CHEMICAL ENGINEERING 2E04

Chapter 1 – Linear Algebraic Equations

Module 1B: Direct Methods to Solve Linear Systems

Dr. Jake Nease

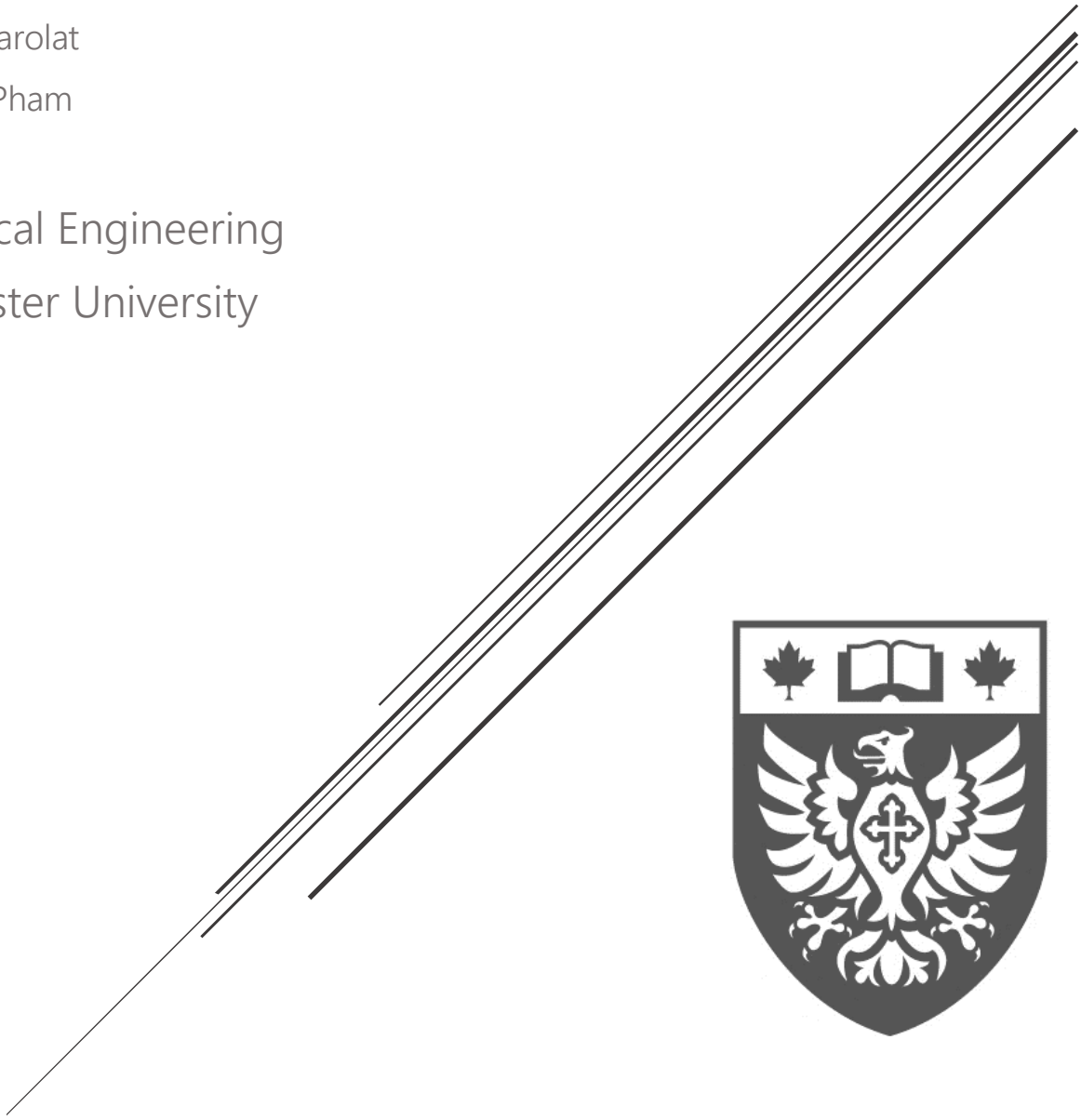
Kieran McKenzie

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)

- Gauss Elimination Method
  - Chapter 4 → *4.2*
- Gauss-Jordan Elimination Method
  - Chapter 4 → *4.4*
- LU Decomposition Method (Gaussian Elimination Method)
  - Chapter 4 → *4.5.1*
- Partial Pivoting with Gauss Elimination and LU Decomposition
  - Chapter 4 → *4.3, 4.5.3*

Introduction and Perspective

In the first Module of this chapter, we looked into defining *linear* systems. Now that we know how to create the LSoE, let's first take a look at the *direct methods* used to solve square LSoE.

Learning Outcomes for this Module

- Introduce *computational complexity* and *rounding errors* in terms of solving LSoE.
- Recognize and describe the two steps of *Gauss Elimination*: Forward Elimination and Backward Substitution.
- Modify Gauss Elimination to develop *Gauss-Jordan Elimination*.
- Derive *LU Decomposition* using multiplication rules for matrices.
- Illustrate how the addition of *partial pivoting* can improve the robustness of our direct methods.

Preface: Computational Complexity and Rounding Errors (Abridged)

Before we begin our journey into solving LSoE using numerical methods, we have two things to consider: computational complexity and rounding errors.

Consideration for Computation Complexity

Measuring Computational Complexity

Computational complexity is measured in *FLOPs* (floating-point operations) – one mathematical operation carried out on a floating-point value in a computer, whether it be a single $+$, $-$, $\times$, or $\div$.

The fewer the FLOPs, the fewer the number of calculations required / the quicker the computer can get you the answer. A common performance metric for computers is *FLOPs/s* (FLOPs per second).

When describing the computational complexity of a method, we normally refer to the *order* of the method. Let's imagine we have an arbitrary numerical method that required the following number of FLOPs, where n is the number of equations:

	Step 1	Step 2	Overall Method
Number of FLOPs	$\frac{2n^3}{3} + \mathcal{O}(n^2)$	$\frac{n^2}{2} + \mathcal{O}(n)$	$\frac{2n^3}{3} + \mathcal{O}(n^2)$

For this method, we would have an order of n^3 *overall!* Therefore, as n (number of equations / variables) grows, the computational requirements to solve the system for $\boldsymbol{x}$ grows at a cubic rate. Let's make this into a definition we can follow for other methods:

- The first term in the expression above is called the *dominant order* (in this case, the n^3).
- All other terms are collapsed using symbol $\mathcal{O}$ *("order")* because all coefficient terms are small in comparison to the rapidly scaling value of $\frac{2}{3}n^3$. Thus, the $\mathcal{O}(n^2)$ term refers to all FLOPs that occur n^2 times or less.

You may ask: how did we get the expression for our overall method?

- Both Step 1 and Step 2 are required to complete our arbitrary method. The *overall* computational complexity is the same as the *rate limiting step* – the step which has the *highest order* (in this case, step 1).

Estimating Computational Complexity from Algorithm Structure

There is a way to estimate the computational complexity of algorithms by looking at their structure. We will investigate this idea below.

Answer the questions prompted for each of the following arbitrary matrices and accompanying algorithms written in MATLAB:

Matrix	Algorithm	Questions

$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	`A = zeros(3,3);`
	`n = length(A);`
	
	`FLOPS = 0;`
	`for i = 1:n`
	
	`  A(i,1) = A(i,1) + 1;`
	`  FLOPS = FLOPS + 1`
	
	`end`
		A) What will the value of FLOPS be when the algorithm is complete?
		
		B) Express FLOPS in terms of the size of the matrix, n.

	`A = zeros(3,3);`
	`n = length(A);`
	
	`FLOPS = 0;`
	`for i = 1:n  % Cycle through Rows`
	`  for j = 1:n % Cycle through Columns`
	
	`  A(i,j) = A(i,j) + 1;`
	`  FLOPS = FLOPS + 1;`
	
	`  end`
	`end`
		A) What will the value of FLOPS be when the algorithm is complete?
		
		B) Express FLOPS in terms of the size of the matrix, n.

$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	`A = zeros(2,3);`
	`[m, n] = size(A);`
	
	`FLOPS = 0;`
	`for i = 1:m  % Cycle through Rows`
	`  for j = 1:n  % Cycle through Columns`
	
	`  A(i,j) = A(i,j) + 1;`
	`  FLOPS = FLOPS + 1;`
	
	`  end`
	`end`
		A) What will the value of FLOPS be when the algorithm is complete?
		
		B) Express FLOPS in terms of the dimensions of the matrix, m and n.

1) Suggest a general rule for how the value of FLOPS depends on the number and range of the for-loops in an algorithm:

2) In a LSoE, operations must be done on both the coefficient matrix, $\boldsymbol{A}$, as well as the $\boldsymbol{b}$ vector. The linear system given below is composed of $\mathbf{1000}$ equations with $\mathbf{1000}$ unknowns.

How might your answer from 1) change for the following algorithm, which is performed on a LSoE instead of just a matrix alone? Does the general relationship between FLOPS and for-loops differ?

Algorithm	Questions

`A = zeros(1000,1000);`
`b = zeros(1000,1);`
`n = length(A);`
 	
`FLOPS = 0;`
`for i = 1:n  % Cycle through Rows`
`  for j = 1:n  % Cycle through Columns`
 	
`  A(i,j) = A(i,j) + 1;`
`  FLOPS = FLOPS + 1;`
 	
`  end`
 	
`  b(i) = b(i) + 1;`
`  FLOPS = FLOPS + 1;`
 	
`end`
	A) What the value of FLOPS be when the algorithm is complete?
	Hint: Your findings from the first two workshop examples above can help you if you're stuck!
	
	B) Express FLOPS in terms of the size of the matrix, $\boldsymbol{n}$, after solving for the term in the exponent!

3) Reason why your findings from B) make sense:	

Consideration for Round-Off Errors

Computers do not have the ability to express numbers as fractions. For example, $^2/_3 = 0.\bar{6} \approx 0.66667$ in computer world! We know the number is "two thirds" or $0.\bar{6}$, but computers remember information in *discrete quantities* known as *bits*. There are only a distinct number of bits that can store information (64 on most modern computers, as many of you know). This means that ALL computers, regardless of how powerful, *round off* after several digits, losing all numerical information thereafter.

Consider the following LSoE:

$$0.0003x_1 + 3.0000x_2 = 2.0001$$
$$1.0000x_1 + 1.0000x_2 = 1.0000$$

Solving by elimination readily gives:

$$x_2 = \frac{2}{3} \qquad x_1 = \frac{2.001 - 3x_2}{0.0003}$$

Computers can't use $^2/_3$. Instead, it is rounded to a specified number of significant figures depending on the *memory* dedicated to that variable (discussed in tutorial). See the chart below to see the effects this can have on the final solution for our example system:

Significant Figures	Round $\boldsymbol{x_2}$ to…	Resulting $\boldsymbol{x_1}$ solution
3	0.667	$\mathbf{-3}$
4	0.6667	$\mathbf{0}$
5	0.66667	0.3
6	0.666667	0.33

Clearly, this is not a good situation. What if you are trying to solve for a variable (x_1) that is the number of $\boldsymbol{mg}$ of an active pharmaceutical ingredient for a painkiller? Or if you are solving for the weight fraction of a particular gasoline component in a blend? The result in the table above are far too different to be comfortable. We will talk a little later in this course about *scaling and conditioning* with this in mind.

Gauss Elimination

Gauss Elimination (GE) is a direct method where row operations are performed to *eliminate* equations and make the LSoE easier to solve!

Example: Solving By Elimination

Let's remind ourselves of what 'elimination' is, using both standard equation format as well as matrix form:

Elimination in Equation Form	Elimination in Matrix Form, $\boldsymbol{Ax = b}$
Example LSoE:	Arranging the LSoE into matrix form:
Equation 1: $2x + y = 2$ Equation 2: $x + 2y = 5$	$\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$
Eliminating x from Equation 2: $(Equation\ 2) - 1/2 \times (Equation\ 1)$	Eliminating x from Equation 2: $(Row\ 2) - 1/2 \times (Row\ 1)$
$x + 2y = 5$ $-\ 1/2\ (2x + y = 2)$ $0x + 1.5y = 4$ $y = 8/3$	$\begin{bmatrix} 2 & 1 \\ 0 & 1.5 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ $1.5y = 4$ $y = 8/3$
We've solved for y using Equation 2. Use this new knowledge to solve for x using Equation 1!	We've solved for y using Equation 2. Use this new knowledge to solve for x using Equation 1!
$2x + y = 2$ $x = \frac{(2 - 8/3)}{2}$ $x = -\frac{1}{3}$	$2x + y = 2$ $x = \frac{(2 - 8/3)}{2}$ $x = -\frac{1}{3}$

You've all done this before, but have you thought about how to apply it to large systems? There's an underlying pattern to GE that can be split into two steps: *Forward Elimination* and *Backward Substitution*. Computers will always use matrices to perform elimination.

Two Steps of Gauss Elimination

Gauss Elimination can be broken down into two consecutive steps:

1. Forward Elimination: Manipulate the LSoE in $\boldsymbol{Ax = b}$ form to end with an upper triangular coefficient matrix, A.

2. Backward Substitution: Using the upper triangular coefficient matrix configuration, efficiently solve for $\boldsymbol{x}$. A is in upper triangular form, so we can easily solve the LSoE by starting at the n^{th} equation and working upwards using previously calculated values of x_i.

Let's go through what this means in more detail. ☺

Step 1 - Forward Elimination

Goal: Achieve an upper triangular coefficient matrix, $\boldsymbol{A}$, preparing the LSoE to be solved with Backward Substitution.

Approach: Cycle through each *pivot element* (the term on the diagonal) and perform row operations to *all rows beneath* the current pivot row (including the RHS vector $\boldsymbol{b}$) to make all $a_{i,j}$ *below* the pivot element equal to *zero*.

For example, starting with a 3×3 LSoE:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Result:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

The prime (') denotes that an element has changed through matrix operations. The number of primes signify the number of operations done to that coefficient. (i.e. the number of times it has changed).

Workshop: Forward Elimination

Give it a try on your own: perform Forward Elimination on our crude oil example! Try to write out all the steps and operations you perform.

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 2 \\ 3 & 2 & 1 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 19 \\ 25 \\ 31 \end{bmatrix}$$

Now that we practiced with some numbers, let's try to generalize this for *any* 3×3 matrix! Don't fret, this is just the same as before!

Initial Coefficient Matrix $\boldsymbol{A}$:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

Step 1. Turning $\boldsymbol{a_{2,1}}$ into zero

$$Row\ 2 = (Row\ 2) - (Row\ 1) \times \frac{a_{2,1}}{a_{1,1}}$$

Our current pivot element is $a_{1,1}$.

This eliminates the $a_{2,1}$ term, leaving a zero. This also affects the rest of the terms in the row, as well as your $\boldsymbol{b}$ vector. They *cannot be forgotten* (What you do to one side of the equation, the same must be done to the other)!

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ \underbrace{a_{2,1} - a_{1,1}\left(\frac{a_{2,1}}{a_{1,1}}\right)}_{0} & \square & \square \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

Giving:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

Effect on $\boldsymbol{b}$:

$$\begin{bmatrix} b_1 \\ \square \\ b_3 \end{bmatrix}$$

Step 2. Turning $\boldsymbol{a_{3,1}}$ into zero

$$Row\ 3 = (Row\ 3) - (Row\ 1) \times \frac{a_{3,1}}{a_{1,1}}$$

The pivot element is still $a_{1,1}$.

Do you see a pattern yet? If not, you will!

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ \underbrace{a_{3,1} - \left(\frac{a_{3,1}}{a_{1,1}}\right)a_{1,1}}_{0} & \underbrace{a_{3,2} - a_{1,2}\left(\frac{a_{3,1}}{a_{1,1}}\right)}_{a'_{3,2}} & \underbrace{a_{3,3} - a_{1,3}\left(\frac{a_{3,1}}{a_{1,1}}\right)}_{a'_{3,3}} \end{bmatrix}$$

Giving:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & a'_{3,2} & a'_{3,3} \end{bmatrix}$$

Effect on $\boldsymbol{b}$:

$$\begin{bmatrix} b_1 \\ b'_2 \\ \underbrace{b_3 - \left(\frac{a_{3,1}}{a_{1,1}}\right)b_1}_{b'_3} \end{bmatrix}$$

Step 3. Turning $\boldsymbol{a_{3,2}}$ into zero

Do the above procedure again but working now with a 2×2 'sub-matrix' instead of a 3×3. We call this a *sub-block*. All we do is move the pivot element one entry down the diagonal to $a'_{2,2}$:

$$Row\ 3 = (Row\ 3) - (Row\ 2) \times \frac{a'_{3,2}}{a'_{2,2}}$$

Remember, our $a_{i,j}$ and b_i *are constants*, so even though it looks chaotic, each term is only a number.

Now our LSoE is ready to be solved via Backward Substitution!

For larger matrices, the above procedure is performed until there is *only one element* in your n^{th} row (i.e. only one term in your matrix's bottom row). Notice the pattern: $a_{i,i}$ is the pivot element for the i^{th} sub-block.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & \boldsymbol{a'_{2,2}} & \boldsymbol{a'_{2,3}} \\ 0 & \boldsymbol{a'_{3,2}} & \boldsymbol{a'_{3,3}} \end{bmatrix}$$

Zooming in on the sub-block:

$$\begin{bmatrix} a_{2,2}' & a_{2,3}' \\ \underbrace{a'_{3,2} - a'_{2,2}\left(\frac{a'_{3,2}}{a'_{2,2}}\right)}_{0} & \underbrace{a'_{3,3} - a_{2,3}\left(\frac{a'_{3,2}}{a'_{2,2}}\right)}_{a''_{3,3}} \end{bmatrix}$$

Giving:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix}$$

Effect on $\boldsymbol{b}$:

$$\begin{bmatrix} b_1 \\ b'_2 \\ \underbrace{b'_3 - b'_2\left(\frac{a'_{3,2}}{a'_{2,2}}\right)}_{b''_3} \end{bmatrix}$$

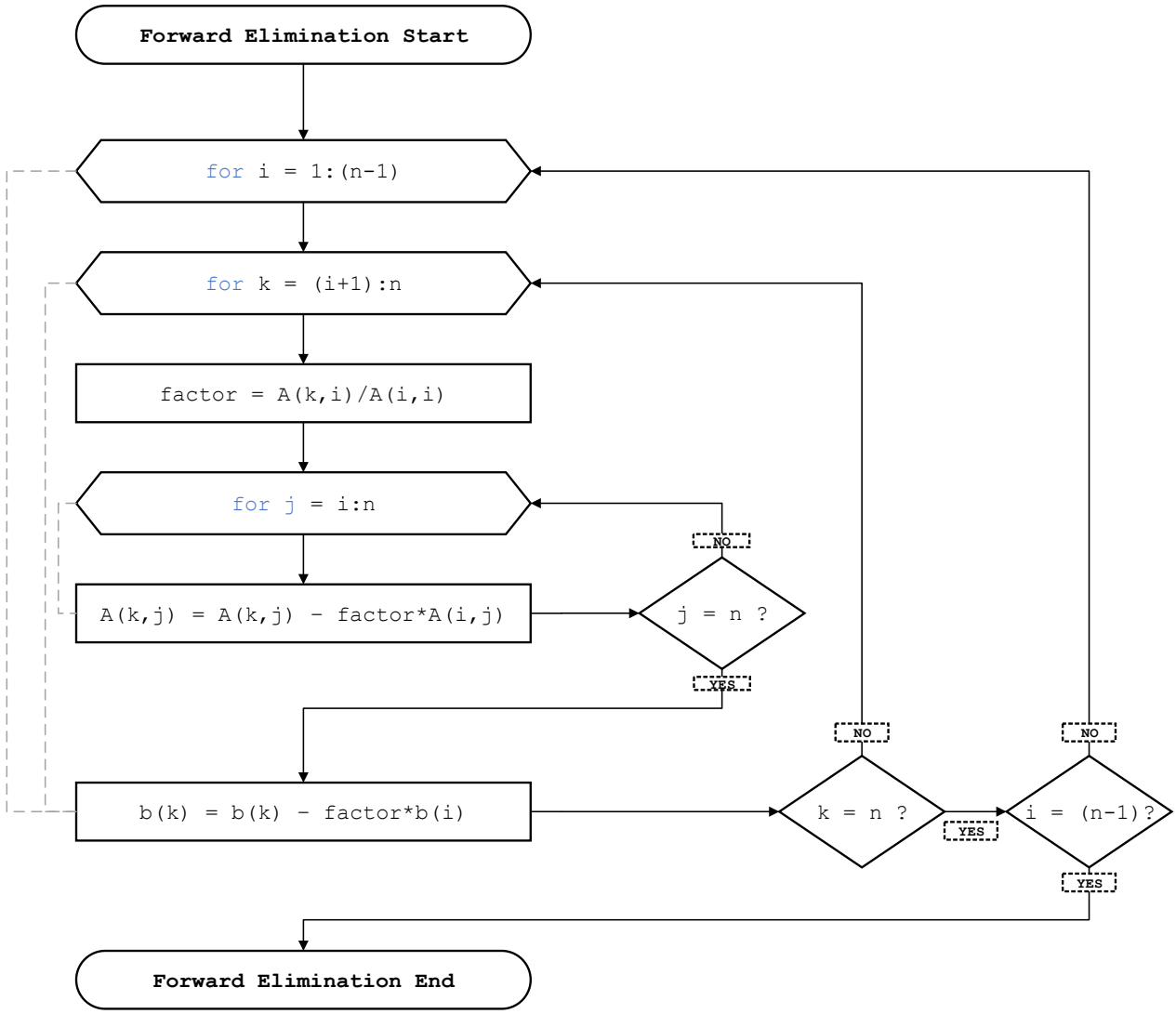
All of which comes together to create:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix} \text{ and } \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

This process gets *repetitive*, which works to our advantage – it works itself very well into a computer program! Let's write the Forward Elimination procedure out in pseudo code and make a code in `MATLAB`! In the below example, recall that n is the number of equations in our system of equations.

Pseudo Code	**MATLAB Code**
Index the current pivot row with i. Cycle from the first row to the second last row. *Note:* Keeping track of the pivot row	`for i = 1:(n-1)`
Cycle through all rows below the pivot element, leaving the pivot row unchanged.	`  for k = (i+1):n`
Calculate and store the *row factor*, which is the ratio of $a_{j,i}$ to the pivot element $a_{i,i}$.	`    factor = A(k,i) / A(i,i)`
Indexing the column position with k, move horizontally across all columns of the matrix starting from the pivot element.	`    for j = i:n`
Overwrite each current element with the new element. Effectively, we are doing the following row subtraction element by element: $(current\ row) - (row\ factor) \times (pivot\ row)$	`      A(k,j) = A(k,j) - factor*A(i,j)`
Terminate k loop when have moved across all columns.	`    end`
Do the same to $\boldsymbol{b}$ as was done to A.	`    b(k) = b(k) - factor*b(i)`
Terminate j loop when have cycled through all rows below the pivot element.	`  end`
Terminate when have cycled through all rows except the last row. All terms beneath the diagonal have now been zeroed!	`end`

For a visual perspective of the Forward Elimination algorithm as MATLAB reads it, observe the following flowchart:



Step 2 - Backward Substitution

Goal: Find the solution to the LSoE, $\boldsymbol{x}$! We do this by solving for the final value of $\boldsymbol{x}$ first (x_n) via the final equation, use the $(\boldsymbol{n-1})^{\text{th}}$ equation to solve for x_{n-1}, and so on.

Recall, you have something like this:
(an upper triangular matrix and a manipulated $\boldsymbol{b}$ vector)

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix} \quad \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Let's take a look at the first system for which we performed Forward Elimination on before…

Workshop: Backward Substitution

Continuing with our matrix, use Backward Substitution to determine x_i, the production rates at each refinery i:

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 2 & 0 \\ 0 & 0 & -4 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 19 \\ 6 \\ -20 \end{bmatrix}$$

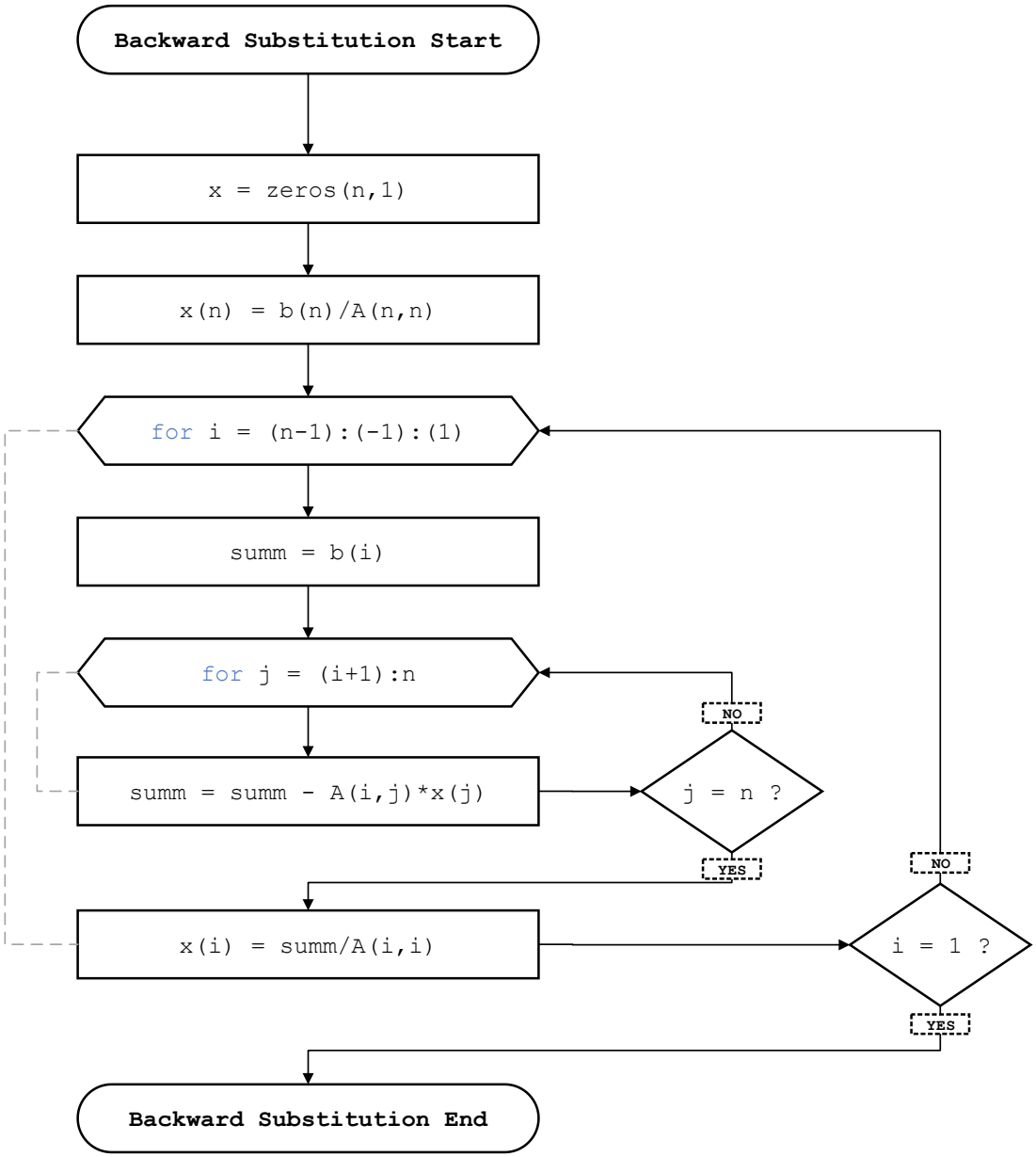
Now let's bring back the algebra! Work from the bottom up to determine the values of x_i :

For our $\mathbf{3 \times 3}$ Matrix:	In General:
$Working \uparrow \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix}$ and $\begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$	$Working \uparrow \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,3} \\ 0 & a_{2,2} & \dots & a_{2,3} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{bmatrix}$ and $\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$
$x_3 = \frac{b''_3}{a''_{3,3}}$	$x_n = \frac{b_n}{a_{n,n}}$
Now that we know the value of $\boldsymbol{x_3}$, we can use it to solve for x_2 in the second row: $x_2 = \frac{b'_2 - a'_{2,3}x_3}{a'_{2,2}}$	$x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$
Using x_2 & $x_3$to solve for x_1: $x_1 = \frac{b_1 - a_{1,2}x_2 - a_{1,3}x_3}{a_{1,1}}$	$x_{n-2} = \frac{b_{n-2} - a_{n-2,n-1}x_{n-1} - a_{n-2,n}x_n}{a_{n-2,n-2}}$ Continued until we have found vales for all x_i. The general equation is below: $\boldsymbol{x_i = \left(b_i - \sum_{j=i+1}^{n} a_{i,j}x_j \right)\left(\frac{1}{a_{i,i}}\right)}$

Just like Forward Elimination, Backward Substitution is repetitive and thus is great for being programmed! Let's write the procedure out in pseudo code and make a code in MATLAB!

Pseudo Code	**MATLAB Code**
Initialize a solution vector $\boldsymbol{x}$ that is n rows by 1 column. This will be used to store our solution.	`x = zeros(n,1)`
Compute $x(n)$ immediately – the n^{th} row of the A matrix after Forward Elimination only has 1 non-zero element.	`x(n) = b(n) / A(n,n)`
Cycle from the second last row to the first row in steps of -1, since we need to solve for all *previous* values of $\boldsymbol{x}$.	`for i = (n-1):(-1):(1)`
Start a dummy variable '`summ`' that we can overwrite to eventually give us the term $\left(\boldsymbol{b_i} - \sum_{j=i+1}^{n} \boldsymbol{a_{i,j}x_j}\right)$. We can start with $b(i)$.	`summ = b(i)`
Cycle through all terms after the current x that we are solving for. All terms to the left of $x(i)$ will be zero, and all terms to the right will have been previously solved for!	`for j = (i+1):n`
Overwrite our original sum by subtracting off the $A(i,j) \times x(j)$ terms in that row for previously solved $\boldsymbol{x}$ values.	`summ = summ - A(i,j)*x(j)`
Terminate the j loop when every x value in that row has been accounted for.	`end`
Solve for $x(i)$.	`x(i) = summ / A(i,i)`
Terminate the i loop when all $x(i)$ have been computed.	`end`

A visual perspective of the Backward Substitution algorithm as MATLAB reads/executes it is shown in the following flowchart:



Key 2E04 Insight*: "Repetitive, 'grindy' ways of doing things are what we strive for, because computers are very good at those things." – Steven*

Computational Complexity for Gauss Elimination

Remember our definition of computational complexity? For Gauss Elimination, we have the following:

	Forward Elimination	Backward Substitution	Overall GE
Number of FLOPs	$\frac{2n^3}{3} + \mathcal{O}(n^2)$	$\frac{n^2}{2} + \mathcal{O}(n)$	$\frac{2n^3}{3} + \mathcal{O}(n^2)$

Look familiar? This was the method we used in our example for computational complexity earlier! As you can see, Gauss Elimination has an overall order of $\mathcal{O}(n^3)$ due to the Forward Elimination step. Later in this module, we will look at LU decomposition, which takes advantage of this in certain cases.

Gauss-Jordan Elimination

Gauss-Jordan Elimination

Goal: Manipulate our original coefficient matrix, A, to become an *identity matrix*, containing ones along the diagonal and zeros everywhere else! We can read the answer directly from here (no need for Backward Substitution!).

General Approach: Normalize each current pivot row (the pivot element will become equal to 1) and then eliminate ALL other elements above and below the pivot element to be zero (skipping the diagonal element).

Question: What does it mean to 'normalize' something?

Answer: Normalizing refers to a process which adjusts a set of values into a set of values with a common scale. Typically, a significant value in this set will be chosen to normalize the set – this will cause that value itself to become 1, and all other values will be scaled based on it (See below!).

Algebraically it looks like this:

$\underbrace{\begin{bmatrix} a_{1,1} & a_{2,1} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}}_{\boldsymbol{b}}$	1. Initial coefficient matrix and $\boldsymbol{b}$ vector.	Goal
$\underbrace{\begin{bmatrix} 1 & \frac{a_{2,1}}{a_{1,1}} & \frac{a_{1,3}}{a_{1,1}} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} \frac{b_1}{a_{1,1}} \\ b_2 \\ b_3 \end{bmatrix}}_{\boldsymbol{b}}$	2. Normalize the first row by dividing each term by $a_{1,1}$ (the pivot element.)	
$\underbrace{\begin{bmatrix} 1 & a'_{2,1} & a'_{1,3} \\ 0 & a'_{2,2} & a'_{2,3} \\ 0 & a'_{3,2} & a'_{3,3} \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \end{bmatrix}}_{\boldsymbol{b}}$	3. Perform row operations to eliminate coefficients below the pivot element. Recall: The prime symbol (') means that an operation has been performed to the original variable.	$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} b'''_1 \\ b'''_2 \\ b'''_3 \end{bmatrix}}_{\boldsymbol{b}}$ Diagonal Matrix!
$\underbrace{\begin{bmatrix} 1 & a'_{2,1} & a'_{1,3} \\ 0 & 1 & a''_{2,3} \\ 0 & a'_{3,2} & a'_{3,3} \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} b'_1 \\ b''_2 \\ b'_3 \end{bmatrix}}_{\boldsymbol{b}}$	4. Perform Step 2, now using the second row as the pivot row, and $a'_{2,2}$ as the pivot element.	
$\underbrace{\begin{bmatrix} 1 & 0 & a''_{1,3} \\ 0 & 1 & a''_{2,3} \\ 0 & 0 & a''_{3,3} \end{bmatrix}}_{A} \quad \underbrace{\begin{bmatrix} b''_1 \\ b''_2 \\ b''_3 \end{bmatrix}}_{\boldsymbol{b}}$	5. Perform row operations to eliminate coefficients above and below the pivot element. Continue these steps for remaining columns / rows.	

Workshop: Editing Forward Elimination to Make Gauss-Jordan Elimination

Provided below is our previous Forward Elimination code from Gauss Elimination. Modify this code by making a few small changes to generate a *diagonal matrix* instead of a lower triangular matrix.

Bonus: Space is provided below for you to add the 'Back-Solve' step for Gauss-Jordan Elimination as well! Remember, because of the diagonal configuration of A, we *do not need* Backward Substitution.

Forward Elimination

```
for i = 1:(n-1)



    for k = (i+1):n

        factor = ( A(k,i) ) / A(i,i);

        for j = i:n

            A(k,j) = A(k,j) -
            factor*(A(i,j));

        end

        b(k) = b(k) - factor*(b(i));

    end

end
```

Gauss-Jordan Elimination

```
for i = 1:n

    A(i,:) = (A(i,:)) / A(i,i);
    b(i) = b(i) / A(i,i);

    for k =

        if

            factor =

            for j = i:n

                A(k,j) = A(k,j) -
                factor*(A(i,j));

            end

            b(k) = b(k) - factor*(b(i));

        end
    end
end

%%% Back-solve step --> solve for x %%%
```

Figure 1: Minion going 'Tada'!

LU Decomposition

This method *decomposes* a given coefficient matrix, A, into *two* separate matrices, a *Lower triangular matrix* (L), and an *Upper triangular matrix* (U).

LU Decomposition

An $n \times n$ coefficient matrix, A, can be decomposed into the product of two matrices, L and U, where L is a lower triangular matrix, and U is an upper triangular matrix.

$$A = L \times U$$

$$\underbrace{\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ 0 & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n} \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} l_{1,1} & 0 & \dots & 0 \\ l_{2,1} & l_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \dots & l_{n,n} \end{bmatrix}}_{L} \times \underbrace{\begin{bmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ 0 & u_{2,2} & \dots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{n,n} \end{bmatrix}}_{U}$$

In this module, we will investigate *one* (of two) algorithms used to define L and U such that $A = L \times U$ is satisfied: *The Gaussian Elimination Method*.

A 3×3 coefficient matrix, A, is written below as the product of L and U matrices:

$$\underbrace{\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} \end{bmatrix}}_{L} \times \underbrace{\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ 0 & u_{2,2} & u_{2,3} \\ 0 & 0 & u_{3,3} \end{bmatrix}}_{U}$$

Using the rules of matrix multiplication, we can write each term of the coefficient matrix, A, in the following way:

$a_{1,1} = l_{1,1}u_{1,1}$	$a_{1,2} = l_{1,1}u_{1,2}$	$a_{1,3} = l_{1,1}u_{1,3}$
$a_{2,1} = l_{2,1}u_{1,1}$	$a_{2,2} = l_{2,1}u_{1,2} + l_{2,2}u_{2,2}$	$a_{2,3} = l_{2,1}u_{1,3} + l_{2,2}u_{2,3}$
$a_{3,1} = l_{3,1}u_{1,1}$	$a_{3,2} = l_{3,1}u_{1,2} + l_{3,2}u_{2,2}$	$a_{3,3} = l_{3,1}u_{1,3} + l_{3,2}u_{2,3} + l_{3,3}u_{3,3}$

Workshop: DOF Analysis for L and U

For our 3×3 coefficient matrix, let's answer the following questions:

How many *unknown variables* are there? How many *equations* do we have? How many *DOF* do we have?

From the workshop above, you found that we have -3 degrees of freedom. If we generalize this, we would have – n *degrees of freedom* for any $n \times n$ matrix – an under-specified system! This means that we must define n variables in order to make the system able to solve.

In the Gaussian Elimination Method of LU Decomposition, we will choose to set the n diagonal elements in the Lower Triangular Matrix to be 1! We'll see why!

Workshop: Deriving $\boldsymbol{L}$ and $\boldsymbol{U}$ (Gaussian Elimination Method)

From the name (Gaussian Elimination Method of LU Decomposition), you may have guessed that Gauss Elimination may be involved in the method somehow!

We have chosen n elements on the diagonal of L to be 1. Let's rearrange the equations that we derived above to solve for the elements of L and U.

Hint: Rearrange these equations to solve for $u_{i,j}$:

$a_{1,1} = l_{1,1}u_{1,1} \rightarrow$

$a_{1,2} = l_{1,1}u_{1,2} \rightarrow$

$a_{1,3} = l_{1,1}u_{1,3} \rightarrow$

Hint: Rearrange these equations for $l_{i,j}$:

$a_{2,1} = l_{2,1}u_{1,1} \rightarrow$

$a_{3,1} = l_{3,1}u_{1,1} \rightarrow$

Hint: Rearrange these equations for $u_{2,2}$, $u_{2,3}$:

$a_{2,2} = l_{2,1}u_{1,2} + l_{2,2}u_{2,2} \rightarrow$

$a_{2,3} = l_{2,1}u_{1,3} + l_{2,2}u_{2,3} \rightarrow$

Hint: Rearrange this equation for $l_{3,2}$:

$a_{3,2} = l_{3,1}u_{1,2} + l_{3,2}u_{2,2} \rightarrow$

Hint: Rearrange this equation for our last unknown variable ($u_{3,3}$):

$a_{3,3} = l_{3,1}u_{1,3} + l_{3,2}u_{2,3} + l_{3,3}u_{3,3} \rightarrow$

From the workshop, we can see that U is the forward-eliminated version of the coefficient matrix A, while L stores the row factors! Amazing!

Defining $\boldsymbol{L}$ and $\boldsymbol{U}$ (Gaussian Elimination Method)

Shown below is the general form for L and U using the Gaussian Elimination Method:

$$U = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ 0 & a'_{2,2} & \dots & a'_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{n,n}^{(n-1)} \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ r_{2,1} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ r_{n,1} & r_{n,2} & \dots & 1 \end{bmatrix}$$

So how do we apply all this to solving a system of equations in the form of $\boldsymbol{Ax} = \boldsymbol{b}$?

Solving $\boldsymbol{Ax} = \boldsymbol{b}$ using LU Decomposition

Consider an $n \times n$ matrix A that is invertible. It may be decomposed into the product of two matrices L and U:

$$\begin{aligned} A\boldsymbol{x} &= \boldsymbol{b} && \textit{And} && [L][U] = A \\ \therefore [L][U]\boldsymbol{x} &= \boldsymbol{b} && \textit{If} && [U]\boldsymbol{x} = \boldsymbol{y} \\ \textit{Then}\ [L]\underbrace{[U]\boldsymbol{x}}_{\boldsymbol{y}} &= \boldsymbol{b} && \Rightarrow && [L]\boldsymbol{y} = \boldsymbol{b} \end{aligned}$$

Summary:

1. Solve $\boldsymbol{Ly} = \boldsymbol{b}$ for $\boldsymbol{y}$. This can be done using *Forward Substitution*, due to the shape of L.
2. Use your new $\boldsymbol{y}$ column vector in the equation $\boldsymbol{Ux} = \boldsymbol{y}$ to solve for $\boldsymbol{x}$! This can be done with our regular Backward Substitution.

Note: L and U are placed in square brackets to show the operations more clearly.

Programming Tweaks for LU Decomposition

Programming LU Decomposition into a computer only requires a few changes from the Gauss-Elimination code! The GE algorithm will return U. For L, just add an identity matrix to store the $r_{i,j}$ elements at each step. After that, you can split into the two-step process, first solving $\boldsymbol{Ly} = \boldsymbol{b}$ for $\boldsymbol{y}$ using Forward Substitution, then solving for $\boldsymbol{x}$ in $\boldsymbol{Ux} = \boldsymbol{y}$ using Backward Substitution!

Sweet! We know how to find the L and U, and from before, we know how to solve for $\boldsymbol{x}$ using L and U. We already know how to do Backward Substitution from Gauss Elimination to find the $\boldsymbol{x}$ when we have $\boldsymbol{Ux} = \boldsymbol{y}$, but how do we do the Forward Substitution step when solving $\boldsymbol{Ly} = \boldsymbol{b}$ in LU decomposition?

Workshop: Forward Substitution for LU Decomposition

Provided below is a rough formatting of a code for Backward Substitution on the LSoE, $\boldsymbol{Ux = y}$, where $\boldsymbol{U}$ is upper triangular. How would we modify this code to resemble Forward Substitution for the LSoE $\boldsymbol{Ly = b}$, where $\boldsymbol{L}$ is lower triangular? (it's like reverse Backward Substitution ☺)

Backward Substitution on $\boldsymbol{Ux = y}$

```
x = zeros(n,1);
x(n) = y(n) / U(n,n);

for i = (n-1):(-1):(1)

    summ = y(i);

    for j = (i+1):n

        summ = summ - U(i,j)*x(j);

    end

    x(i) = summ / U(i,i);

end
```

Forward Substitution on $\boldsymbol{Ly = b}$

```
y = zeros(n,1);

y(1) =  [          ]

for i = [          ]

    summ = b(i);

    for j = [          ]

        summ = summ - L(i,j)*y(j);

    end

    y(i) = summ / 1;

end
```

Strengths and Applications of LU Decomposition

Strength: Potential Decrease in Computational Requirement

Decomposition can be useful for solving LSoE which have the *same coefficient matrix,* $\boldsymbol{A}$, but *different values* of $\boldsymbol{b}$. For example, we could have a situation where in one calculation, $\boldsymbol{Ax = b_1}$, but in another, $\boldsymbol{Ax = b_2}$, and so on….

This is because LU Decomposition breaks the coefficient matrix into $\boldsymbol{L}$ and $\boldsymbol{U}$ *without* requiring changes to $\boldsymbol{b}$. Some Chemical Engineering examples that may do this include:

- Changing inputs to a steady-state mass or energy balance
- Changing reference conditions (temperatures, pressures, *etc.*) for an equation system
- Performing a *sensitivity analysis* by changing equation constants or parameters

Key Advantage of LU Decomposition

Breaking a matrix $\boldsymbol{A}$ into the product of an upper and lower triangular matrix before using it with $\boldsymbol{b}$ has the same computational complexity as Forward Elimination ($\mathcal{O}(n^3)$, the slowest step of Gauss Elimination).

But we'll only need to do this ONCE if $\boldsymbol{A}$ doesn't change - different RHS $\boldsymbol{b}$ vectors only require the Backward Substitution step, returning a solution n times faster! If you have $n = 5000$ mass and energy balances, this means that the second solution is 5000 TIMES FASTER than the first! 🔥

Example: Demonstrating the Efficiency of LU Decomposition

For a 1000×1000 LSoE:

First Solution:	10 seconds	All Following Solutions:	0.01 seconds ($1000\times$faster!)
Computation Complexity:	$\mathcal{O}(n^3)$	*Computation Complexity*:	$\mathcal{O}(n^2)$

Remember that to achieve these benefits, the coefficient matrix, A, must be held constant for varying $\boldsymbol{b}$ vectors.

Application: Inverting a Matrix with LU Decomposition:

Another useful application of LU Decomposition is that it allows us to easily and efficiently invert a matrix:

Suppose $A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$ $\quad A^{-1} = W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix}$

Remember that $AA^{-1} = I$.
$\therefore$ It is also true that A multiplied by the first column of A^{-1} is equal to the first column of I.

$A \times W_{i,j=1} = I_{i,j=1}$	$A \times W_{i,j=2} = I_{i,j=2}$	$A \times W_{i,j=3} = I_{i,j=3}$
$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} w_{1,1} \\ w_{2,1} \\ w_{3,1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} w_{1,2} \\ w_{2,2} \\ w_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$	$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} w_{1,3} \\ w_{2,3} \\ w_{3,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

Looks familiar? We have one A matrix with varying $\boldsymbol{b}$ vectors (each $\boldsymbol{b}$ vector is a column of the Identity matrix)! Each $\boldsymbol{x}$ solution represents a column in the inverse matrix!

Using LU, finding the inverse requires the same computational complexity as our LU Decomposition method described previously: $\mathcal{O}(n^3)$ for the first column, and $O(n^2)$ for each remaining columns.

Imagine This:

One of the classic ways of finding A^{-1} is to use the formula $A^{-1} = \frac{1}{\det(A)} \times adj(A)$, where $adj(A)$ is the *adjoint* of A. This method has computational complexity of $\mathcal{O}(\ (n!)^2\ )$ FLOPs. A *very demanding* amount.

Imagine if $n = 1000$. Type $1000!$ into your calculator and read what it says. WOOF! Thank goodness for LU Decomposition.

Partial Pivoting

To demonstrate the importance of partial pivoting, let's complete the following workshop:

Workshop: Identifying the Error in Forward Elimination

ONE of the below systems of linear equations is *guaranteed* to fail when it is solved with Gauss elimination. From the options below, *circle* the LSoE I am referring to.

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 6 & 10 & 3 & 4 \\ 2 & 5 & 1 & 11 \\ 8 & 7 & 0 & 20 \end{bmatrix} \quad \boldsymbol{b}_1 = \begin{bmatrix} 4 \\ 10 \\ 12 \\ -2 \end{bmatrix} \qquad A_2 = \begin{bmatrix} 6 & 4 & 6 & 8 \\ 0 & 2 & 8 & 4 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 2 & 8 \end{bmatrix} \quad \boldsymbol{b}_2 = \begin{bmatrix} 4 \\ 1 \\ -0.5 \\ 8 \end{bmatrix} \qquad A_3 = \begin{bmatrix} 1 & 2 & 7 & 1 \\ 0 & 1 & 8 & 2 \\ 5 & 3 & 3 & 8 \\ 0 & 1 & 5 & 0 \end{bmatrix} \quad \boldsymbol{b}_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Write out the calculation which Forward Elimination will try to perform, and indicate *what type of error* this is:

In an LSoE, $\boldsymbol{Ax} = \boldsymbol{b}$, we have learned that each distinct row of $\boldsymbol{A}$ and $\boldsymbol{b}$ correspond to an independent equation. Therefore, the order in which we insert equations into $\boldsymbol{A}$ and $\boldsymbol{b}$ is arbitrary – we can swap the location of equations in the LSoE if needed with no effect on the solution vector $\boldsymbol{x}$.

Swapping the placement of equations in an LSoE is called *partial pivoting.*

Workshop: Preventing the Error in Forward Elimination

The LSoE that you indicated in the previous workshop will crash on the next step of Forward Elimination due to an error. Partial pivoting can be used to avoid this error, and allow Forward Elimination to proceed.

Swap the appropriate equations of your indicated LSoE by partial pivoting, and write the new A and b below:

As shown in the workshop above, partial pivoting can be used to *improve the robustness* of the direct methods that we've discussed – Gauss Elimination, Gauss-Jordan Elimination, and LU Decomposition. (In the next module, we'll see that partial pivoting can also be used to for achieving 'diagonal dominance'.)

Partial Pivoting

The partial pivoting algorithm is as follows, and is performed *every time the pivot element changes*:

1. Check whether the current pivot element is zero or very close to zero.
   - IF $a_{i,i} = 0$, enter partial pivoting code.
   - ELSE continue with Forward Elimination.

2. Staying within the same column as the pivot element, loop through the elements below the pivot element checking for non-zero elements.
   - IF $a_{h,i} \neq 0$, $h = [i+1, \dots, n]$
     break the loop. The equation in the LSoE containing this non-zero element will be chosen to be pivoted.

3. Pivot the equation from (1) with the selected equation from (2).
   - Store Equation 1 in a 'dummy variable'. This requires storing the elements both in A and $\boldsymbol{b}$.
     i. $StoreRow1_A = Row1_A$
     ii. $StoreRow1_{\boldsymbol{b}} = Row1_{\boldsymbol{b}}$
   - Set Equation 1 equal to Equation 2.
     i. $Row1_A = Row2_A$
     ii. $Row1_{\boldsymbol{b}} = Row2_{\boldsymbol{b}}$
   - Set Equation 2 equal to the 'dummy variable'.
     i. $Row2_A = StoreRow1_A$
     ii. $Row2_{\boldsymbol{b}} = StoreRow1_{\boldsymbol{b}}$

After partial pivoting has been performed, you can proceed with Forward Elimination.

Partial Pivoting: Thorough Example

Refer to the file 'LivePivot.mlx' in the supplementary material for a more thorough investigation of partial pivoting as it is applied to an example using Gauss Elimination. The material is written as a 'Live Script' in MATLAB, which allows users to easily present information, codes, and their code's outputs, all in the same file! To view the material, open the file in MATLAB, and click 'run all'. Then feast your eyes.

Conclusion and Summary

In this module, we have:

- Analyzed three methods to directly solve an LSoE: Gauss Elimination, Gauss-Jordan Elimination and LU decomposition.
- Compared the strengths and potential drawbacks of each direct method with regards to computational complexity and rounding error.
- Employed partial pivoting in an example, summarized the general algorithm, and discussed the benefits of adding partial pivoting as a security method into our codes.

Next up: Iterative Methods to Solve LSoE!