CHEMICAL ENGINEERING 2E04

Chapter 1 – Linear Algebraic Equations

Module 1C: Iterative Methods for Linear Systems

Dr. Jake Nease

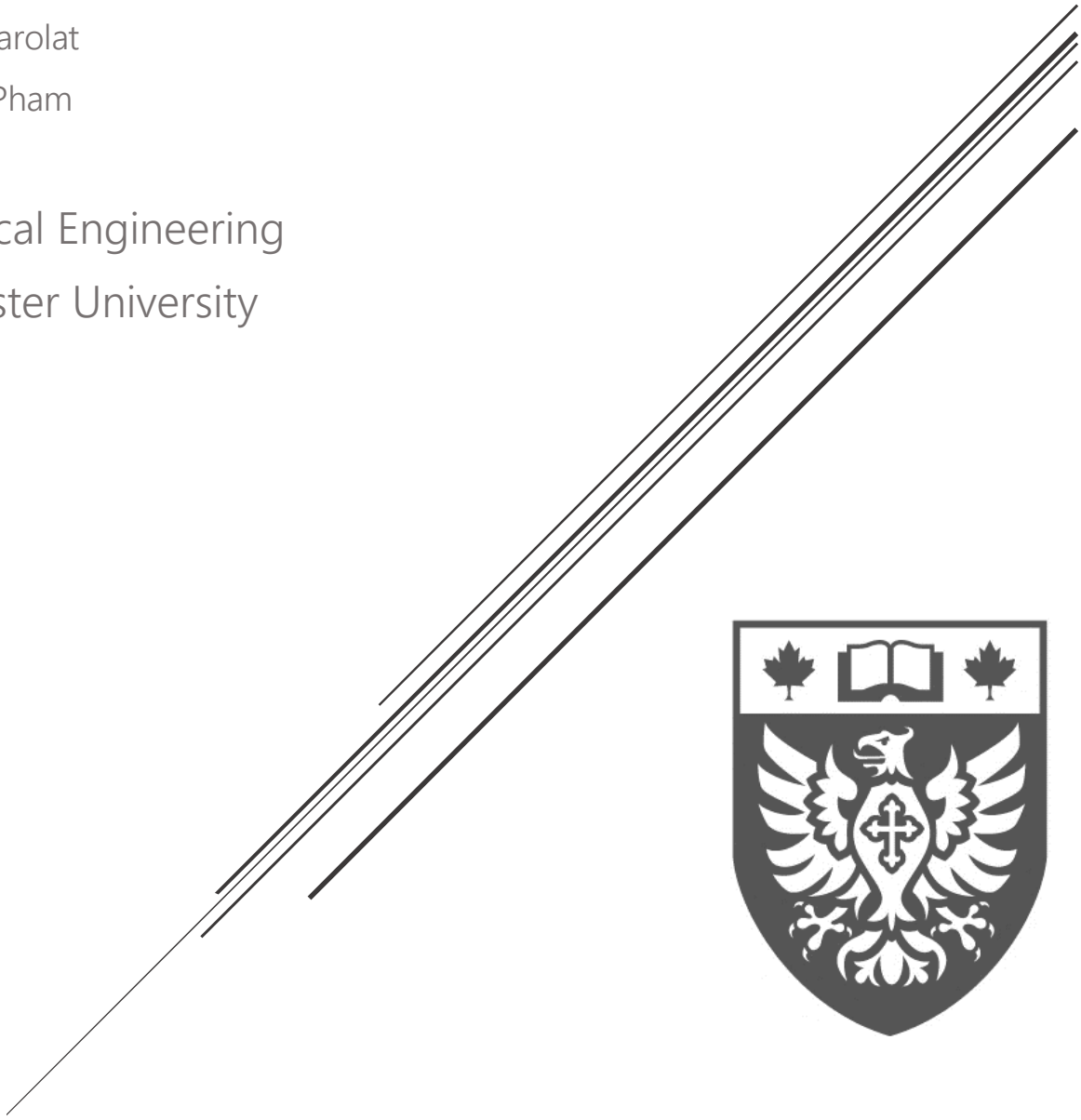
Kieran McKenzie

Steven Karolat

Cynthia Pham

Chemical Engineering

McMaster University



Updated August 27, 2019

Contents

Supplementary Material

Suggested Readings

Gilat, V. Subramaniam: *Numerical Methods for Engineers and Scientists.* Wiley. Third Edition (2014)

- Jacobi Method
  - Chapter 4 → *4.7.1*
- Gauss-Seidel Method
  - Chapter 4 → *4.7.2*
- Matrix Condition Number and Norms
  - Chapter 4 → *4.10.2*

Online Material

Gregg Waterman > The Jacobi Method *(Click to follow hyperlink)*

- Applies the Jacobi Method to an example LSoE, using hand calculations to demonstrate procedure

Gregg Waterman > The Gauss-Seidel Method *(Click to follow hyperlink)*

- Applies the Gauss-Seidel Method to an example LSoE, using hand calculations to demonstrate procedure

Overview

Up to now, we have covered three direct methods for solving LSoE. Let us now look at some iterative methods!

Learning Outcomes for this Module

- Introduce and derive the *Jacobi Method* using elementary algebra operations.
- Adapt the Jacobi method to develop the *Gauss-Seidel* method.
- List and compare the *trade-offs* of iterative methods versus elimination methods.
- Derive the *condition number* and its relation to numerical *stability* and *precision*.
- Analyze the trade-offs between *speed* and *reliability* by exploring algorithm *relaxation*.

Iterative Methods

Iterative methods are a different approach to solving LSoE. Here, you begin with an *initial guess* of the solution $\boldsymbol{x}$ and use it to develop a sequence of further guesses of the solution. Think of iterative methods as an automated form of guess-and-check.

There are three main steps to ANY iterative method (to be continued in future sections of this course):

1. Pick a value as your *initial guess* of the solution.
2. Perform some sort of *algorithmic adjustment* to your guess that hopefully takes you closer to the solution.
3. Terminate once the *termination criterion* has been reached.

Important: Notation

Each element in the solution vector $\boldsymbol{x}^{(k)}$ is denoted as $x_i^{(k)}$. The index i represents the i^{th} element in the vector in the *same* iteration, (*i.e.* $x_1^{(k)}, x_2^{(k)}, x_3^{(k)}$), while k tracks which iteration we are on, starting at $k = 0$ for the initial guess.

There are two iterative algorithms covered in this module: *Jacobi*, and *Gauss-Seidel*. Jacobi is the "base" method, and Gauss-Seidel is a variant of it with only one (possibly significant) difference.

Jacobi Method

Suppose we have a $n \times n$ LSoE and we make an educated selection for our initial guess to the solution to be:

$$\boldsymbol{x}^{(0)} = \left[x_1^{(0)}, \dots, x_n^{(0)}\right]$$

We will use $\boldsymbol{x}^{(0)}$ to individually find each term in:

$$\boldsymbol{x}^{(1)} = \left[x_1^{(1)}, \dots, x_n^{(1)}\right]$$

This is done in a systematic fashion in the Jacobi Method. Beginning with a square system of equations (n equations and n unknowns), cycle through each i^{th} equation and solve for the i^{th} unknown (where $i = 1, \dots, n$):

$$\begin{matrix} i = 1, \dots, n \\ j = 1, \dots, n, j \neq i \end{matrix} \quad \begin{cases} x_1 = & \frac{-1}{a_{1,1}}\left(a_{1,2}x_2 + \dots + a_{1,j}x_j + \dots + a_{1,n}x_n - b_1\right) \\ \vdots & \vdots \\ x_i = & \frac{-1}{a_{i,i}}\left(a_{i,1}x_1 + \dots + a_{i,j}x_j + \dots + a_{i,n}x_n - b_i\right) \\ \vdots & \vdots \\ x_n = & \frac{-1}{a_{n,n}}\left(a_{n,1}x_1 + \dots + a_{n,j}x_j + \dots + a_{n,(n-1)}x_{(n-1)} - b_n\right) \end{cases}$$

Now that the equations are set up, an initial guess at what the solution vector might be, $\boldsymbol{x}^{(0)} = \left[x_1^{(0)}, \dots, x_n^{(0)}\right]$, is made.

Jacobi uses $\boldsymbol{x}^{(0)}$ to calculate $\boldsymbol{x}^{(1)}$ by individually calculating each $x_i^{(1)}$ using the equations above, and the previous iteration, as follows:

$$\begin{matrix} i = 1, \dots, n \\ j = 1, \dots, n \,||\, j \neq i \end{matrix} \quad \begin{cases} x_1^{(1)} = & \frac{-1}{a_{1,1}}\left(a_{1,2}x_2^{(0)} + \dots + x_j^{(0)} + \dots + a_{1,n}x_n^{(0)} - b_1\right) \\ \vdots & \vdots \\ x_i^{(1)} = & \frac{-1}{a_{i,i}}\left(a_{i,1}x_1^{(0)} + \dots + x_j^{(0)} + \dots + a_{i,n}x_n^{(0)} - b_i\right) \\ \vdots & \vdots \\ x_n^{(1)} = & \frac{-1}{a_{n,n}}\left(a_{n,1}x_1^{(0)} + \dots + x_j^{(0)} + \dots + a_{n,(n-1)}x_{(n-1)}^{(0)} - b_n\right) \end{cases}$$

As you can see, *each equation* from above has only *one unknown*, and can be solved for $x_i^{(1)}$ explicitly. This completes the *first iteration* of Jacobi.

Further iterations are calculated in the same way:

$$\begin{matrix} i = 1, \dots, n \\ j = 1, \dots, n \,||\, j \neq i \\ k = 1{:}\, maxiters \end{matrix} \quad \begin{cases} x_1^{(k+1)} = & \frac{-1}{a_{1,1}}\left(a_{1,2}x_2^{(k)} + \dots + x_j^{(k)} + \dots + a_{1,n}x_n^{(k)} - b_1\right) \\ \vdots & \vdots \\ x_i^{(k+1)} = & \frac{-1}{a_{i,i}}\left(a_{i,1}x_1^{(k)} + \dots + x_j^{(k)} + \dots + a_{i,n}x_n^{(k)} - b_i\right) \\ \vdots & \vdots \\ x_n^{(k+1)} = & \frac{-1}{a_{n,n}}\left(a_{n,1}x_1^{(k)} + \dots + x_j^{(k)} + \dots + a_{n,(n-1)}x_{(n-1)}^{(k)} - b_n\right) \end{cases}$$

Where '$maxiters$' is the user specified value for the maximum allowable iterations. Iterations are performed until a stopping condition is met (either $maxiters$ have been performed, or a tolerance is reached.

The hand calculation of the theory is shown in an example below – the algorithm is shown in the blue box.

Ex. (Fausett, Applied Numerical Analysis Using MATLAB, 2008. Page 218.)

Given LSoE:

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 1 & 2 & -1 \\ 1 & -1 & 2 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} -1 \\ 6 \\ -3 \end{bmatrix}$$

Writing the Equations:

$$\begin{aligned} 2x_1 - x_2 + x_3 &= -1 \\ x_1 + 2x_2 - x_3 &= 6 \\ x_1 - x_2 + 2x_3 &= -3 \end{aligned}$$

→ Rearrange each equation to find the $\boldsymbol{x}$ vector:

$$\begin{aligned} x_1 &= 0x_1 + 0.5x_2 - 0.5x_3 - 0.5 \\ x_2 &= -0.5x_1 + 0x_2 + 0.5x_3 + 3 \\ x_3 &= -0.5x_1 + 0.5x_2 + 0x_3 - 1.5 \end{aligned}$$

Inserting our isolated equations into a coefficient matrix:

$$\begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ x_3^{(k+1)} \end{bmatrix} = \begin{bmatrix} 0 & 0.5 & -0.5 \\ -0.5 & 0 & 0.5 \\ -0.5 & 0.5 & 0 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ x_3^{(k)} \end{bmatrix} + \begin{bmatrix} -0.5 \\ 3.0 \\ -1.5 \end{bmatrix}$$

Solving the next iteration using the iteration before it! Start with an initial guess.

If we take our initial guess as $\boldsymbol{x}^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}] = [0, 0, 0]$, then:

$$\begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \begin{bmatrix} 0 & 0.5 & -0.5 \\ -0.5 & 0 & 0.5 \\ -0.5 & 0.5 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -0.5 \\ 3.0 \\ -1.5 \end{bmatrix} = \begin{bmatrix} -0.5 \\ 3.0 \\ -1.5 \end{bmatrix}$$

→ We just solved $\boldsymbol{x}^{(1)}$. Use this to find the second iteration, $\boldsymbol{x}^{(2)}$:

$$\begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \\ x_3^{(2)} \end{bmatrix} = \begin{bmatrix} 0 & 0.5 & -0.5 \\ -0.5 & 0 & 0.5 \\ -0.5 & 0.5 & 0 \end{bmatrix} \begin{bmatrix} -0.5 \\ 3.0 \\ -1.5 \end{bmatrix} + \begin{bmatrix} -0.5 \\ 3.0 \\ -1.5 \end{bmatrix} = \begin{bmatrix} 1.75 \\ 2.50 \\ 0.25 \end{bmatrix}$$

→ Stopping Condition: Continue this process until the Euclidean Norm of the difference between two successive $\boldsymbol{x}^{(k)}$ is less than 0.001.

This system converges after *12 iterations* of the Jacobi method to the vector shown below, based on the chosen stopping condition. Our approximation brought us strikingly close to the real answer!

Jacobi Method

$$\boldsymbol{x}^{(13)} = \begin{bmatrix} 1.0002 \\ 2.0001 \\ -0.9997 \end{bmatrix}$$

True Solution

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

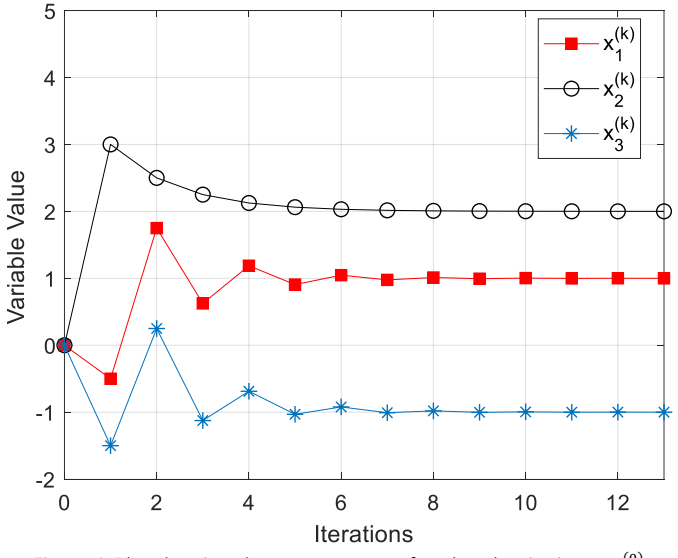


Figure 1: Plot showing the convergence of each x_i beginning at $x_i^{(0)}$.

Primary Calculation: Jacobi Iteration

For each variable $x_i^{(k)}$ $(i = 1, \ldots, n)$, the general equation to determine the next iteration is $x_i^{(k+1)}$ is:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{i,j} x_j^{(k)} \right), \quad a_{i,i} \neq 0$$

Continue iterations until the termination criterion is met. If $a_{i,i} = 0$, use partial pivoting before initiating this method.

Computational Complexity: $\mathcal{O}(in^2)$, where i is the number of iterations required for convergence.

Workshop: Analyzing a **MATLAB** Code for Jacobi Iteration

A MATLAB code for Jacobi iteration is provided. Analyze each step of the code!

```
x = initial_guess
for k = 1:max_iters
    xstore = zeros(1,n);
    for i = 1:n
        summ = 0;
        div = A(i,i)
        for j = 1:n
            if j ~= i
                summ = summ + A(i,j)*x(j)
            end
        end
        xstore(i) = 1/div * (b(i) - summ);
    end
    x = xstore;
end
```

What does this loop do?

Why do we want to create a storage variable for x?

Why are we setting `summ = 0`?

Why do we want to skip the i^{th} row?

What are we doing here?

What is this expression for?

Gauss-Seidel Method

Gauss-Seidel (GS) is the *same* as the Jacobi method, with *one exception:* When we find $x_i^{(k+1)}$ we *immediately* use it to solve for $x_{i+1}^{(k+1)}$. Instead of trying to figure out ALL of $\boldsymbol{x}^{(k+1)}$ before using those values, we *use the most recent* elements as soon as they become available.

Using our previous Jacobi Method example:

Given LSoE:	*Solving for $\boldsymbol{x} = \{x_1, x_2, x_3\}$, we get:*
$2x_1 - x_2 + x_3 = -1$ $x_1 + 2x_2 - x_3 = 6$ $x_1 - x_2 + 2x_3 = -3$	$x_1 = 0x_1 + 0.5x_2 - 0.5x_3 - 0.5$ $x_2 = -0.5x_1 + 0x_2 + 0.5x_3 + 3$ $x_3 = -0.5x_1 + 0.5x_2 + 0x_3 - 1.5$

If we take our initial guess as $\left[x_1^{(0)}, x_2^{(0)}, x_3^{(0)}\right] = [0, 0, 0]$ then:

$$x_1^{(1)} = 0x_1^{(0)} + 0.5x_2^{(0)} - 0.5x_3^{(0)} - 0.5 = \ldots \quad \text{(Same as Jacobi)}$$
$$x_2^{(1)} = -0.5x_1^{(\mathbf{1})} + 0x_2^{(0)} + 0.5x_3^{(0)} + 3 = \ldots \quad \text{(Updated using } x_1^{(1)} \text{ now)}$$
$$x_3^{(1)} = -0.5x_1^{(\mathbf{1})} + 0.5x_2^{(\mathbf{1})} + 0x_3^{(0)} - 1.5 = \ldots \quad \text{(Updated with } x_1^{(1)} \text{ and } x_2^{(1)}\text{)}$$

The actual math for this method is the exact same! For this system, the Gauss-Seidel method converges after only 9 iterations to the same accuracy as our Jacobi example:

Gauss-Seidel Method

$$\boldsymbol{x}^{(10)} = \begin{bmatrix} 1.0001 \\ 1.9999 \\ -1.0001 \end{bmatrix}$$

True Solution

$$\boldsymbol{x} = \begin{bmatrix} 1 \\ 2 \\ -1 \end{bmatrix}$$

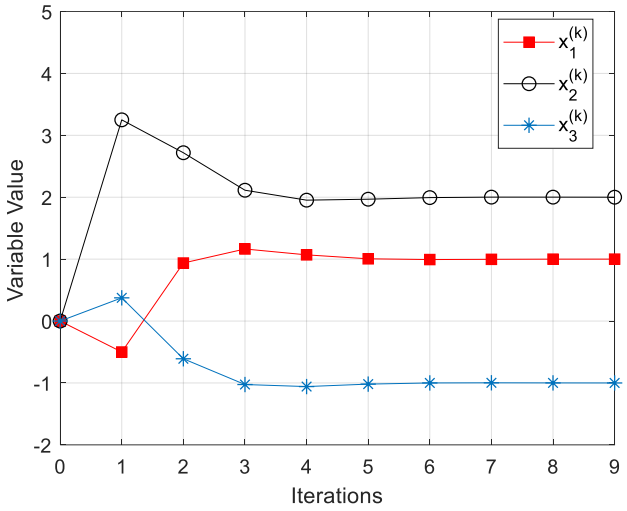


Figure 2: Plot showing the convergence of each x_i beginning at $x_i^{(0)}$

Primary Calculation: Gauss-Seidel Iteration

For each variable $x_i^{(k)} (i = 1, \ldots, n)$ the general equation for $x_i^{(k+1)}$ is:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right), \quad a_{i,i} \neq 0$$

The Gauss-Seidel variant typically converges 2 × faster than the Jacobi Method! *Pay special attention* to the iterations used to compute the summations and summation bounds themselves.

Computational Complexity: $\mathcal{O}(in^2)$ where i is the number of iterations required for convergence. Note that this is the same complexity as Jacobi, and thus is only faster if less iterations are required.

Workshop: Making a **MATLAB** Code for Gauss-Seidel

The code for Gauss-Seidel can be made by making a few small tweaks to the Jacobi code! Let's do that together now!

Hint: There will be changes to the initial guess n times in each iteration. Since we are updating the values as we go, we should find somewhere to store the non-updated values ☺

Jacobi

```
x = initialGuess

for k = 1:maxIterations

      xStore = zeros(1,n);

      for i = 1:n

            summ = 0;
            div = A(i,i)

            for j = 1:n
                  if j ~= i
                        summ = summ + A(i,j)*x(j)
                  end
            end

            xstore(i) = (1/div)*(b(i)-summ);

      end

      x = xstore;

end
```

Gauss-Seidel

```
x = initialGuess

for k = 1:maxIterations

      xStore = [          ]

      for i = 1:n

            summ = 0;
            div = A(i,i)

            for j = 1:n
                  if j ~= i
                        summ = summ + A(i,j)*x(j)
                  end
            end

            [          ] = (1/div)*(b(i)-summ);

      end

      x = xstore;

end
```

Picking an Initial Guess

There is *no guaranteed method* to picking an initial guess. If a system converges for one starting value, it will not necessarily converge for all. It is critical to provide a *good initial guess* based on *engineering intuition*. Some systems will converge for all initial guesses – with the only difference being the number of iterations it takes to get to the final solution from the initial guess (*number of FLOPs*).

Termination Criteria

Knowing when to stop is critical when working with iteration methods (and in life). We stop when we see very minimal changes in the approximations for $\boldsymbol{x}$ - if more iterations hardly result in any change, we can assume that our solution is close to the true solution. *It is your decision as an engineer* to determine what level of accuracy is appropriate.

Iterative Methods: Termination Criteria

Here are two commonly used methods:

When the relative absolute error between element iterations is less than a specified error tolerance (ε).

$$\left|\frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k)}}\right| < \varepsilon_i^{rel}, \forall_i$$

When the relative magnitude (Euclidean Norm) of $\Delta\boldsymbol{x} \triangleq \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}$ (the difference between iterations of $\boldsymbol{x}$) is less than the specified error tolerance (ε).

$$\frac{\left\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\right\|}{\left\|\boldsymbol{x}^{(k)}\right\|} < \varepsilon^{rel}, \forall_i$$

Set the value of ε to be small - typical values are in the 10^{-6} range. *Calculations of norms will be discussed shortly.*

BE CAREFUL: Convergence is not always guaranteed for iterative methods. Restrict the number of max allowable iterations to avoid infinitely running scripts!

Note: "Ctrl+C" can be used to stop an infinitely running MATLAB script.

Workshop: Adding Termination Criterion

Below, we have the previous code for Jacobi iteration. Let's apply our termination criterion to stop this code if we've found our desired solution ☺

```
x = initialGuess
tolerance = 10^-6

for m = 1:maxIterations

     xStore = zeros(1,n);

     for i = 1:n

          summ = 0;
          div = A(i,i)

          for j = 1:n

               if j ~= i
                    summ = summ + A(i,j)*x(j)
               end

          end
          xstore(i) = (1/div)*(b(i)-summ);

     end

     x = xstore;

end
```

Here's some space to write your termination criterion! When you're done, put an arrow to where you think it should go. You shouldn't need all the space.

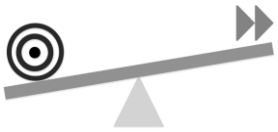
Improved Reliability: Iterative Relaxation / Smoothing

Relaxation (AKA 'smoothing') can make our iterative methods less "aggressive". This technique also has uses outside of engineering, such as in business forecasting.

Speed vs. Reliability

With iterative methods, there is always a trade-off between speed and reliability. The *relaxation factor* (ω) is how we choose which is more important to us.

Models that are too fast/aggressive may overshoot and not converge, while others - although more reliable - may take too long to find the solution to be practical.



Relaxation

$x_i^{(k+1)}$ is updated after it is determined by a specified relaxation factor to keep a *weighted average* between previous and new estimates of x_i (*i.e.* We can choose *how much emphasis* we put on our new estimate compared to our old one).

$$x_i^{(k+1)} = \boldsymbol{\omega} x_i^{(k+1)} + (\mathbf{1} - \boldsymbol{\omega}) x_i^{(k)} \qquad 0 < \omega \leq 2$$

In the case of normal Jacobi or GS, $\omega = 1$, meaning we have no regard for previous values once we find $x_i^{(k+1)}$.

If $0 < \omega < 1$: *Under-relaxation*, means a more reliable convergence (WHY?).
If $1 < \omega \leq 2$: *Over-relaxation*, means it *may* converge faster (WHY?).
If $2 < \omega$: The system *will always diverge*.

Increasing ω narrows the acceptable range of workable guesses:

- Under-relaxation: your initial guess can be farther away from the solution but still converge over time.
- Over-relaxation will only work if the guess is nearer to the actual solution

In the end, it's up to YOU as an *engineer* to make the decision to decide what the value of ω is.

The Below plots use varying values of ω with the Jacobi Method for the prior problem. The true solutions should be $\boldsymbol{x} = [1\,, 2\,,\ -1\,]$.

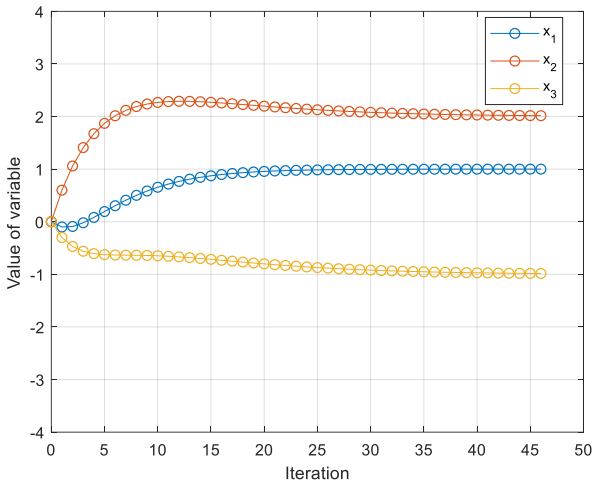


Figure 3: Using a ω of 0.2 – takes 46 iterations to converge

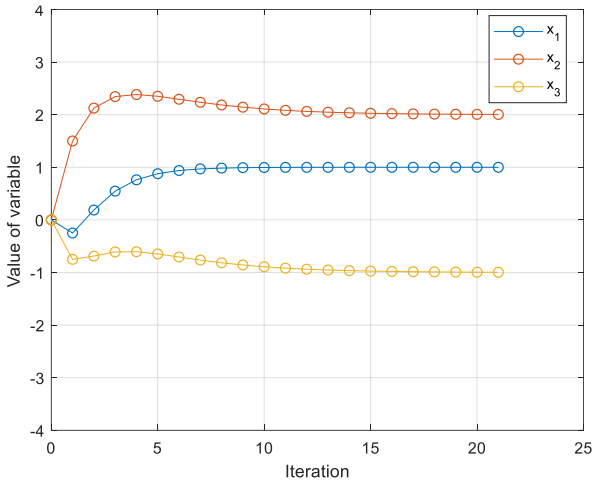


Figure 4: Using a ω of 0.5 – takes 21 iterations to converge

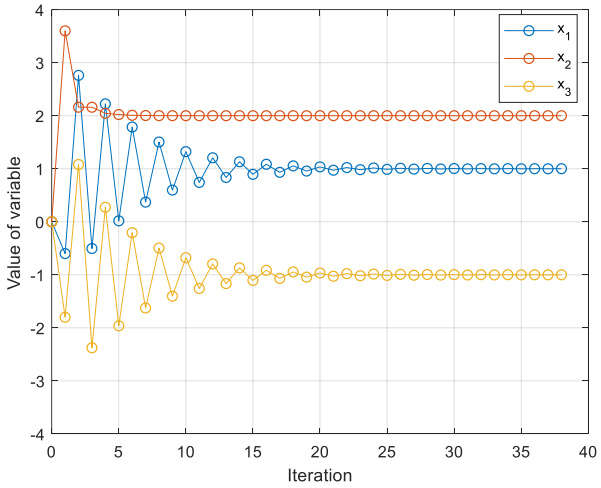


Figure 5: Using a ω of 1.2 - takes 38 iterations to converge

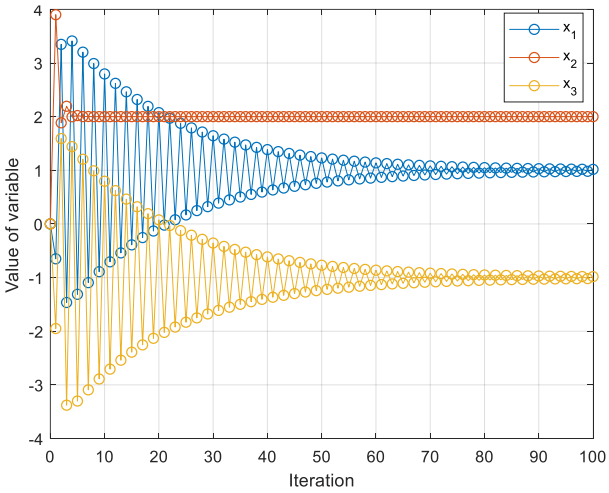


Figure 6: Using a ω of 1.3 – no convergence in 100 iterations.

Diagonal Dominance

Convergence is not always guaranteed for iterative methods; however, a diagonally dominant matrix can *guarantee convergence of GS or Jacobi*. (It also allows us to confidently apply GE or LU decomposition without pivoting! Craziness!)

Diagonal Dominance

A matrix is *diagonally dominant* if the two conditions below are met:

1. The magnitude of the diagonal element in *each row* is *at least as big* as the sum of all others in its row:

$$|a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i,j}| \ \forall i$$

2. In at least one row, the diagonal element must be *greater* than the sum of its row:

$$|a_{i,i}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i,j}| \ \forall i$$

If your original A doesn't meet the two criteria above, then it isn't diagonally dominant – *use pivoting* to rearrange. Even if you can't make A diagonally dominant, there's still a chance it will converge.

Workshop: Diagonal Dominance

The following coefficient matrices were each retrieved from an LSoE in the form $\boldsymbol{Ax} = \boldsymbol{b}$. Are these coefficient matrices diagonally dominant?

If not, can you use partial pivoting to make the coefficient matrices diagonally dominant? If yes, show the new matrix and $\boldsymbol{b}$ vector after pivoting is performed.

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 6 & 10 & 4 \\ 8 & 7 & 20 \end{bmatrix} \quad \boldsymbol{b}_1 = \begin{bmatrix} 4 \\ 10 \\ -2 \end{bmatrix} \qquad B = \begin{bmatrix} 10 & 8 & 2 \\ -2 & 5 & 3 \\ 6 & 14 & 20 \end{bmatrix} \quad \boldsymbol{b}_2 = \begin{bmatrix} 5 \\ -7 \\ 8 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 2 & 1 \\ 4 & 1 & 2 \\ 5 & 3 & 8 \end{bmatrix} \quad \boldsymbol{b}_3 = \begin{bmatrix} 5 \\ 0 \\ 8 \end{bmatrix}$$

Matrix Condition Numbers and Norms

Numerical solutions are slightly inaccurate in many cases - numerical precision (round-off) error can occur, and our predefined tolerance also affects how accurate our solution is.

Luckily, we can estimate the magnitude of error in a solution by using the matrix's condition number. To compute the condition number, we must use norms.

Norms

A norm is a societal que that is instilled in a certain culture, such as liking an Instagram post, using an Apple© product in Starbucks™, or other not-as-cynical things.

You likely have learned that the norm of a vector was equal to the 'length' of that vector, right? That's true! At least, that is the definition for the *Euclidean Norm*. There are other types of norms – types for both vectors AND matrices.

The Warm Norm Restaurant

	Name	Description	Ingredients (Equation)
Vector Norms	1-Norm ($\boldsymbol{L_1}$)	This norm gives you the sum of *all absolute values* of elements in a given vector of your choice.	$L_1 = \{ \sum \lvert v_i \rvert \}^{\frac{1}{1}}$
	Euclidean 2-Norm ($\boldsymbol{L_2}$)	The classic loved by all first years. This dish features the *magnitude* of a vector, calculated in the standard way.	$L_2 = \{ \sum \lvert v_i \rvert^2 \}^{\frac{1}{2}}$
	Infinity Norm ($\boldsymbol{L_\infty}$)	The most expensive item on this menu. Why? This norm gives you the *maximum absolute value* that is contained within a vector.	$L_\infty = \{ \sum \lvert v_i \rvert^\infty \}^{\frac{1}{\infty}}$ $= \max(\lvert \boldsymbol{v} \rvert)$
Note: In general, the equation of a vector norm is $L_p = \{ \sum \lvert v_i \rvert^p \}^{\frac{1}{p}}$ for a given vector, $\boldsymbol{v}$.			
Matrix Norms	1-Norm [$\boldsymbol{A_1}$]	This norm will output the *maximum column sum* of a matrix. Tasty!	$A_\infty = \max_{1 \le j \le n} \lvert a_{i,j} \rvert$
	Euclidean Norm [$\boldsymbol{A_e}$] *(NOT same as [$\boldsymbol{A_2}$] for matrices)	The ultimate. This norm squares each element in the matrix, adds those squares, then takes the square root. Dig in.	$A_e = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2 \right)^{1/2}$
	Infinity Norm [$\boldsymbol{A_\infty}$]	This norm will output the *maximum* row sum of a matrix. Also tasty!	$A_\infty = \max_{1 \le i \le n} \lvert a_{i,j} \rvert$

Norms are ways to measure and compare the magnitude of vectors / matrices. As you can see, there are many ways of doing this! These norms can be used to find the condition number.

Condition Number

The *condition number* of a coefficient matrix is used to determine the degree of ill-conditioning of a set of equations. An ill-conditioned system is a system with a *high condition number*. In ill-conditioned systems, small changes in the system's coefficients will result in large changes in the calculated solution – very unfavourable. This may occur due to:

- Nearly dependent rows (very minor numerical differences can cause *rounding errors*).
- Combining equations of vastly different *scales* (such as very small concentrations in large units combined with very high temperatures).

In both cases, that computer's ability to remember all *significant digits* is limited to *digital memory*, and so a concentration like $\mathbf{0.0000001}$ mol/m^3 may be rounded to $\mathbf{0}$ mol/m^3 (not OK if we are investigating mercury, toxic species, or active pharmaceutical ingredients!).

Condition Number

The condition number of a matrix A is defined as:

$$cond(A) = ||A|| \times ||A^{-1}|| = 10^s$$

Where $||A||$ is any chosen norm of the matrix, and s represents the digits of accuracy that *may* be lost in the solution. Different norm types will output different condition numbers but will still give roughly the same s.

- If $cond(A) = 10^s \approx 1 \Rightarrow s \approx log(1) = 0$ (roughly zero digits will be lost).
  NOTE: $cond(I) \equiv 1$
- If $cond(A) = 10^s \gg 1 \Rightarrow s \approx log(\gg 1) \gg 0$ then the system is ill-conditioned. We may not be able to trust the calculated solution.

Illustrated below is an example of an ill-conditioned system and the large variance in the calculated solutions for only a small change in the coefficient matrix:

Example: Ill-Conditioned Systems and Variance between Solutions

(Retrieved from Gilat and Subramaniam, Numerical Methods for Engineers and Scientists, 2014. Page 153)

$$\begin{bmatrix} 6 & -2 \\ 11.5 & -3.85 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 17 \end{bmatrix}$$

Solutions:

$$x_1 = 45$$
$$x_2 = 130$$

$$\begin{bmatrix} 6 & -2 \\ 11.5 & -3.84 \end{bmatrix} = \begin{bmatrix} 10 \\ 17 \end{bmatrix}$$

Solutions:

$$x_1 = 110$$
$$x_2 = 325$$

Question: How did we know it'd be an ill-conditioned system? Did they randomly find a system with a high condition number?
Answer: Nope! There is way to predict a high condition number! Look closely - the second row of this coefficient matrix is *almost* 2× the first row. The row vectors of this matrix are *approaching linear dependence*.

The closer that the row vectors of a matrix are to being linearly dependent, *the higher its condition number* will be. This idea is proved in the following theorem:

Theorem: Approaching Linear Dependence

If a matrix $A_{n\times n}$ is linearly dependent, a zero row can be formed and $det(A) = 0$; the matrix will not possess an inverse or have a unique solution vector. However, as a matrix only *approaches* linear dependence, $det(A) \to 0$, and the matrix remains invertible.

Recall the classic equation of solving for A^{-1} :

$$A^{-1} = \frac{1}{det(A)} \times adj(A)$$

If $det(A) \to 0$, then the elements of $A^{-1} \to \infty$. Therefore, $\left||A^{-1}|\right| \to \infty$.

$$cond(A) = 10^S = \left||A|\right| \times \left||A^{-1}|\right| \qquad \begin{array}{c} if\ \left||A^{-1}|\right| \to \infty, \\ Then\ cond(A) \to \infty\ and\ s \to \infty \end{array}$$

The digits of accuracy that may be lost in the solution increases toward infinity!

Conclusion and Summary

You've made it to the end of Chapter 1! In this module we have covered:

- Two iterative methods of solving LSoE - Jacobi, and Gauss-Seidel.
- Termination criterion was used using norms as a stopping condition for our iterations
    - There are also other ways!
- Diagonal dominance guarantees convergence of iterative methods.
- The condition number of a matrix can be used to determine if we can trust our calculated solution.
    - Applies for both direct and iterative methods!

By now, you have learned everything you require to solve most LSoE in your future! You have gained practice translating your knowledge into programming code - a skill that will make your life much easier for times to come. In life, it will be your job to recognize when/where to use the ideas you've learned here in real life applications. I wish you good luck.

In the spirit of the Warm Norm Restaurant: https://www.youtube.com/watch?v=vE5gBfCa2Co

Figure 7: Happy Sun[1]

Next up: Non-Linear Algebraic Equations

[1] *https://www.flaticon.com/free-icon/happy-sun_76532*

Appendix A: Jacobi Method Flowchart