Iterative Methods

Wanzhu Zheng

May 2023

1 Problem Description

This report demonstrates how to solve systems of linear equations $Ax = b$ using iterative methods: Jacobi, Gauss-Seidel, and SOR methods.
Jacobi:
Given a matrix A and vector b, we want to find our solution set x. First, we have our initial guess $x^{(0)} = (0, 0, ..., 0)^T$. Then, we find $x^{(1)=(x_1^{(1},x_2^{(1},...,x_i^{(k}}$ by plugging in $x^{(0)}$ into our system. We continue to iterate this process to find a sequence of increasingly better approximations $x^{(0)}, x^{(1)}, x^{(2)}, ...$
This method tends to be less accurate and requires more iterations because it doesn't update x_{i+1}while we solve the system. Instead, we have to solve for all values with $x^{(k)}$ before moving on to $x^{(k+1)}$.

Gauss-Siedel:
If the true solution is $x = (x_1, x_2, ..., x_n)$, if $x_1^{(k+1)}$ is a better approximation to the true value of x_1 than $x_1^(k)$ is, then once we have the new value $x_1(k + 1)$, we will use that to find $x_2(k + 1), ..., x_n(k + 1)$. This method is an improved version of Jacobi because it is more accurate since we are using updated values, and thus, will have less iterations, making it more efficient.

SOR:
For the SOR method, we have another variable ω which is the weight of the equation. When finding $x^{(k+1)}$ from $x^{(k)}$, we move a certain amount in a particular direction from $x^{(k)}$) to $x^{(k+1)}$. This direction is the vector $x^{(k+1)} - x^{(k)}$, since $x^{(k+1)} = x^{(k)} + (x^{(k+1)} - x^{(k)})$. Thus, we compute towards this direction to approach our true solution x. This method is a generalization of Gauss-Siedel method and tends to be more efficient because of the additional ω.

2 Results

Here, we demonstrate the result of our analysis with problems 2,3 and 4 from Problem Set 3 for $TOL < 10^{-8}$.
Jacobi Method:

$$\begin{bmatrix}1.45 & -0.8375 & -0.0375\end{bmatrix}$$

Gauss-Siedel Method:

$$\begin{bmatrix}1.45 & -0.8375 & -0.0375\end{bmatrix}$$

SOR Method with $\omega = 1.05$:

$$\begin{bmatrix}1.44999999 & -0.8375 & -0.0375\end{bmatrix}$$

SOR Method with $\omega = 0.95$:

$$\begin{bmatrix}1.45 & -0.8375 & -0.0375\end{bmatrix}$$

As expected, the vector x is the same for all methods. However, notice that the SOR Method for $\omega = 1.05$ $x_1 = 1.44999999$. This is due to rounding errors when the tolerance is 10^{-8}. The value is close enough to 1.45 such that it's not significant.

Now, create a random matrix A and vector b with the distribution:

$$A = \frac{n}{2} \cdot I \cdot randn(n,n) \tag{1}$$

for size $n = 10, 25, 50, 100, 200, 500$.

Wall-Clock Times for Jacobi Method for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 0.00558114 & 0.00283384 & 0.00797129 & 0.29888272 & 0.24597502 & 0.40766382 \end{bmatrix}$$

Iterations for Jacobi Method for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 32 & 28 & 18 & 14 & 11 & 9 \end{bmatrix}$$

Wall-Clock Times for Gauss-Siedel Method for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 0.01381803 & 0.01219487 & 0.02262402 & 0.06466413 & 0.28978205 & 1.01899886 \end{bmatrix}$$

Iterations for Gauss-Siedel Method for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 20 & 15 & 12 & 10 & 9 & 8 \end{bmatrix}$$

Wall-Clock Times for SOR Method ($\omega = 1.05$) for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 0.00516295 & 0.00922108 & 0.01689696 & 0.03259993 & 0.13355994 & 0.84033179 \end{bmatrix}$$

Iterations for SOR Method ($\omega = 1.05$) for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 20 & 16 & 14 & 12 & 11 & 10 \end{bmatrix}$$

Wall-Clock Times for SOR Method ($\omega = 0.95$) for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 0.01103115 & 0.01188803 & 0.02306199 & 0.04971695 & 0.12415266 & 0.64514375 \end{bmatrix}$$

Iterations for SOR Method ($\omega = 0.95$) for $n = 10, 25, 50, 100, 200, 500$:

$$\begin{bmatrix} 16 & 15 & 13 & 12 & 11 & 10 \end{bmatrix}$$

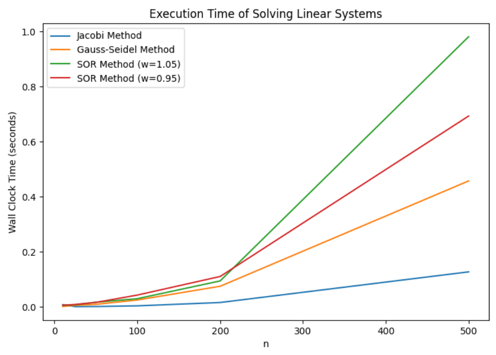

Figure 1. Execution Time of Solving Linear Systems

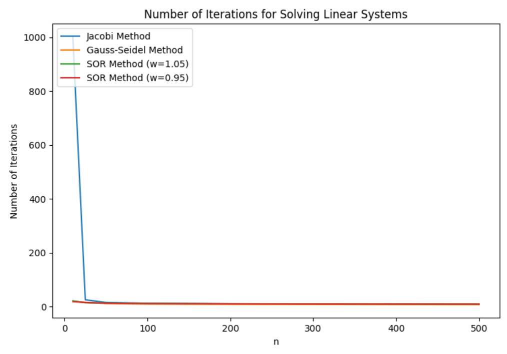

Figure 2. Number of Iterations for Solving Linear Systems

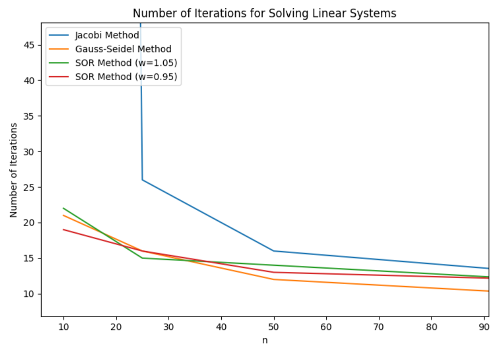

Figure 2.5. Zoomed in Version of Number of Iterations of Solving Linear Systems

Next, we find the residuals for a 500 x 500 matrix and plot residuals versus the number of iterations.

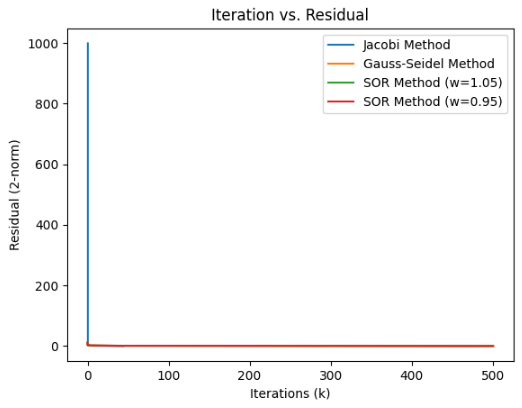

Figure 3. Iterations vs. Residual

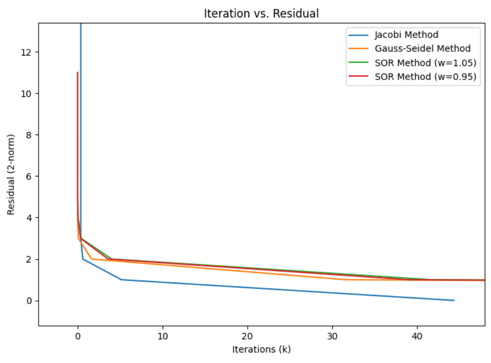

Figure 3.5. Zoomed in Version of Iterations vs. Residual

As seen from Figures 1 and 2, Jacobi Method takes less time, but requires more iterations while SOR requires more time, but takes less iterations. This is probably due to the fact that SOR has nested for loops, so it has worse run-time while Jacobi only takes one for loop. However, we see from the number of iterations that Jacobi takes more iterations than SOR, which is what we expected because SOR is faster. Similarly with Gauss-Siedel, it requires less time to run compared to SOR but more iterations. This is also because Gauss-Siedel only takes one for loop so the wall clock time will be faster. However, Gauss-Siedel requires less time and iterations compared to Jacobi, which shows that it's more efficient because each iteration is more accurate.
As seen in figure 3, all methods converge which is why the residuals end up being very small. However, Jacobi starts off with a very high residual, indicating the inaccuracy of the first couple iterations compared to Gauss-Siedel and SOR.

When trying experiments with $A = randn(n, n)$, I notice that a smaller n leads to more iterations compared to a larger n. I believe this is because a larger matrix could lead to less rounding errors or numerical instability. For a smaller matrix, these errors have a larger impact on the iterative process, requiring more iterations while having less impact on a smaller matrix, allowing for faster convergence.

3 Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import time


def jacobi(A, b, tol=1e-8, max_iter=1000):
    x = np.zeros_like(b, dtype=float)
    T = A - np.diag(np.diagonal(A))

    for k in range(max_iter):
        x_og = x.copy()
        x[:] = (b - np.dot(T, x)) / np.diagonal(A)
        if np.linalg.norm(x - x_og, ord=np.inf) < tol:
            break
    return x, k +1


def gauss_seidel(A, b, tol=1e-8, max_iter=1000):
    x = np.zeros_like(b, dtype=float)
    for k in range(max_iter):
        x_og = x.copy()
        for i in range(A.shape[0]):
            x[i] = (b[i] - np.dot(A[i, :i], x[:i]) - np.dot(A[i, (i + 1):], x_og[(i + 1):])) / A[i, i]
        if np.linalg.norm(x - x_og, ord=np.inf) < tol:
            break
    return x, k + 1


def sor(A, b, omega, tol=1e-8, max_iter=1000):
    x = np.zeros_like(b, dtype=float)

    for k in range(max_iter):
        x_og = x.copy()
        for i in range(A.shape[0]):
            sigma1 = np.dot(A[i, :i], x_og[:i])
            sigma2 = np.dot(A[i, i + 1:], x[i + 1:])
            x_og[i] = (1 - omega) * x[i] + (omega / A[i, i]) * (b[i] - sigma1 - sigma2)
        if np.linalg.norm(x - x_og, ord=np.inf) < tol:
            break
        x = x_og.copy()
    return x, k + 1


B = np.array([[4, 1, -1],
              [-1, 3, 1],
              [2, 2, 6]])
z = np.array([5, -4, 1])
print(jacobi(B, z))
print(gauss_seidel(B, z))
print(sor(B, z, 1.05))
print(sor(B, z, 0.95))


def matrix(n):
    A = ((n/2) * np.eye(n)) + np.random.normal(size=(n, n))
    return A


def sol(n):
    b = np.random.normal(size=(n, n))
    return b


def clock(n, type=0):
    A = matrix(n)
    b = sol(n)
    if type == 0:
        t0 = time.time()
        jacobi(A, b)
        t1 = time.time()
        total_time = t1 - t0
        return total_time, jacobi(A, b)[1]
    if type == 1:
        t0 = time.time()
        gauss_seidel(A, b)
        t1 = time.time()
        total_time = t1 - t0
        return total_time, gauss_seidel(A, b)[1]
    if type == 2:
        t0 = time.time()
        sor(A, b, 1.05)
        t1 = time.time()
        total_time = t1 - t0
        return total_time, sor(A, b, 1.05)[1]
    if type == 3:
        t0 = time.time()
        sor(A, b, 0.95)
        t1 = time.time()
        total_time = t1 - t0
        return total_time, sor(A, b, 0.95)[1]


def time_iter(type=0):
    times = np.empty((0, 0))
    iterations = np.empty((0, 0))
    if type == 0:
        times = np.append(times, (clock(10, 0)[0], clock(25, 0)[0], clock(50, 0)[0], clock(100, 0)[0], c
        iterations = np.append(iterations, (clock(10, 0)[1], clock(25, 0)[1], clock(50, 0)[1], clock(10
        return times, iterations
    if type == 1:
        times = np.append(times, (clock(10, 1)[0], clock(25, 1)[0], clock(50, 1)[0], clock(100, 1)[0], c
        iterations = np.append(iterations, (clock(10, 1)[1], clock(25, 1)[1], clock(50, 1)[1], clock(10
        return times, iterations
    if type == 2:
        times = np.append(times, (clock(10, 2)[0], clock(25, 2)[0],  clock(50, 2)[0], clock(100, 2)[0],
        iterations = np.append(iterations, (clock(10, 2)[1], clock(25, 2)[1], clock(50, 2)[1], clock(10
        return times, iterations
    if type == 3:
        times = np.append(times, (clock(10, 3)[0], clock(25, 3)[0],  clock(50, 3)[0], clock(100, 3)[0],
        iterations = np.append(iterations, (clock(10, 3)[1], clock(25, 3)[1], clock(50, 3)[1], clock(10
        return times, iterations


print(time_iter(type=0))
print(time_iter(type=1))
print(time_iter(type=2))
print(time_iter(type=3))

dims = [10, 25, 50, 100, 200, 500]
# size vs. time
plt.plot(dims, time_iter(type=0)[0])
plt.plot(dims, time_iter(type=1)[0])
plt.plot(dims, time_iter(type=2)[0])
plt.plot(dims, time_iter(type=3)[0])
plt.xlabel("n")
plt.ylabel("Wall Clock Time (seconds)")
plt.title("Execution Time of Solving Linear Systems")
plt.legend(["Jacobi Method", "Gauss-Seidel Method", "SOR Method (w=1.05)", "SOR Method (w=0.95)"], loc=
plt.show()

# size vs. iterations
plt.plot(dims, time_iter(type=0)[1])
plt.plot(dims, time_iter(type=1)[1])
plt.plot(dims, time_iter(type=2)[1])
plt.plot(dims, time_iter(type=3)[1])
plt.xlabel("n")
plt.ylabel("Number of Iterations")
plt.title("Number of Iterations for Solving Linear Systems")
plt.legend(["Jacobi Method", "Gauss-Seidel Method", "SOR Method (w=1.05)", "SOR Method (w=0.95)"], loc=
plt.show()


def residual(max_iter=1000, tol=1e-8, type=0):
    A = matrix(500)
    b = np.squeeze(sol(500))
    x = np.zeros_like(b, dtype=float)
    norms = np.empty((0, 0))
    iter = 0

    if type == 0:
        for k in range(max_iter):
            iter += 1
            r = b - A.dot(x)
            norm = np.linalg.norm(r, 2)
            norms = np.append(norms, norm)
            if norm < tol:
                break
            x[:] = (b - np.dot(A - np.diag(np.diagonal(A)), x)) / np.diagonal(A)
        return norms, iter
    if type == 1:
        for k in range(max_iter):
            iter += 1
            r = b - A.dot(x)
            norm = np.linalg.norm(r)
            norms = np.append(norms, norm)
            if norm < tol:
                break
            for i in range(A.shape[0]):
                x[i] = (b[i] - A[i, :i].dot(x[:i]) - A[i, i + 1:].dot(x[i + 1:])) / A[i, i]
        return norms, iter
    if type == 2:
        for k in range(max_iter):
            iter += 1
            r = b - A.dot(x)
            norm = np.linalg.norm(r)
            norms = np.append(norms, norm)
            if norm < tol:
                break
            for i in range(A.shape[0]):
                x[i] = (1 - 1.05) * x[i] + (1.05 / A[i, i]) * (
                            b[i] - A[i, :i].dot(x[:i]) - A[i, i + 1:].dot(x[i + 1:]))
        return norms, iter
    if type == 3:
        for k in range(max_iter):
            iter += 1
            r = b - A.dot(x)
            norm = np.linalg.norm(r)
            norms = np.append(norms, norm)
            if norm < tol:
                break
            for i in range(A.shape[0]):
                x[i] = (1 - 0.95) * x[i] + (0.95 / A[i, i]) * (
                        b[i] - A[i, :i].dot(x[:i]) - A[i, i + 1:].dot(x[i + 1:]))
        return norms, iter


# iteration vs. norm for 500x500 matrix
plt.plot(residual(type=0)[0], list(range(0, residual(type=0)[1])))
plt.plot(residual(type=1)[0], list(range(0, residual(type=1)[1])))
plt.plot(residual(type=2)[0], list(range(0, residual(type=2)[1])))
plt.plot(residual(type=3)[0], list(range(0, residual(type=3)[1])))
plt.xlabel("Iterations (k)")
plt.ylabel("Residual (2-norm)")
plt.title("Iteration vs. Residual")
plt.legend(["Jacobi Method", "Gauss-Seidel Method", "SOR Method (w=1.05)", "SOR Method (w=0.95)"], loc=
plt.show()
```