),eps(j),NaN,num_iter};
140         fill_row = fill_row + 1;
141
142
143         % Weighted Jacobi Loop
144         w = get_optimal_w('Weighted Jacobi', u_init, N(i), eps(j));
145         [~, num_iter] = weighted_jacobi_loop(u_init, tol, N(i), eps(j), w);
146         % Save number of iterations
147         output_c(fill_row,:)={'Weighted Jacobi',N(i),eps(j),w,num_iter};
148         fill_row = fill_row + 1;
149
150         % Gauss-Seidel Loop
151         [~, num_iter] = gauss_seidel_loop(u_init, tol, N(i), eps(j));
152         % Save number of iterations
153         output_c(fill_row,:)={'Gauss-Seidel',N(i),eps(j),NaN,num_iter};
154         fill_row = fill_row + 1;
155
156         % Red-Black Gauss-Seidel Loop
157         [~, num_iter] = rb_gauss_seidel_loop(u_init, tol, N(i), eps(j));
158         % Save number of iterations
159         output_c(fill_row,:)={'Red-Black Gauss-Seidel',N(i),eps(j),NaN,
                num_iter};
160         fill_row = fill_row + 1;
161
162         % SOR Loop
163         w = get_optimal_w('SOR', u_init, N(i), eps(j));
```

```
        [~, num_iter] = SOR_loop(u_init, tol, N(i), eps(j), w);
        % Save number of iterations
        output_c(fill_row,:)={'SOR',N(i),eps(j),w,num_iter};
        fill_row = fill_row + 1;

        % SSOR Loop
        w = get_optimal_w('SSOR', u_init, N(i), eps(j));
        [~, num_iter] = SSOR_loop(u_init, tol, N(i), eps(j), w);
        % Save number of iterations
        output_c(fill_row,:)={'SSOR',N(i),eps(j),w,num_iter};
        fill_row = fill_row + 1;

        % Kaczmarz Loop
        [~, num_iter] = kaczmarz_loop(u_init, tol, N(i), eps(j));
        % Save number of iterations
        output_c(fill_row,:)={'Kaczmarz',N(i),eps(j),NaN,num_iter};
        fill_row = fill_row + 1;
    end
end
```

6 MATLAB Script: ω Sensitivity Plot

```
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This program iterates Weighted Jacobi, SOR, and SSOR 100 times to solve
% the PDE of interest using varying values of omega. After cycling through
% the various values of omega, a plot is graphed of the max residual as a
% function of omega. This shows the sensitivity of convergence as a
% function of omega.

N = 128; % Number of interior points
eps = 1;
tol = 10^-7;

% Weighted Jacobi
w_wj = 0.91:0.01:.95;
max_residual_wj = zeros(size(w_wj));
for i = 1:length(w_wj)
    % (a) initial u
    u = starting_data_a(N);

    % Weighted Jacobi 100 iterations
    for j = 1:100
        u = weighted_jacobi_iteration(u, N, eps, w_wj(i));
    end
    % Save number of iterations
    max_residual_wj(i) = max(max(abs(u)));
end
figure(1)
plot(w_wj,max_residual_wj)
```

```
xlabel('\omega')
ylabel('max|u_{ij}|')

% SOR
w_SOR = 1.71:0.01:1.99;
max_residual_SOR = zeros(size(w_SOR));
for i = 1:length(w_SOR)
    u = starting_data_c(N);
        % SOR 100 iterations
    for j = 1:100
        u = SOR_iteration(u, N, eps, w_SOR(i));
    end
    % Save number of iterations
    max_residual_SOR(i) = max(max(abs(u)));
end
figure(2)
plot(w_SOR, max_residual_SOR)
xlabel('\omega')
ylabel('max|u_{ij}|')

% SSOR
w_SSOR = 1.71:0.01:1.99;
max_residual_SSOR = zeros(size(w_SSOR));
for i = 1:length(w_SSOR)
    u = starting_data_c(N);
        % SOR 100 iterations
    for j = 1:100
        u = SSOR_iteration(u, N, eps, w_SSOR(i));
    end
    % Save number of iterations
    max_residual_SSOR(i) = max(max(abs(u)));
end
figure(3)
plot(w_SSOR, max_residual_SSOR)
xlabel('\omega')
ylabel('max|u_{ij}|')
```

7 MATLAB Script: SSOR with $\omega = 1$

```
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This program cycles through all the values of N and epsilon for the given
% PDE of interest with the given initial data which can be set in line
% 15. The PDE is solved with SSOR with omega=1 to get symmetrized
% Gauss-Seidel. The number of iterations it takes to converge is then
% stored.

% Values of N and epsilon to cycle through
N = [16,32,64,128];
eps = [1,.1,.01,.001];
```

```
13
14 num_iters = [];
15 for i = 1:length(N)
16     for j = 1:length(eps)
17         % Set initial data
18         u = starting_data_c(N(i));
19         % remember, one iteration actually includes two sweeps
20         num_iter = 0;
21         % Set omega = 1 to get symmetrized Gauss-Seidel
22         w = 1;
23         % Solve the PDE using SSOR
24         while max(max(abs(u))) > tol
25             u = SSOR_iteration(u, N(i), eps(j), w);
26             num_iter = num_iter + 1;
27         end
28         % Store the number of iterations it took
29         num_iters = [num_iters; num_iter];
30     end
31 end
```

8 MATLAB Script: 10 Kaczmarz Iterations

```
1  % Author: Nicolas Guerra
2  % Date: January 29, 2023
3  %
4  % This program runs 10 iterations of the Kaczmarz method cyclying through
5  % each value of N and epsilon. The max residual after 10 iterations is then
6  % stored. The initial starting data is set in line 15.
7
8  N = [16,32,64,128];
9  eps = [1,.1,.01,.001];
10
11 % Data (a)
12 max_residuals_a = [];
13 for i = 1:length(N)
14     for j = 1:length(eps)
15         u = starting_data_a(N(i));
16         % Construct matrix for Kaczmarz for interior points
17         A = diag(-ones(N(i)^2,1)*(2*eps(j)+2))+...
18             diag(ones(N(i)^2-1,1)*eps(j),1)+...
19             diag(ones(N(i)^2-1,1)*eps(j),-1)+...
20             diag(ones(N(i)^2-N(i),1),N(i))+...
21             diag(ones(N(i)^2-N(i),1),-N(i));
22         for ii = N(i):N(i):(N(i)^2-N(i))
23             A(ii,ii+1) = 0;
24             A(ii+1,ii) = 0;
25         end
26         % Since A is mainly zeros, store as sparse matrix
27         A = sparse(A);
28         % Kaczmarz method
29         for iter = 1:10
```

```
            u = kaczmarz_iteration(u, A);
        end
        % Store max residual
        max_residuals_a = [max_residuals_a; max(max(abs(u)))];
    end
end
```

9 MATLAB Helper Functions

```
function [u_init] = starting_data_a(N)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function takes in a value N and returns a matrix of size (N+2)^2
% containing the initial data as specific in (a) of relaxation project
% INPUT: (int) N
% OUTPUT: (matrix of size (N+2)^2) u
    u_init = zeros(N+2,N+2);
    for j = 2:(N+1)
        for k = 2:(N+1)
            u_init(j,k) = (-1)^(j+k);
        end
    end
end
```

```
function [u_init] = starting_data_b(N)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function takes in a value N and returns a matrix of size (N+2)^2
% containing the initial data as specific in (b) of relaxation project
% INPUT: (int) N
% OUTPUT: (matrix of size (N+2)^2) u
    u_init = zeros(N+2,N+2);
    h = 1/(N+1);
    for j = 2:(N+1)
        for k = 2:(N+1)
            u_init(j,k) = sin(8*pi*j*h)*sin(8*pi*k*h);
        end
    end
end
```

```
function [u_init] = starting_data_c(N)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function takes in a value N and returns a matrix of size (N+2)^2
% containing the initial data as specific in (c) of relaxation project
% INPUT: (int) N
% OUTPUT: (matrix of size (N+2)^2) u
    u_init = zeros(N+2,N+2);
    h = 1/(N+1);
```

```
    for j = 2:(N+1)
        for k = 2:(N+1)
            u_init(j,k) = sin(pi*j*h)*sin(pi*k*h);
        end
    end
end
```

```
function [new_u] = point_jacobi_iteration(u, N, eps)
% Author: Nicolas Guerra
% Date: January 6, 2023
%
% This function performs one iteration of point Jacobi for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% INPUT: (matrix) u, (int) N, (double) eps
% OUTPUT: (matrix) u
    new_u = u;
    for j = 2:(N+1)
        for k = 2:(N+1)
            Au = eps*u(j+1,k)+eps*u(j-1,k)-(2*eps+2)*u(j,k)+u(j,k+1)+u(j,k-1);
            new_u(j,k) = (-(2*eps+2)*u(j,k) - Au)/(-(2*eps+2));
        end
    end
end
```

```
function [u, num_iter] = point_jacobi_loop(u_init, tol, N, eps)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs point Jacobi iterations until the desired
% tolerance is reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0;
    u = u_init;
    while max(max(abs(u))) > tol
        u = point_jacobi_iteration(u, N, eps);
        num_iter = num_iter + 1;
    end
end
```

```
function [new_u] = weighted_jacobi_iteration(u, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of weighted Jacobi for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% INPUT: (matrix) u, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) new_u
    new_u = u;
    for j = 2:(N+1)
        for k = 2:(N+1)
            Au = eps*u(j+1,k)+eps*u(j-1,k)-(2*eps+2)*u(j,k)+u(j,k+1)+u(j,k-1);
```

```
            new_u(j,k) = (-(2*eps+2)*u(j,k) - w*Au)/(-(2*eps+2));
        end
    end
end
```

```
function [u, num_iter] = weighted_jacobi_loop(u_init, tol, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs weighted Jacobi iterations until the desired
% tolerance is reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0;
    u = u_init;
    while max(max(abs(u))) > tol
        u = weighted_jacobi_iteration(u, N, eps, w);
        num_iter = num_iter + 1;
    end
end
```

```
function [u] = gauss_seidel_iteration(u, N, eps)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of Gauss-Seidel for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% INPUT: (matrix) u, (int) N, (double) eps
% OUTPUT: (matrix) u
    for j = 2:(N+1)
        for k = 2:(N+1)
            % Doesn't include u you are currently updating in equation
            Au = eps*u(j+1,k)+eps*u(j-1,k)+u(j,k+1)+u(j,k-1);
            u(j,k) = (-Au)/(-(2*eps+2));
        end
    end
end
```

```
function [u, num_iter] = gauss_seidel_loop(u_init, tol, N, eps)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs Gauss-Seidel iterations until the desired
% tolerance is reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0;
    u = u_init;
    while max(max(abs(u))) > tol
        u = gauss_seidel_iteration(u, N, eps);
        num_iter = num_iter + 1;
    end
```

```
end
```

```
function [u] = rb_gauss_seidel_iteration(u, N, eps)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of Red-Black Gauss-Seidel for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% INPUT: (matrix) u, (int) N, (double) eps
% OUTPUT: (matrix) u
for j = 2:(N+1)
    for k = 2:2:(N+1)
        zigzag_k = k+mod(j,2);
        % Black
        % Don't include u you are currently updating in equation
        Au = eps*u(j+1,zigzag_k)+eps*u(j-1,zigzag_k)+u(j,zigzag_k+1)+u(j,
            zigzag_k-1);
        u(j,zigzag_k) = (-Au)/(-(2*eps+2));
    end
end
for j = 2:(N+1)
    for k = 2:2:(N+1)
        zigzag_k = k+mod(j+1,2);
        % Red
        % Don't include u you are currently updating in equation
        Au = eps*u(j+1,zigzag_k)+eps*u(j-1,zigzag_k)+u(j,zigzag_k+1)+u(j,
            zigzag_k-1);
        u(j,zigzag_k) = (-Au)/(-(2*eps+2));
    end
end
```

```
function [u, num_iter] = rb_gauss_seidel_loop(u_init, tol, N, eps)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs red-black Gauss-Seidel iterations until the
% desired tolerance is reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0;
    u = u_init;
    while max(max(abs(u))) > tol
        u = rb_gauss_seidel_iteration(u, N, eps);
        num_iter = num_iter + 1;
    end
end
```

```
function [u] = SOR_iteration(u, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of Successive Over Relaxation for
```

```
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% This method uses immediate replacement.
% INPUT: (matrix) u, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) u
    for j = 2:(N+1)
        for k = 2:(N+1)
            Au = eps*u(j+1,k)+eps*u(j-1,k)-(2*eps+2)*u(j,k)+u(j,k+1)+u(j,k-1);
            u(j,k) = -w*Au/(-(2*eps+2)) + u(j,k);
        end
    end
end
```

```
function [u, num_iter] = SOR_loop(u_init, tol, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs SOR iterations until the desired tolerance is
% reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0;
    u = u_init;
    while max(max(abs(u))) > tol
        u = SOR_iteration(u, N, eps, w);
        num_iter = num_iter + 1;
    end
end
```

```
function [u] = SSOR_iteration(u, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of Symmetric Successive Over Relaxation
    for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% This method uses immediate replacement
% INPUT: (matrix) u, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) u
    for j = 2:(N+1)
        for k = 2:(N+1)
            Au = eps*u(j+1,k)+eps*u(j-1,k)-(2*eps+2)*u(j,k)+u(j,k+1)+u(j,k-1);
            u(j,k) = -w*Au/(-(2*eps+2)) + u(j,k);
        end
    end
    for j = (N+1):-1:2
        for k = (N+1):-1:2
            Au = eps*u(j+1,k)+eps*u(j-1,k)-(2*eps+2)*u(j,k)+u(j,k+1)+u(j,k-1);
            u(j,k) = -w*Au/(-(2*eps+2)) + u(j,k);
        end
    end
end
```

```
function [u, num_iter] = SSOR_loop(u_init, tol, N, eps, w)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs SSOR iterations until the desired tolerance is
% reached.
% INPUT: (matrix) u_init, (double) tol, (int) N, (double) eps, (double) w
% OUTPUT: (matrix) u, (int) num_iter
    num_iter = 0; % remember, one iteration actually includes two sweeps
    u = u_init;
    while max(max(abs(u))) > tol
        u = SSOR_iteration(u, N, eps, w);
        num_iter = num_iter + 1;
    end
end
```

```
function [u_new] = kaczmarz_iteration(u,A)
% Author: Nicolas Guerra
% Date: January 29, 2023
%
% This function performs one iteration of Kaczmarz for
% the diff eq: -(epsilon*u_xx+u_yy)=0 with zero at the boundaries
% INPUT: (matrix) u, (matrix) A
% OUTPUT: (matrix) u_new
    A = sparse(A);
    u=u(2:end-1,2:end-1);
    N = size(u,1); % number of interior points
    x = reshape(u, N^2 ,1);
    for i = 1:N^2
        r_i = -A(i,:)*x; % residual of ith row
        delta_i = r_i/norm(A(i,:))^2;
        x = x + delta_i*A(i,:)';
    end
    reshaped_x = reshape(x, N, N);
    u_new = zeros(size(reshaped_x)+2);
    u_new(2:end-1,2:end-1)=reshaped_x;
end
```