Coding Report

Joe Lenning

1 Problem Description

This report presents the results obtained from iteratively solving the system of linear equations Ax = b using three different methods: the Jacobi Method, the Gauss-Seidel method, and the Successive Over-Relaxation (SOR) method. The Jacobi Method begins by expressing each variable x_i (i = 1 to n) in terms of the other variables and constants in the nxn matrix. With an initial guess for each variable, the method computes a solution. The obtained solution is then utilized to update the system and solve for the next variable, this process continues to find a precise solution. The Gauss-Seidel method shares similarities with the Jacobi Method but offers a notable improvement. Instead of using the initial guess values throughout the iteration, the Gauss-Seidel method employs the most recently updated values. This means that each variable xi is calculated using the most current value xi. By utilizing these updated values during the iterative procedure, the Gauss-Seidel method tends to converge faster than the Jacobi Method. Lastly, the SOR method introduces an additional parameter called omega (ω). Similar to the previous methods, it iteratively solves the system of equations. However, the SOR method aims to enhance convergence by reducing the spectral radius. By carefully selecting an appropriate value for omega, the SOR method effectively decreases the spectral radius, resulting in accelerated convergence. This parameter introduces an element of relaxation into the iterative process, enabling faster and more accurate solutions. Overall, these iterative methods provide valuable tools for solving systems of linear equations. While the Jacobi Method, Gauss-Seidel method, and SOR method share similarities, each offers its own advantages in terms of convergence speed and accuracy.

2 Results

For our basic Ax= b

$$\begin{bmatrix} 4 & 1 & -1 \\ -1 & 3 & 1 \\ 2 & 2 & 6 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 5 \\ -4 \\ 1 \end{bmatrix}$$

We first found our x values at an approximation of [0,0,0]

$$\begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} \left(\frac{-x2-x3+5}{4}\right) \\ \frac{x1+x3-4}{3} \\ \frac{-2x1-2x2+1}{6} \end{bmatrix} \text{plugging in 0 we get } \begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} 5/4 \\ -4/3 \\ 1/6 \end{bmatrix}$$

We then plug in our last results into our next iteration to find our next set of solutions, and repeat the process

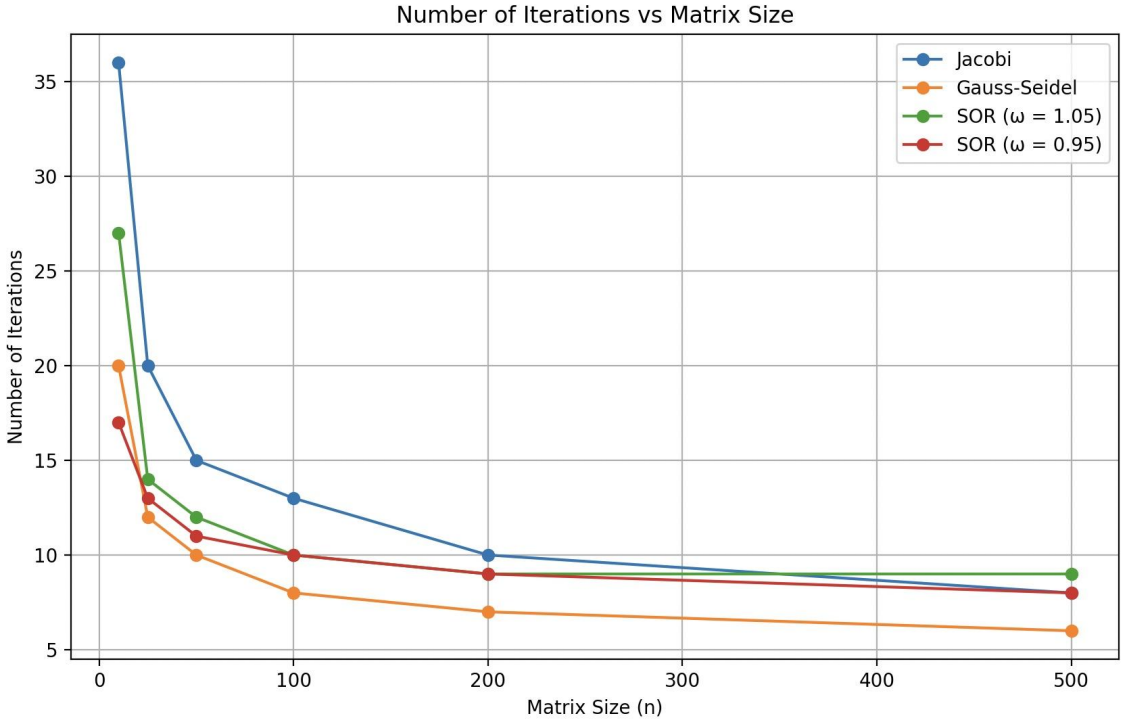
For Gauss-Seidel and SOR we would just the most recent finding of x in our iteration. As we can see below:

Given that we know x1 is $5/4$ we use this in our solution for x2

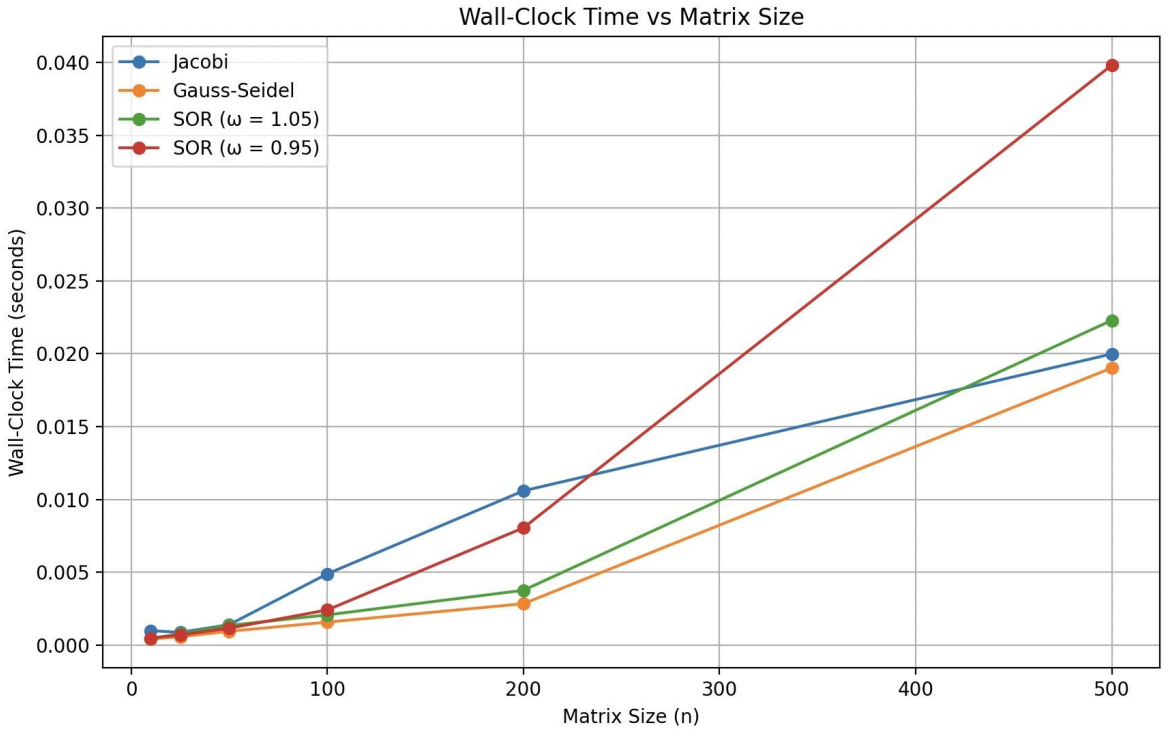
$$x2 = \frac{5/4 - 0 - 4}{3}, x2 = -11/12$$

$$x3 = \frac{-2(5/4) - 2(-11/12) + 1}{6} = 1/18$$

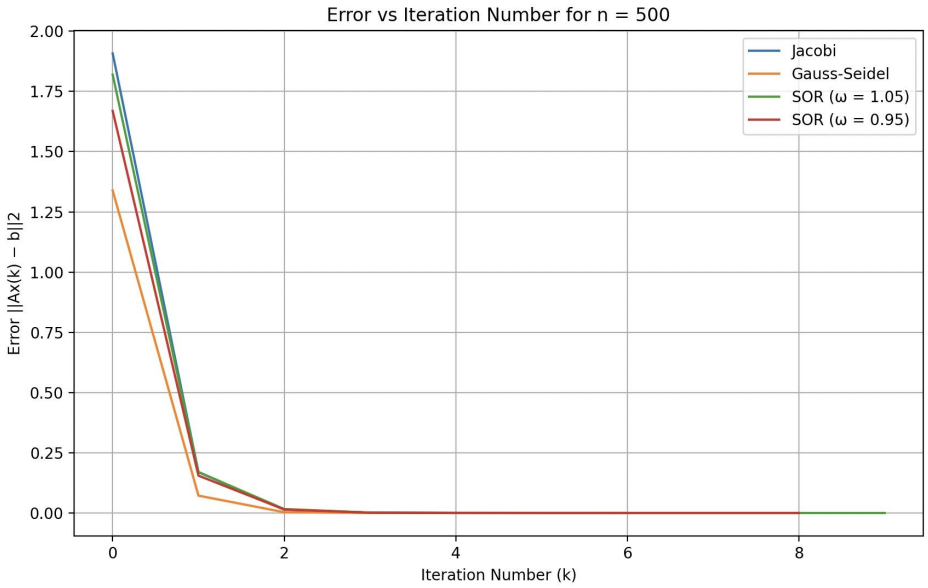
Specific Results:
All methods eventually converged to the same result of [1.45 -0.8375 -0.0375], which is the true result of our problem of Ax = b



This graph should be upward-sloping. As more and more elements are introduced into the system, there are more things to compute and thus more iterations are needed to find a solution. However, it is true that both SOR methods are the fastest. This is because it reduces the spectral radius and thus it makes solving the system much faster.



From the graph, we see that SOR was the slowest(wall clock). This does make sense because there are more operations involved in both types of SOR methods, so while it may converge within fewer iterations the actual time it takes may be longer. Additionally, our random matrices may be been more complex than other random matrices so it would also take longer to find solutions to them.



We see that our graph appears to be correct because we see that after more and more iterations our error decreases. Additionally, we see that they all methods converge to the same error(0.00) in about 2-3 iterations, meaning after 2-3 iterations we have found our true solution to Ax =b .

For the case of family A = randn(n, n) , it would be the case that it would take less time(wall clock) to find solutions. In our original A, we took n/2 * I which involved more operations. However now we have a few fewer operations for our new base A, thus our methods would be quicker and thus finds a true solution quicker. At larger values of n I suspect similar results

3 Collaboration:

None

4 Academic Integrity

On my personal integrity as a student and member of the UCD community, I have not given nor received any unauthorized assistance on this assignment.

5 Appendix

```
def jacobi_method(A, b, x0, max_iterations, tolerance):
    n = len(A)
    x = np.copy(x0)
    for k in range(max_iterations):
        x_new = np.zeros_like(x)
        for i in range(n):
            sum_val = np.dot(A[i, :i], x[:i]) + np.dot(A[i, i+1:], x[i+1:])
            x_new[i] = (b[i] - sum_val) / A[i, i]
```

```
        if np.linalg.norm(x_new - x) < tolerance:
            return x_new
        x = np.copy(x_new)
    return x
def gauss_seidel(A, b, x0, max_iterations, tolerance):
    n = len(A)
    x = x0.copy()
    for i in range(max_iterations):
        x_new = np.zeros(n)
        for j in range(n):
            s1 = np.dot(A[j, :j], x_new[:j])
            s2 = np.dot(A[j, j + 1:], x[j + 1:])
            x_new[j] = (b[j] - s1 - s2) / A[j, j]
        if np.allclose(x, x_new, rtol=tolerance):
            return x_new
        x = x_new
    return x

import numpy as np

import numpy as np

def sor_method(A, b, x0, max_iterations, tolerance, omega):
    n = len(A)
    x = np.copy(x0)
    omega = float(omega)
    for k in range(int(max_iterations)):
        x_new = np.copy(x)
        for i in range(n):
            sum_val = np.dot(A[i, :i], x_new[:i]) + np.dot(A[i, i+1:], x[i+1:])
            x_new[i] = (1 - omega) * x[i] + (omega / A[i, i]) * (b[i] -
sum_val)
        if np.linalg.norm(x_new - x) / np.linalg.norm(x_new) < tolerance:
            return x_new
        x = np.copy(x_new)
    return x

# Example usage
A = np.array([[4, 1, -1], [-1, 3, 1], [2, 2, 6]])
b = np.array([5, -4, 1])
x0 = np.zeros(3)

# SOR method with omega = 1.05
omega_1 = 1.05
max_iterations_1 = 200
tolerance = 1e-8
solution_1 = sor_method(A, b, x0, max_iterations_1, tolerance, omega_1)
print("SOR (omega = 1.05) solution:", solution_1)
```

```
omega_2 = 0.95
max_iterations_2 = 10000
tolerance = 1e-8
solution_2 = sor_method(A, b, x0, max_iterations_2, tolerance, omega_2)
print("SOR (omega = 0.95) solution:", solution_2)


A = np.array([[4, 1, -1], [-1, 3, 1], [2, 2, 6]])
b = np.array([5, -4, 1])
x0 = np.zeros(3)

# Jacobi method
Sol1 = jacobi_method(A, b, x0, 200, 1e-8)
print("Jacobi solution:", Sol1)

# Gauss-Seidel method
Sol2 = gauss_seidel(A, b, x0, 200, 1e-8)
print("Gauss-Seidel solution:", Sol2)

import time
import matplotlib.pyplot as plt


class RandomMatrixGenerator:
    def __init__(self, n):
        self.n = n

    def generate_matrix(self):
        I = np.eye(self.n)
        random_matrix = np.random.randn(self.n, self.n)
        A = (self.n / 2) * I + random_matrix
        return A


def jacobi_method(A, b, x0, max_iterations=1000, tolerance=1e-8):
    n = len(A)
    x = x0.copy()
    iterations = 0
    errors = []
    while iterations < max_iterations:
        x_new = np.zeros_like(x)
        for i in range(n):
            x_new[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i + 1:],
x[i + 1:])) / A[i, i]
        error = np.linalg.norm(A @ x_new - b)
        errors.append(error)
        if error < tolerance:
            return x_new, iterations, errors
```

```
        x = x_new
        iterations += 1
    return x, iterations, errors


def gauss_seidel_method(A, b, x0, max_iterations=1000, tolerance=1e-8):
    n = len(A)
    x = x0.copy()
    iterations = 0
    errors = []
    while iterations < max_iterations:
        for i in range(n):
            x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, i + 1:], x[i +
1:])) / A[i, i]
        error = np.linalg.norm(A @ x - b)
        errors.append(error)
        if error < tolerance:
            return x, iterations, errors
        iterations += 1
    return x, iterations, errors


def sor_method(A, b, x0, omega, max_iterations=1000, tolerance=1e-8):
    n = len(A)
    x = x0.copy()
    iterations = 0
    errors = []
    while iterations < max_iterations:
        x_new = np.zeros_like(x)
        for i in range(n):
            x_new[i] = (1 - omega) * x[i] + (omega / A[i, i]) * (
                    b[i] - np.dot(A[i, :i], x_new[:i]) - np.dot(A[i, i + 1:],
x[i + 1:]))
        error = np.linalg.norm(A @ x_new - b)
        errors.append(error)
        if error < tolerance:
            return x_new, iterations, errors
        x = x_new
        iterations += 1
    return x, iterations, errors


matrix_sizes = [10, 25, 50, 100, 200, 500]
jacobi_iterations = []
gauss_seidel_iterations = []
sor_1_iterations = []
sor_2_iterations = []
jacobi_times = []
gauss_seidel_times = []
```

```
sor_1_times = []
sor_2_times = []
jacobi_errors = []
gauss_seidel_errors = []
sor_1_errors = []
sor_2_errors = []

for n in matrix_sizes:
   generator = RandomMatrixGenerator(n)
   A = generator.generate_matrix()
   x0 = np.zeros(n)
   b = np.random.randn(n)

   # Jacobi Method
   start_time = time.time()
   jacobi_solution, jacobi_iter, jacobi_errors = jacobi_method(A, b, x0)
   end_time = time.time()
   jacobi_time = end_time - start_time

   # Gauss-Seidel Method
   start_time = time.time()
   gauss_seidel_solution, gauss_seidel_iter, gauss_seidel_errors =
gauss_seidel_method(A, b, x0)
   end_time = time.time()
   gauss_seidel_time = end_time - start_time

   # SOR Method (ω = 1.05)
   start_time = time.time()
   sor_1_solution, sor_1_iter, sor_1_errors = sor_method(A, b, x0, omega=1.05)
   end_time = time.time()
   sor_1_time = end_time - start_time

   # SOR Method (ω = 0.95)
   start_time = time.time()
   sor_2_solution, sor_2_iter, sor_2_errors = sor_method(A, b, x0, omega=0.95)
   end_time = time.time()
   sor_2_time = end_time - start_time

   jacobi_iterations.append(jacobi_iter)
   gauss_seidel_iterations.append(gauss_seidel_iter)
   sor_1_iterations.append(sor_1_iter)
   sor_2_iterations.append(sor_2_iter)

   jacobi_times.append(jacobi_time)
   gauss_seidel_times.append(gauss_seidel_time)
   sor_1_times.append(sor_1_time)
   sor_2_times.append(sor_2_time)

   # Plotting error vs iteration number (k) for n = 500
```

```
    if n == 500:
        plt.figure(figsize=(10, 6))
        plt.plot(range(jacobi_iter + 1), jacobi_errors, label='Jacobi')
        plt.plot(range(gauss_seidel_iter + 1), gauss_seidel_errors,
label='Gauss-Seidel')
        plt.plot(range(sor_1_iter + 1), sor_1_errors, label='SOR (ω = 1.05)')
        plt.plot(range(sor_2_iter + 1), sor_2_errors, label='SOR (ω = 0.95)')
        plt.xlabel('Iteration Number (k)')
        plt.ylabel('Error ||Ax(k) - b||2')
        plt.title('Error vs Iteration Number for n = 500')
        plt.legend()
        plt.grid(True)
        plt.show()

# Plotting wall-clock time vs n
plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, jacobi_times, marker='o', label='Jacobi')
plt.plot(matrix_sizes, gauss_seidel_times, marker='o', label='Gauss-Seidel')
plt.plot(matrix_sizes, sor_1_times, marker='o', label='SOR (ω = 1.05)')
plt.plot(matrix_sizes, sor_2_times, marker='o', label='SOR (ω = 0.95)')
plt.xlabel('Matrix Size (n)')
plt.ylabel('Wall-Clock Time (seconds)')
plt.title('Wall-Clock Time vs Matrix Size')
plt.legend()
plt.grid(True)
plt.show()

# Plotting iterations vs n
plt.figure(figsize=(10, 6))
plt.plot(matrix_sizes, jacobi_iterations, marker='o', label='Jacobi')
plt.plot(matrix_sizes, gauss_seidel_iterations, marker='o',
label='Gauss-Seidel')
plt.plot(matrix_sizes, sor_1_iterations, marker='o', label='SOR (ω = 1.05)')
plt.plot(matrix_sizes, sor_2_iterations, marker='o', label='SOR (ω = 0.95)')
plt.xlabel('Matrix Size (n)')
plt.ylabel('Number of Iterations')
plt.title('Number of Iterations vs Matrix Size')
plt.legend()
plt.grid(True)
plt.show()
```