Successive Over-Relaxation Final Paper

John Bowllan, Josh Rubin, Robert Bernhardt

May 2017

Abstract

Successive Over-Relaxation (SOR) is an efficient iterative method for solving the system $Ax = b$. Since SOR builds upon both the Jacobi and Gauss-Seidel iterative methods, we first introduce the Jacobi method and show how it interprets linear systems in a different manner than FOR. Then we construct the Gauss-Seidel method and subsequently the SOR method by building off of our results from the Jacobi method. Afterwards, we illustrate this method on a trivial system and derive the convergence criteria by establishing bounds on the relaxation parameter; in theory and in practice, although not trivial to find, there exists a unique optimal value of ω for the fastest convergence for different linear systems. We show the iterative efficiency of the algorithm for various values of ω for solving a one-dimensional Model Poisson Problem. We conclude by demonstrating that SOR is more powerful than other iterative methods, such as FOR and Gauss-Seidel, but is still inferior to methods such as the Conjugate Gradient method.

Introduction

In the 21st century, modelers, analysts and forecasters use large systems of linear equations to inform decision making and predictions. For example, companies that host search engines, such as Google, manage matrices of significant dimensions and are required to solve enormous systems with the goal of updating dynamic page rank information almost instantaneously. As the complexities and capabilities of technology increase, so in turn, does the demand for fast, storage-efficient iterative methods to solve many practical problems we encounter today. Despite advances in computing technology, even the most powerful software programs are limited by finite computation speed and roundoff error that can make direct methods highly impractical for very large systems. Instead, numerous iterative methods have been studied in order to find solutions quickly and with a high level of precision. One such method is the Successive Over-Relaxation (SOR) method, the subject of this meta-analysis.

The First-Order Richardson (FOR) method constituted an early attempt to iteratively solve linear systems. This method first interprets the linear system $Ax = b$ as a root-finding problem, i.e. $0 = b - Ax$, adds a parameter ρ such that $\rho(x - x) = b - Ax$, and through algebraic manipulation solves successive approximations with $x^{(n+1)} = T_{FOR}x^{(n)} + b_{FOR}$ where $T_{FOR} = I - \frac{1}{\rho}A$ and $b_{FOR} = \frac{1}{\rho}b$. But FOR is very slow in practical problems. Nonetheless, searching for different ways to reinterpret the system $Ax = b$ allows us to use different mathematical tools that may aid in our efforts to improve iterative methods. Thus, we attempt to distinguish between different methods of transforming $Ax = b$ into an equivalent fixed-point problem of the form $x^{(n+1)} = Tx^{(n)} + c$, yielding faster convergence. Our focus will be centered around the Successive Over-Relaxation method. After deriving both the Jacobi and Gauss-Seidel methods in our quest to define the Successive Over-Relaxation method, we will show that SOR, with certain constraints on ω, greatly outperforms the precursor Jacobi and Gauss-Seidel methods.

1 Deriving the Successive Over-Relaxation Method

1.1 Jacobi Method

In the First-Order Richardson method, the system $Ax = b$ is interpreted as a root-finding problem. FOR can also be considered a matrix splitting problem: Let $A = M - N$ where $M = \rho I$ and $N = \rho I - A$. We can rewrite the FOR algorithm as $Mx^{(n+1)} = Nx^{(n)} + b$. Let us attempt to transform the system into an equivalent fixed-point problem in a different manner, by a different matrix decomposition. Consider the decomposition of A by

$$(D - L - U)x = b$$

where D, the diagonal matrix, L, the strictly lower triangular matrix, and U, the strictly upper triangular matrix, are as follows:

$$D = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & \ddots & 0 \\ 0 & 0 & a_{33} & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}, L = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \ddots & \ddots & 0 \\ a_{31} & a_{32} & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{n,n-1} & 0 \end{bmatrix} U = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \ddots & a_{2n} \\ 0 & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}$$

After distributing the vector x, this becomes

$$Dx - Lx - Ux = b,$$

and adding $(Lx + Ux)$ to both sides produces

$$Dx = Lx + Ux + b.$$

We may factor out x from the right-hand side of the equation and multiply both sides by D^{-1} to obtain

$$x = D^{-1}(L + U)x + D^{-1}b. \tag{1}$$

Thus, the system $Ax = b$ has been transformed into an equivalent form by Equation 1 and is solved by successive approximations

$$x^{(n+1)} = D^{-1}(L + U)x^{(n)} + D^{-1}b \tag{2}$$

where $x^{(n+1)}$ and $x^{(n)}$ are the (n+1)st and nth iterates respectively. Equation 2, an iterative method for computing successive iterates for $Ax = b$, is known as the Jacobi Method. Thus, $x_{JAC}^{(n+1)} = T_{JAC}\ x_{JAC}^{(n)} + b_{JAC}$ where $T_{JAC} = D^{-1}(L + U)$ and $b_{JAC} = D^{-1}b$.

Ultimately, in the Jacobi iteration scheme, all components x_i of the new approximation vector are obtained by only using the components of the previous approximation. Let $x =< x_1, x_2, ..., x_n >$ and

$b =< b_1, b_2, ..., b_n >$. In essence, for the Jacobi method, one solves the first equation for x_1, the second equation for x_2, up to the nth equation for x_n to obtain the following:

$$\begin{aligned}
x_1^{(n+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(n)} - a_{13}x_3^{(n)} - \cdots - a_{1n}x_n^{(n)}),\\
x_2^{(n+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(n)} - a_{23}x_3^{(n)} - \cdots - a_{2n}x_n^{(n)}),\\
&\vdots\\
x_n^{(n+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(n)} - a_{n2}x_2^{(n)} - \cdots - a_{n,n-1}x_{n-1}^{(n)}).
\end{aligned}$$

Thus, to obtain the (n+1)st iterate, substitute the values of the previous iterate $x^{(n)} =< x_1^{(n)}, x_2^{(n)}, ..., x_n^{(n)} >$ into the right-hand side of the equations above to yield the values of $x^{(n+1)} =< x_1^{(n+1)}, x_2^{(n+1)}, ..., x_n^{(n+1)} >$, the components of the new iterate. For the Jacobi method, the values $x_i^{(k)}$ are not updated until the (k+1)st iteration is complete.

1.2 Gauss-Seidel Method

Theoretically, with each successive approximation, the components of $x^{(n)}$ arrive closer and closer to the true solution. The Gauss-Seidel method seeks faster convergence by improving upon the Jacobi method. In the Gauss-Seidel method, for the (n+1)st iteration, the values $x_i^{(n+1)}$ are calculated using the most recent approximations of the other x values. In this case, the (n+1)st iterate is calculated as follows:

$$\begin{aligned}
x_1^{(n+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(n)} - a_{13}x_3^{(n)} - \cdots - a_{1n}x_n^{(n)}),\\
x_2^{(n+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(n+1)} - a_{23}x_3^{(n)} - \cdots - a_{2n}x_n^{(n)}),\\
&\vdots\\
x_n^{(n+1)} &= \frac{1}{a_{nn}}(b_n - a_{n1}x_1^{(n+1)} - a_{n2}x_2^{(n+1)} - \cdots - a_{n,n-1}x_{n-1}^{(n+1)}).
\end{aligned} \tag{3}$$

Equation 3 stems from a slightly different manipulation of the equation $Ax = b$ than that of the Jacobi method. When we derived the Jacobi method, we showed that matrix decomposition to compute successive iterates is $x^{(n+1)} = Tx^{(n)} + \tilde{b}$ for the Jacobi iteration matrix T and vector $\tilde{b}$. In essence, it is a different way of viewing the algorithm. Let us do the same for the Gauss-Seidel method. That is, let us find the iteration matrix and updated b vector for the Gauss-Seidel method in order to observe the difference between it and the Jacobi method. Once again, consider the matrix decomposition

$$(D - L - U)x = b$$

After distributing x

$$Dx - Lx - Ux = b,$$

we add Ux to both sides of the equation

$$Dx - Lx = Ux + b,$$

and factor out an x

$$(D - L)x = Ux + b.$$

Then, we multiply on both sides by $(D - L)^{-1}$ and obtain

$$x = (D - L)^{-1}Ux + (D - L)^{-1}b. \tag{4}$$

Once again, the system $Ax = b$ has been transformed into an equivalent form by Equation 4 and is solved by successive approximations

$$x^{(n+1)} = (D - L)^{-1}Ux^{(n)} + (D - L)^{-1}b.$$

So $x_{GS}^{(n+1)} = T_{GS}\ x_{GS}^{(n)} + b_{GS}$ where $T_{GS} = (D - L)^{-1}U$ and $b_{GS} = (D - L)^{-1}b$. The Gauss-Seidel method, as previously mentioned, achieves faster convergence than the Jacobi method does since it uses the most recent approximations of the components of the iterate instead of using all of the components of the previous iterate. Although Gauss-Seidel constitutes an improvement, the SOR method builds off the Gauss-Seidel method and seeks even faster convergence by introducing the notion of a weighted average.

1.3 Successive Over-Relaxation: The Weighted Average

A weighted average, in practice, is a mean calculated by giving values in a set a certain degree of influence. Each quantity is assigned a weight, and that weight determines the relative importance of each quantity. The SOR method extrapolates the Gauss-Seidel method in computing the next iterate by taking a weighted average of the previous iterate and the updated Gauss-Seidel iterate. The SOR method introduces a weighting factor ω and computes successive iterates as follows:

$$x_{SOR}^{(n+1)} = \omega x_{GS}^{(n+1)} + (1 - \omega)x_{SOR}^{(n)}. \tag{5}$$

In Equation 5, the respective weights assigned to $x_{GS}^{(n+1)}$ and $x_{SOR}^{(n)}$ are ω and $(1 - \omega)$. Hence, altering the value of ω will, in turn, alter the weights of the two iterates. In theory, ω is optimally chosen with the goal of accelerating the convergence of Gauss-Seidel. In section 3, we will explore the convergence criteria for SOR (i.e., the values of ω for which the method converges).

For consistency, let us analytically derive the method in matrix terms. That is, let us derive the method to compute successive iterates of the form $x^{(n+1)} = Tx^{(n)} + \tilde{b}$. Consider $(D - L - U)x = b$ and let us introduce the weighting factor ω such that $\omega(D - L - U)x = \omega b$. We may distribute ω and x to yield

$$\omega Dx - \omega Lx - \omega Ux = \omega b$$

Let $\omega D = \omega D + D - D$. Then our equation becomes

$$(\omega D + D - D)x - \omega Lx - \omega Ux = \omega b.$$

After distributing x and adding both sides of the equation by $(\omega Ux + Dx - \omega Dx)$, we obtain

$$(D - \omega L)x = (\omega U + (1 - \omega)D)x + \omega b,$$

which is equivalent to

$$x = (D - \omega L)^{-1}(\omega U + (1-\omega)D)x + (D - \omega L)^{-1}\omega b.$$

This is solved by successive approximations

$$x^{(n+1)} = (D - \omega L)^{-1}(\omega U + (1-\omega)D)x^{(n)} + (D - \omega L)^{-1}\omega b. \tag{6}$$

Equation 6 is the SOR iterative scheme. Thus, $x^{(n+1)}_{(SOR)} = T_{SOR}\ x^{(n)}_{SOR} + b_{SOR}$ where

$$T_{SOR} = (D - \omega L)^{-1}(\omega U + (1-\omega)D),\ \ b_{SOR} = (D - \omega L)^{-1}\omega b.$$

Note that if $\omega = 1$, then SOR simplifies to the Gauss-Seidel method. When $\omega > 1$, the method is considered an over-relaxation scheme and when $\omega < 1$, it is referred to as an under-relaxation scheme.

2 SOR on a Small Scale

Let us perform the SOR algorithm on a small scale example to understand how the algorithm works in practice. Then, on larger scale problems, we will have a more comprehensive understanding of how each iteration is performed.

Example

We illustrate this method with a 3x3 example. Consider the matrix A and vector b where

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 7 & 4 \\ 3 & 3 & 3 \end{bmatrix}, b = \begin{bmatrix} -6 \\ 7 \\ 6 \end{bmatrix}.$$

Although this system is easy to compute by hand, when the dimension of the system is sufficiently large, direct methods prove to be highly costly and inefficient in terms of storage and time. Thus, it is critical to solve the system by more efficient iterative methods such as SOR. Recall the SOR algorithm:

$$x^{(n+1)}_{SOR} = \omega x^{(n+1)}_{GS} + (1-\omega)x^{(n)}_{SOR}.$$

Let $\omega = 1.2$ and our initial guess be

$$x_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7}$$

We first need to find x^1_{GS}. Using Equation 3, the Gauss-Seidel components x^1_i are as follows:

$$\begin{aligned}
x^1_1 &= \frac{1}{a_{11}}(b_1 - a_{12}x^0_2 - a_{13}x^0_3) = \frac{1}{2}(-6 - 1(0) - (-3)(0)) = -3 \\
x^1_2 &= \frac{1}{a_{22}}(b_2 - a_{21}x^1_2 - a_{13}x^0_3) = \frac{1}{7}(7 - 1(-3) - (4)(0)) = \frac{10}{7} \\
x^1_3 &= \frac{1}{a_{33}}(b_3 - a_{31}x^1_2 - a_{13}x^1_2) = \frac{1}{1}(6 - 3(-3) - (3)(\frac{10}{7})) = \frac{75}{7} \approx 10.7143
\end{aligned}$$

Therefore,

$$x^1_{GS} = \begin{bmatrix} -3 \\ \frac{10}{7} \\ \frac{75}{7} \end{bmatrix}.$$

Then x^1_{SOR} is computed by

$$\begin{aligned} x^1_{SOR} &= \omega x^{(1)}_{GS} + (1-\omega)x^{(0)}_{SOR} \\ &= (1.2)\begin{bmatrix} -3 \\ \frac{10}{7} \\ \frac{75}{7} \end{bmatrix} + (1-1.2)\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} \frac{-18}{5} \\ \frac{12}{7} \\ \frac{90}{7} \end{bmatrix}. \end{aligned}$$

3 Convergence of Successive Over-Relaxation

SOR improves on earlier iterative methods by introducing a relaxation parameter ω to the Gauss-Seidel algorithm. However, the method does not exhibit its improved efficieny for any value of ω. In fact, for all square matrices there exists a unique optimal ω for the fastest convergence. Finding the value of $\omega_{OPTIMAL}$ is very difficult in practice and is most often attained by trial and error with different values of ω. However, it is critical to find an accurate approximation of $\omega_{OPTIMAL}$ since the resulting improvement of convergence can be considerable, which we will demonstrate in subsequent sections. In fact, $\omega_{OPTIMAL} \in (0, 2)$, the interval of convergence. To prove that this interval is, in fact, the interval of convergence, we must first prove the following lemma:

Lemma 3.1. *Let $A \in R^{n\times n}$.*
Then

$$\lim_{k\to\infty} A^k = 0_{n\times n}$$

if and only if $\rho(A) < 1$

Proof. One direction is relatively straightforward. Suppose $\lim_{k\to\infty} A^k = 0_{n\times n}$. Then, by definition, for any matrix norm,

$$\lim_{k\to\infty} ||A^k|| = 0.$$

Recall that $\rho(A) \leq ||A||$ for any matrix norm (for proof see [Layton and Sussman, 69]). Furthermore, recall that $\rho(A^k) = \rho(A)^k$. It follows that $\rho(A)^k \leq ||A^k||$. By the Squeeze Theorem, since $||A^k||$ tends to zero, so does $\rho(A)^k$.

Since we have a geometric sequence tending to zero, the common ratio must be less than 1, that is,

$$\rho(A) < 1 \tag{8}$$

The other direction is more involved; for proof, see [Kahan, 775]. □

With this lemma, we have an essential tool to prove the following theorem.

Theorem 3.2. *SOR converges for $\omega \in (0, 2)$ and diverges otherwise.*

Proof. For SOR, an equivalent iteration matrix is

$$T_{SOR} = (D + \omega L)^{-1}[(1 - \omega)D - \omega U]. \tag{9}$$

Let the eigenvalues of T_{SOR} be denoted

$$\sigma(T_{SOR}) = \{\mu_1, \mu_2, \ldots, \mu_n\}.$$

These eigenvalues are the zeros of the characteristic polynomial of T_{SOR}, which is of degree n and thus has the form

$$P(\mu) = \mu^n + C_{n-1}\mu^{n-1} + \ldots + C_1\mu + C_0,$$

for constants $\{C_i\}_{i=0}^{n-1}$. We can factor out the roots as follows:

$$P(\mu) = (\mu - \mu_1)(\mu - \mu_2)\cdots(\mu - \mu_n)$$

Setting $\mu = 0$, we obtain

$$\begin{aligned} P(0) &= (0 - \mu_1)(0 - \mu_2)\ldots(0 - \mu_n) \\ P(0) &= (-\mu_1)(-\mu_2)\ldots(-\mu_n) \\ P(0) &= (-1)^n(\mu_1)(\mu_2)\cdots(\mu_n) \\ P(0) &= (-1)^n\prod_{j=1}^{n}\mu_n. \end{aligned} \tag{10}$$

However, we can also express P(0) by

$$\begin{aligned} P(0) &= \det(0I - B) \\ &= \det(-B) \\ &= (-1)^n\det(B). \end{aligned} \tag{11}$$

Setting Equation 10 equal to Equation 11, we obtain

$$(-1)^n\det(B) = (-1)^n\prod_{j=1}^{n}\mu_n.$$

We may deduce that $\det(B) = \prod_{j=1}^{n}\mu_n$, that is, the determinant of the iteration matrix is the product of its eigenvalues and

$$\prod_{j=1}^{n}\mu_n = \det((D + \omega L)^{-1}) \cdot \det((1 - \omega)D - \omega U)$$

due to the multiplicative property of determinants. Note that both $(D+\omega L)$ and $((1-\omega)D-\omega U)$ are triangular matrices. The eigenvalues of a diagonal or triangular matrix are the entries in its main diagonal. Thus,

$$\det{(D+\omega L)^{-1}} = \det{(D)^{-1}} = \frac{1}{\det D} \tag{12}$$

because a lower triangular matrix has zeros along its main diagonal. By similar reasoning

$$\begin{aligned}\det(D-\omega D-\omega U) &= \det(D-\omega D) \\ &= ((1-\omega)d_{11})((1-\omega)d_{22})\cdots((1-\omega)d_{nn}) \\ &= (1-\omega)^n[d_{11}d_{22}\cdots d_{nn}]\end{aligned}$$

$$= (1-\omega)^n \det D \tag{13}$$

Therefore, from Equations 12 and 13, we have

$$T_{SOR} = (D+\omega L)^{-1}[(1-\omega)D-\omega U]$$

$$\det(T_{SOR}) = (\frac{1}{\det D})(1-\omega)^n \det D$$

$$= (1-\omega)^n$$

By the lemma, we know $\rho(T_{SOR}) < 1$ because the iteration matrix converges. We also know

$$\prod_{j=1}^{n} \mu_n = (1-\omega)^n. \tag{14}$$

Now, if all the eigenvalues of the iteration matrix were strictly less than $|1-\omega|$, then their product would be strictly less than $|1-\omega|^n$. Therefore, at least one eigenvalue is at least $|1-\omega|$, and since the spectral radius is by definition the magnitude of the largest eigenvalue, we have:

$$\rho(T_{SOR}) \geq |(1-\omega)|$$

Given that $\rho(T_{SOR}) \geq |(1-\omega)|$ and $\rho(T_{SOR}) < 1$, it follows that

$$1 > |1-\omega|,$$

or equivalently,

$$1 > 1-\omega > -1$$

We multiply the inequality by -1

$$-1 < \omega - 1 < 1$$

and adding 1 to both sides of the inequality, we obtain

$$0 < \omega < 2.$$

□

3.1 Low Dimension Example of Convergence and Divergence of SOR

Consider the 3x3 system where

$$A = \begin{bmatrix} 3 & -2 & 5 \\ 4 & -7 & -1 \\ 5 & -6 & 4 \end{bmatrix}, b = \begin{bmatrix} 2 \\ 19 \\ 13 \end{bmatrix}, x_{\text{TRUE}} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}.$$

Let our first guess be denoted $x_{\text{INITIAL}} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$. Consider Tables 2 and 3:

Iterations	x_1	x_2	x_3
1	0.6667	-2.3333	-1.0833
2	1.0417	-1.8869	-0.8824
3	0.7984	-2.1851	-1.0255
4	0.9796	-1.9713	-0.9314
⋮	⋮	⋮	⋮
94	0.9999	-2.0000	-1.0000
95	1	-2	-1

Table 2: SOR on 3x3 example problem with $\omega = 1.5$. Convergence is achieved.

Iterations	x_1	x_2	x_3
1	0.6667	-2.3333	-1.0833
2	0.4167	-2.6310	-1.2173
3	-0.1081	-3.2595	-1.5041
4	-1.2167	-4.5913	-2.1166
⋮	⋮	⋮	⋮
94	-6.2120×10^{29}	-7.4850×10^{29}	-3.4625×10^{29}
95	-1.3205×10^{30}	-1.5911×10^{30}	-0.7360×10^{30}
⋮	⋮	⋮	⋮
872	-3.8798×10^{284}	-4.6749×10^{284}	-2.1626×10^{284}
873	-8.2473×10^{284}	-9.9374×10^{284}	-4.5969×10^{284}
⋮	⋮	⋮	⋮

Table 3: SOR on 3x3 example problem with $\omega = -1$. Convergence is not achieved.

Table 2 depicts the SOR method's performance on the system with $\omega = 1.5$, which is indeed contained in our interval of convergence. A tolerance of convergence was not specified but it appears that the method converges to the true solution in approximately 95 iterations. Depending on the desired tolerance, the method will take either additional or fewer iterations to achieve a solution within the tolerance. Nevertheless, with $\omega = 1.5$, the method will converge to a solution. Table 3 shows the SOR method's performance on the same problem, but with $\omega = -1$, a value not contained in the interval of convergence. Evidently, the method never converges to the true solution. In fact, subsequent iterations significantly exacerbate the relative error. Setting $\omega = 3$, another value outside of the interval, yielded very similar behavior as depicted in Table 3.

4 Modern Applications: 1D Model Poisson Problem

The power of the Successive Over-Relaxation method reveals itself in problems of much larger scale. To illustrate this power, let us consider a 1D Model Poisson Problem. We are asked to find $u(x)$ defined on the interval $\Omega = (a, b)$ satisfying

$$\begin{gathered} -u''(x) = f(x), \quad a < x < b \\ u(a) = g_1, u(b) = g_2, \end{gathered}$$

where $f(x)$, g_1, and g_2 are given. In the problem, with $n = 100$, $a = 0$, and $b = 1$, we define h by

$$h = \frac{b-a}{n+1}.$$

Let $f(x) = \frac{x}{200}$. We use our x-points mesh ($xpts$) of 100 points equally spaced by h on the interval (a, b) to calculate our b vector where

$$b = \begin{bmatrix} h^2 f(x_1) + g(a) \\ h^2 f(x_2) \\ h^2 f(x_3) \\ \vdots \\ h^2 f(x_{n-2}) \\ h^2 f(x_{n-1}) \\ h^2 f(x_n) + g(b) \end{bmatrix}.$$

Thus, we are tasked with solving the system

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-2} \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} h^2 f(x_1) + g(a) \\ h^2 f(x_2) \\ h^2 f(x_3) \\ \vdots \\ h^2 f(x_{n-2}) \\ h^2 f(x_{n-1}) \\ h^2 f(x_n) + g(b) \end{bmatrix}$$

where $g(a) = g(b) = 0$. Consider "MATLAB Code 1", the script that carries out the SOR algorithm. In a helper script, we created an omega mesh, or a vector of different ω values equally spaced by a small constant on the interval $(0, 2)$. We then ran the function for each value of ω on the 1DMPP in order to observe how the method fares with different values of ω. At the same time, with each iteration, we stored the number of iterations the algorithm required to converge within a tolerance of 10^{-8}. Therefore, we could determine the rate of convergence of the algorithm for each ω in our omega mesh. Our initial guess was the zero vector of size $n \times 1$.

Consider Figure 1. Starting with $\omega \approx 0.01$, the iteration count monotonically decreases up to $\omega \approx 1.6$ where the iteration count achieves a minimum. After, for $\omega > 1.6$, the iteration count linearly increases. In theory, the formula for $\omega_{optimal}$ is as follows:

$$\omega_{optimal} = \frac{2}{1 + \sqrt{1 - (spr(T_{JAC}))^2}} \qquad [2]$$

For $\omega_{optimal} \approx 1.6$, the SOR algorithm converged in approximately 7,800 iterations, a significant improvement over the FOR algorithm (with $\rho = 2$) in Homework 5 on the same problem, which required approximately 37,574 iterations to converge. Thus, the SOR algorithm converged in approximately 21% of the number of iterations it required the FOR algorithm to converge, indicative of its significantly superior performance.

```
%% SOR Method
function [time,iteration] = SOR(A,n,b,x0,tol,itMax,omega)
% initialize it and xOld
it = 0;
xOld = x0;
tic % start timer
while it < itMax
    % SOR method for computing components of new iterate xNewSOR
    xNew(1) = (b(1) - A(1,2:n)*xOld(2:n))/A(1,1);
    for i = 2:n-1
       xNew(i) = (b(i) - A(i,1:i-1)*transpose(xNew(1:i-1)) ...
           -A(i,i+1:n)*xOld(i+1:n))/A(i,i);
    end
    xNew(n) = (b(n) - A(n,1:n-1)*transpose(xNew(1:n-1)))/A(n,n);
    xNewSOR = omega.*transpose(xNew) + (1-omega).*xOld;
    % calculate delta, deltaNorm, and xNorm
    delta = xNewSOR - xOld;
    deltaNorm = norm(delta);
    xNorm = norm(xNewSOR);
    % Convergence criteria
    if (deltaNorm < tol*xNorm)
        display(['Solution converged in ' num2str(it) ' iterations']);
        iteration = it;
        xApprox = xNewSOR;
        time = toc;
        return
    end
    xOld = xNewSOR; % set xOld to your new iterate
    it = it+1; % increment iterations
end
% display error message if convergence is not attained
error(['Convergence in ' num2str(itMax) ' iterations failed'])
```

MATLAB Code 1: The Successive Over-Relaxation (SOR) code for the 1D Model Poisson Problem

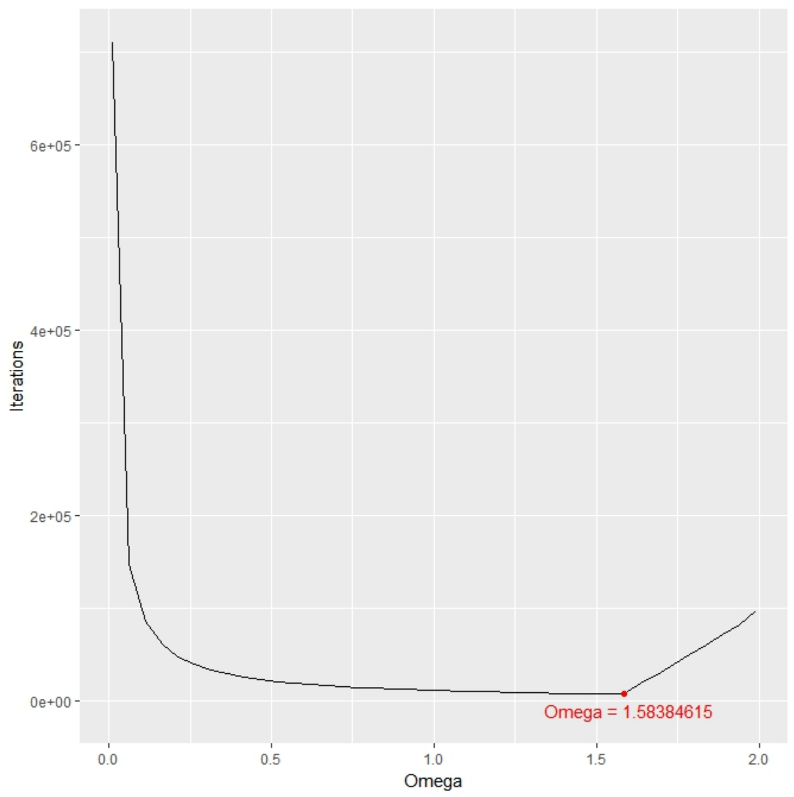

Figure 1: Illustration of how the SOR algorithm performs with regards to iteration count for different values of $\omega \in (0, 2)$.

5 The Big Picture

There are myriad ways to manipulate linear systems to exploit various mathematical properties. To place SOR in the grand scheme of prominent iterative methods, consider Figure 2 below, which provides the iteration counts of several iterative methods, including SOR, performed on the 1D Model Poisson Problem from section 4:

Performance of Iterative Methods on 1DMPP Problem	
Iterative Method	Iterations for Convergence
SOR ($\omega_{optimal} \approx 1.6$)	7,809
Gauss-Seidel	11,860
Jacobi	22,277
FOR ($\rho = 2$)	37,574
2-step FOR ($\rho_1 = 1, \rho_2 = 3$)	14,088
Steepest Descent	38,214
Conjugate Gradient	100

Figure 2: Synopsis of iterative methods studied in class and in this paper.

First, we wrote separate function files for the Jacobi and Gauss-Seidel methods, which can be found in the Appendix. Then, using the same criteria from the 1D Model Poisson Problem found in Section 4, we ran all of the methods listed in Figure 2 and determined when each method converged to the true solution within a tolerance of 10^{-8}. According to the table, both FOR and Steepest Descent converged the slowest while SOR converged faster than every iterative method with the exception of the Conjugate Gradient Method (which takes advantage of orthogonality and Krylov Subspaces). In theory, the Conjugate Gradient method converges in as many iterations as the dimension of the matrix A. Overall, the table above provides a framework for observing the evolution of iterative methods with the goal of achieving faster convergence. In the beginning of the term, we derived FOR and explored its convergence criteria. Striving to improve the convergence speed, we derived 2-step FOR and explored its convergence criteria by building off of regular FOR. As the table depicts, it took 2-step FOR approximately 63% fewer iterations to converge to the solution than that of regular FOR. In this paper, we improved on both the Jacobi and Gauss-Seidel methods by deriving the SOR method. Additionally, as the table depicts, we see a significant decrease in iterations necessary for convergence from Jacobi to Gauss-Seidel to SOR.

6 Conclusion

The source of these iterative methods stems from one equation: $Ax = b$. Manipulating this linear system into different, yet equivalent, forms allows us to use different mathematical tools to accelerate convergence. Contrary to the derivation of FOR, we explored a different matrix splitting as we constructed SOR from the Jacobi method and then the Gauss-Seidel method. Although the modifications of SOR are seemingly simple compared to Jacobi and Gauss-Seidel, these changes yield not only significant advantages in convergence over Jacobi and Gauss-Seidel, but also over FOR. Nevertheless, the SOR method is not the end-all method. As we saw in the previous section, there are even faster and more efficient iterative methods that either build upon SOR or approach manipulating the system $Ax = b$ in a completely different manner.

The Symmetric Successive Over-Relaxation (SSOR) and Conjugate Gradient (CG) methods are modern improvements over SOR. The SSOR method modifies SOR to ensure that the part of the algorithm approximating A is symmetric, while CG focuses on computing a global (rather than local) minimum at each iteration. For both of these methods, the SOR iteration matrix can serve as a preconditioner, a way of converting a linear system into an equivalent, but more computationally tractable formulation. In other words, a preconditioner P of a matrix A is a matrix such that $P^{-1}A$ has a smaller condition number than A, which is useful since the rate of convergence of most iterative methods increases as the condition number of the matrix decreases. We can devise a matrix P from the SOR method that can decrease the condition number of an ill-conditioned system. CG requires approximately $O(\sqrt{cond(A)})$ iterations per significant digit of accuracy. Thus, reducing the condition number via preconditioning is a viable solution in practical applications.

Ultimately, solving $Ax = b$, in many applications, is only a small portion of a much larger and more complex problem. Thus, optimizing these iterative methods will allow us both to solve modern problems more efficiently and to pursue problems of greater complexity.

7 Appendix

7.1 Jacobi Method

```
%% Jacobi Method
function [time,iteration] = Jacobi(A,n,b,x0,tol,itMax)
% initialize it and xOld
it = 0;
xOld = x0;
tic % start timer
while it < itMax
    % Jacobi method for computing components of new iterate xNewJac
    xNew(1) = (b(1) - A(1,2:n)*xOld(2:n))/A(1,1);
    for i = 2:n-1
       xNew(i) = (b(i) - A(i,1:i-1)*(xOld(1:i-1)) ...
           -A(i,i+1:n)*xOld(i+1:n))/A(i,i);
    end
    xNew(n) = (b(n) - A(n,1:n-1)*(xOld(1:n-1)))/A(n,n);
    xNewJac = transpose(xNew);
    % calculate delta, deltaNorm, and xNorm
    delta = xNewJac - xOld;
    deltaNorm = norm(delta);
    xNorm = norm(xNew);
    % Convergence criteria
    if (deltaNorm < tol*xNorm)
        display(['Jacobi solution converged in ' num2str(it) ' iterations']);
        iteration = it;
        xApprox = xNewJac;
        time = toc;
        return
    end
    xOld = xNewJac; % set xOld to your new iterate
    it = it+1; % increment iterations
end
% display error message if convergence is not attained
error(['Convergence in ' num2str(itMax) ' iterations failed'])
```

MATLAB Code 2: The Jacobi Method code for the 1D Model Poisson Problem.

7.2 Gauss-Seidel Method

```
%% GS Method
function [time,iteration] = GS(A,n,b,x0,tol,itMax)
% initialize it and xOld
it = 0;
xOld = x0;
tic % start timer
while it < itMax
    % Gauss-Seidel method for computing components of new iterate xNew
    xNew(1) = (b(1) - A(1,2:n)*xOld(2:n))/A(1,1);
    for i = 2:n-1
       xNew(i) = (b(i) - A(i,1:i-1)*transpose(xNew(1:i-1)) ...
           -A(i,i+1:n)*xOld(i+1:n))/A(i,i);
    end
    xNew(n) = (b(n) - A(n,1:n-1)*transpose(xNew(1:n-1)))/A(n,n);
    % Calculate delta, deltaNorm, and xNorm
    delta = transpose(xNew) - xOld;
    deltaNorm = norm(delta);
    xNorm = norm(xNew);
    % Convergence criteria
    if (deltaNorm < tol*xNorm)
        display(['GS solution converged in ' num2str(it) ' iterations']);
        iteration = it;
        xApprox = xNew';
        %display(xApprox)
        time = toc;
        return
    end
    xOld = xNew'; % set xOld to your new iterate
    it = it+1; % increment iterations
end
% display error message if convergence is not attained
error(['Convergence in ' num2str(itMax) ' iterations failed'])
```

MATLAB Code 3: The Gauss-Seidel Method code for the 1D Model Poisson Problem.

Honor Code

I have neither given nor received any unauthorized aid on this project. I have read through this project in its entirety and approve of all its contents.

...

John Anthony Bowllan
Josh Rubin
Robert Bernhardt

References

[1] Kendall Atkinson. *Introduction to Numerical Analysis.* Wiley, 2 edition, 1989.

[2] Brian Bradie. *A Friendly Introduction to Numerical Analysis.* Pearson, 1 edition, 2005.

[3] W. Kahan. *Numerical Linear Algebra.* Canadian Mathematical Society, 2 edition, 1966.

[4] Abdelwahab Kharab and Ronald B. Guenther. *An Introduction to Numerical Methods, A MATLAB Approach.* CRC Press, 3 edition, 2011.

[5] Rainer Kress. *Graduate Texts in Mathematics, Numerical Analysis.* Springer, 1 edition, 1998.

[6] William Layton and Myron Sussman. *Numerical Linear Algebra.* Lulu, 2014. http://www.lulu.com/spotlight/Layton_Sussman.

[7] MathWorks. MATLAB. Version R2017a. https://www.mathworks.com.