Mathematics For Me

Tommy Nguyen

November 2024

Chapter 1

Linear Algebra

1.0.1 The Beginning, Systems of Linear Equations

When we first started learning about variables in middle school, one of the earliest forms of an equation we were introduced to may have looked something like this:

$$mx + b = c$$

Your first instinct is probably, "Oh look! It's the slope-intercept formula!" and you'd be right! The slope-intercept formula is an equation representing a line in 2D space (because you can move in the x and y direction).

As we move along and learn about more complex equations, we most likely encountered equations of higher dimensions (i.e., containing more variables). For example:

$$ax + +cz = d \text{ Plane in 3D space.} \tag{1.1}$$

$$ax_1 + by + cz + dx_2 = e \text{ Something in 4D space.} \tag{1.2}$$

$$a_1x_1 + a_2x_2 + ... + a_nx_n = d \ n\text{D space.} \tag{1.3}$$

As you might have guessed from the equations above, we can easily scale our **linear equations** to higher dimensions just by adding another **constant** multiplied by a new **variable**. You might have been thrown off by the number of terms introduced, so let's give them a proper definition!

- **Constant**: Any term that does not change. For example, in the equation $2x + 3$, 2 and 3 are constants because we know their value. We also know that 2 and 3 are definite values, meaning they do not change at all, regardless of what x is. In the equations provided above, a, b, c, a_1, a_2, etc are constants.
- **Variable**: Any value that will change. Typically, these are represented by an alphabetic letter (x, y, z, etc). We are usually trying to solve the

equation to find the value for these variables. For example, we might be provided a question that asks us to solve for x in: 10 = 2x + 5. In this case, x is the variable we are solving for.

- **Linear Equation**: A simple equation that *does not involve any exponents or square roots of a variable.* Basically, if you see equations like: $x^2+x=0$ or $x*y+x=2$, then they are NOT linear equations. Equations like: $x+2=10$ or $x+y+z=10$ would be considered linear equations (because they do not involve any powers or square roots, which would alter the **linearity** of the equation, basically it will no longer be a straight line).

In any linear equation, the *constants cannot all be 0's.* "Why not?" well, because if they were, then the equation would just be 0 = some number (remember, 0 times anything is going to be 0)! That's not going to be a straight line, or even a point! However, the d and e (shown in equation 1.3) CAN be 0. Any linear equation where the other side of the = is 0 is called a **homogeneous linear equation**.

Generally, a **linear equation in n variables** (where n is the number of variables for the linear equation) can be expressed via the equation shown in 1.3 (the nD one)!

There are more of them now!

Just like how we can add more variables, we can also add more equations.

A finite set of linear equations is called a **systems of linear equations** or **linear system** (we will be calling them linear systems from now on for ease of typing). In a linear system, the variables may also be called **unknowns**.

The equation below is an example of a simple linear system in 2 variables (x_1 and y_1):

$$\begin{aligned} 4x_1 + 10y_1 &= 6 \\ 6x_1 - y_1 &= 10 \end{aligned} \tag{1.4}$$

Again, like the number of variables...we can scale the number of equations as much as we want!

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_3x_3 + ... + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n &= b_m \end{aligned} \tag{1.5}$$

For a general linear system, we can say that a **solution** in n unknowns (x_1, x_2, ..., x_n) is a sequence of n numbers that, when substituted into unknowns x_1, x_2, ..., x_n, will make all linear equations in the linear system to be true.

Let's walk through an example. Given the linear system below, solve for x and y:

$$\begin{aligned} 2x - y &= 7 \\ 3x - 2y &= 10 \end{aligned} \tag{1.6}$$

Let's use the good old elimination method (if you need review, here is a great link: Khan Academy is great!). What variables can we easily eliminate? $-2y$ seems like a good candidate! Let's multiple $2x - y = 7$ by -2 and add the results to $3x - 2y = 10$.

$$\begin{aligned} 2x - y &= 7 \\ 3x - 4x - 2y + 2y &= 10 - 14 \\ \texttt{=>} \quad -x &= -4 \\ \texttt{=>} \quad x &= 4 \end{aligned} \tag{1.7}$$

Let's plug $x = 4$ back into our first equation to find y!

$$\begin{aligned} 2 * (4) - y &= 7 \\ \texttt{=>} \quad 8 - y &= 7 \\ \texttt{=>} \quad y &= 1 \end{aligned} \tag{1.8}$$

Now we know equation 1.6 has a solution: $x = 4$ and $y = 1$. We can also write the solution as coordinates too: (4, 1). More often times than not, we'll be writing the solution as coordinates because it is a lot easier to write out and visualize. Furthermore, we can think of the solutions to our linear systems as coordinates where all of the equations in the linear system intersect (cross). For example:

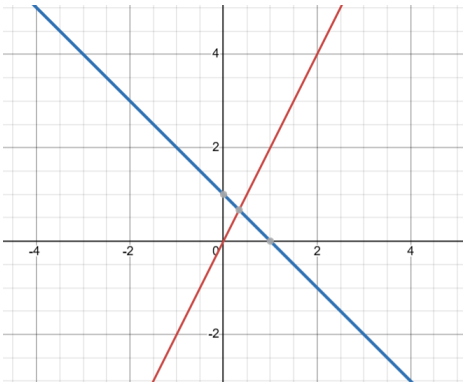

Figure 1.1: The two lines shown intersect only at point (0.33333, 0.66667).

Figure 1.1 shows two lines intersecting at one point. That is, it would only have **1 solution**. Okay, that is easy to understand...but what about other

scenarios? What if there are so many solutions, we can't really pick one and be done? In that scenario, you have **infinitely many solutions**! You can think of it like *two lines that are on top of each other* or *two planes stacked on top of one another*.

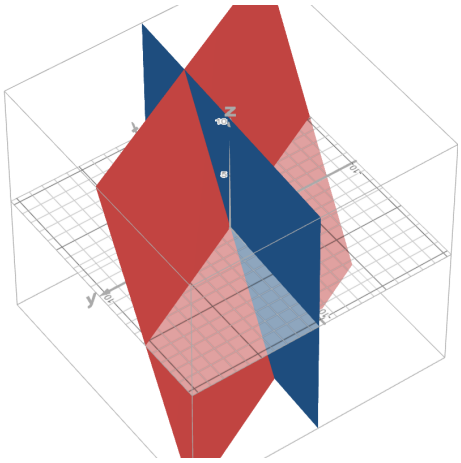

Figure 1.2: The red plane and the blue plane share an entire line where they both intersect each other. If you think back to calculus (specifically limits), you'll remember that there are infinitely many steps you can take along a line. In turn, the two planes share an infinite number of points of intersection. Likewise, if you stacked the plane on top of each other, they will also have infinitely many points of intersection, because they are basically crossing at every point!

Figure 1.2 showcases an example of infinitely many solutions. The two planes cross along an entire line, this indicates that the two planes share **MANY** points of intersections. You can think of it like this: If we stretched the planes out into infinity, the points of intersection will ALSO stretch to infinity.

In situations where we are expected to give an answer, but there are just too many answers to give, mathematicians created a neat way to provide a solution without explicitly mentioning a point! We call this way of writing our solution, **parametric equations**. Long story short, we rewrite one of the linear equations in our linear system. For example, the equation below has infinitely many solutions:

$$\begin{aligned} 4x + 2y &= 1 \\ 16x + 8y &= 4 \end{aligned} \tag{1.9}$$

If you tried solving for the linear system above, you would ultimately get $0 = 0$. This means that any solution that solves for $4x + 2y = 1$, will also solve for $16x + 8y = 4$ and vise versa! If you looked closely, you'll also notice that the *two equations are just scalar multiples of each other*!

Okay, now that we know for sure the linear system has infinitely many solutions, let's write a parametric equation! The steps are easy, really:

1. Rewrite the equation so that one variable is isolated to one side.
2. Substitute all other variables (except for the isolated variables) with a different letter.

For the first linear equation in 1.9, one of the parametric equation can be:

$$\begin{aligned} 4x + 2y &= 1 \\ \texttt{=>}\ 4x &= 1 - 2y \\ \texttt{=>}\ x &= \frac{1}{4} - \frac{y}{4} \\ \texttt{=>}\ x &= \frac{1}{4} - \frac{t}{4}, y = t \end{aligned} \tag{1.10}$$

If you had more unknowns, it's basically the same thing:

$$\begin{aligned} 2x - 3y + z - 9d &= 10 \\ \texttt{=>}\ 2x &= 10 + 3y - z + 9d \\ \texttt{=>}\ x &= \frac{10}{2} + \frac{3y}{2} - \frac{z}{2} + \frac{9d}{2} \\ \texttt{=>}\ x &= \frac{10}{2} + \frac{3t}{2} - \frac{r}{2} + \frac{9s}{2}, y = t, z = r, d = s \end{aligned} \tag{1.11}$$

Basically what we're doing in the last step is assigning a **parameter** to every unknown variable (except for the isolated variable).

There are some scenarios where you might also get **no solutions** too!

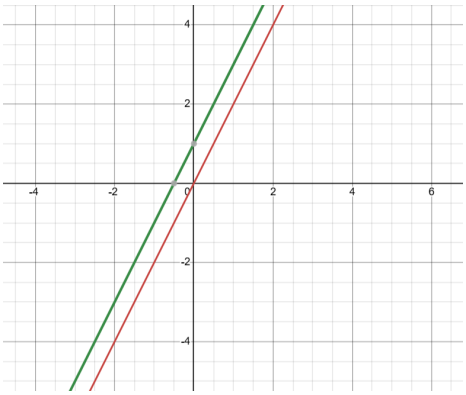

Figure 1.3: The lines are parallel, they will never cross.

Based on our investigation of one solution and infinitely many solutions, can you guess why Figure 1.3 has no solutions? If you said it was because they never cross, then you would be correct!

The two lines in Figure 1.3 never cross, they never intersect. This means that there are no solutions for x and y that satisfies both linear equations.

An example of a linear system with no solutions would be:

$$\begin{aligned} x + y &= 4 \\ 2x + 2y &= 12 \end{aligned} \tag{1.12}$$

If we tried solving for that linear system, we would end up with $0 = 4$, which will never be true.

Okay...at this point you may be thinking, "Tommy, WHY are you saying all of this now? This seems like its coming out of left field!". Well, my curious and mathematically driven reader, the reason why I bring this up now is because of an interesting rule/property that comes with linear systems: **Every linear systems has zero, one or infinitely many solutions. There are no in betweens!** As such, a linear system can be **consistent** (if it has at least 1 solution) or **inconsistent** (if it has no solutions).

The Matrix: Not the 1999 Film

As we add more equations and more unknowns, it becomes increasingly more difficult to perform computations on them, both in terms of manual computation and on the computer.

For one, keeping track of all of the "+", "-", unknowns, etc will become increasingly more difficult the more equations and unknowns we have. To combat the visual complexity of a large linear system, we introduce **augmented matrices** (**augmented matrix** for 1 matrix).

Long story short, we can convert our long and tedious-to-write-out linear system into a matrix of numbers. So our long equation below:

$$\begin{aligned} 2x_1 + 3x_2 + 4x_3 + ... + 10x_n &= 20 \\ &. \\ &. \\ &. \\ 20x_1, +30x_2 + 12x_3 + ... + x_n &= 198 \end{aligned} \tag{1.13}$$

Can turn into the matrix:

$$\begin{bmatrix} 2 & 3 & 4 & ... & 10 & 20 \\ . & & & & & \\ . & & & & & \\ . & & & & & \\ 20 & 30 & 12 & ... & 1 & 198 \end{bmatrix}$$

As you may have noticed, the last column on the right represents the numbers on the right-hand side of the equation 1.13. Broadly, all numbers on the right-most column will be the c in $mx + b = c$, and all other columns will have the coefficients of the unknowns.

Here are a few more examples so you can see what an augmented matrix may look like.

$$\begin{aligned} 2x - 3y &= 6 \\ x + 10y &= 7 \end{aligned} \tag{1.14}$$

$$\begin{bmatrix} 2 & -3 & 6 \\ 1 & 10 & 7 \end{bmatrix}$$

$$\begin{aligned} 3x - 1y - 20z &= 6 \\ 10x + 5y + 8z &= 7 \\ 10x + y + .5z &= 7 \end{aligned} \tag{1.15}$$

$$\begin{bmatrix} 3 & -1 & -20 & 6 \\ 10 & 5 & 8 & 7 \\ 10 & 1 & .5 & 7 \end{bmatrix}$$

Now that we know what an augmented matrix looks like, we can start describing how to perform operations on said matrix.

It's Elementary, My Dear Augmented Matrix!

When we perform operations on an augmented matrix to solve for unknowns, we are perform what are called **elementary row operations**. When it comes to row operations, there are 3 operations we can perform:

1. Multiply a row by a non-zero number.
2. Swap rows.
3. Add one row multiplied by a constant to another row.

As you can see, these operations are performed on rows, much like the operations we performed in previous examples were performed on one equation.

Let's explain the reason behind each operation:

- **Multiply a row by a non-zero number**: We perform this operation to get nice numbers. In other words, whenever we can multiply an entire row by a constant to get nice whole numbers, we probably should.
- **Swap rows**: We perform this operation just so we can get a matrix where the solutions are on the left diagonal (we will see an example soon). Long story short, we do this so we can easily see the answers as well. Another possible reason might be that we want to perform operations on the matrix where the left diagonal has non-zero values.
- **Add one row multiplied by a constant to another row**: Same reason for the steps we took in previous examples; we want to isolate a single unknown in one equation so we can solve for that single unknown.

Lets go through a brief example. Say we had the following augmented matrix:

$$\begin{bmatrix} 2 & 10 & 4 & 6 \\ 3 & 2 & 4 & 1 \\ 1 & 2 & 1 & 3 \end{bmatrix}$$

Here is a step-by-step walkthrough:

$$\text{Swap row 1 with row 2.} \begin{bmatrix} 1 & 2 & 1 & 3 \\ 3 & 2 & 4 & 1 \\ 2 & 10 & 4 & 6 \end{bmatrix}$$

$$\text{Swap row 2 with row 3.} \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 10 & 4 & 6 \\ 3 & 2 & 4 & 1 \end{bmatrix}$$

$$\text{Multiply row 1 by -2 and add to row 2.} \begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 6 & 2 & 0 \\ 3 & 2 & 4 & 1 \end{bmatrix}$$

Multiply row 1 by -3 and add to row 3. $\begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 6 & 2 & 0 \\ 0 & -4 & 1 & -8 \end{bmatrix}$

Multiply row 2 by 2/3 and add to row 3. $\begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 6 & 2 & 0 \\ 0 & 0 & 7/3 & -8 \end{bmatrix}$

Multiply row 2 by -1/3 and add to row 1. $\begin{bmatrix} 1 & 0 & 1/3 & 3 \\ 0 & 6 & 2 & 0 \\ 0 & 0 & 7/3 & -8 \end{bmatrix}$

Multiply row 3 by -6/7 and add to row 2. $\begin{bmatrix} 1 & 0 & 1/3 & 3 \\ 0 & 6 & 0 & 48/7 \\ 0 & 0 & 7/3 & -8 \end{bmatrix}$

Multiply row 3 by -1/7 and add to row 1. $\begin{bmatrix} 1 & 0 & 0 & 29/7 \\ 0 & 6 & 0 & 48/7 \\ 0 & 0 & 7/3 & -8 \end{bmatrix}$

Thhe final augmented matrix shows a diagonal containing the numbers 1, 6, and 5/3. As you can probably infer, those are the coefficients for the unknowns in our linear system. We can go ahead and solve for the unknowns by dividing the values on the right-most columns by the coefficients of our unknowns (i.e., 48/7 will be divided by 6 and -8 will be divided by 7/3. First row does not need to be divided because the coefficient of the unknown is 1).

While writing out the equation, messed up a few times because I had incorrectly added/multiplied fractions. As you can probably tell, this would be where we would multiply a row by a constant, so that we can get nice round numbers to work with!

Like with linear systems, we can see if there are infinite or no solutions if the unknowns in the augmented matrix are 0 for an entire row.

No solution. 0 will never equal to 1. $\begin{bmatrix} 1 & 2 & 3 & 10 \\ 2 & 4 & 1 & -9 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Infinite solutions. Any value can be multiplied by 0 to equal 0. $\begin{bmatrix} 1 & 2 & 3 & 10 \\ 2 & 4 & 1 & -9 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Summary

Let's summarize everything we have gone through so far!

A linear system is a finite (limited) set of linear equations. Each linear equation has the potential to be a homogeneous linear equation (sum of unknowns = 0) or not (sum of unknowns = non-zero value). When we try to solve for a

linear system, we are trying to find the values for all of the unknowns, which we call solutions. For a linear system, there are three potential outcomes: the linear system has one solution, zero solutions, or infinitely many solutions. If the linear system has at least one solution, then it is consistent. If it has no solutions, then it is inconsistent. If a linear system has infinitely many solutions, we can't provide a specific point as a solution. In this scenario, we would rewrite our equation into a parametric equation, with our unknown replaced with a parameter (basically replacing it with a different letter(s)).

Outside of mathematics, the way linear systems are typically organized is in the form of an augmented matrix. When it comes to augmented matrix, we use elementary row operations to solve for the linear system. There are three operations we can perform: swap rows, multiply a row by a constant, multiply a row by a constant and add it to another row.

Practical Applications

Linear systems are a fundamental part of AI/ML. Essentially, all AI/ML models follow a similar formula:

$$W * x + b \tag{1.16}$$

Where W are the weights, b the biases, and x the inputs. Each of these values are vectors (which are just 1D matrices). ML models are trained with the expectation that the weights and biases found brings the model as **close** to the expected output as possible. In other words, we can think of training ML models as solving for the linear system $W * x + b$.

Extending it beyond simply training ML models, solving for linear systems are also used in a plethora of other ML use cases such as dimensionality reduction (PCA and SVD, where SVD is a Linear Algebra concept in-and-of itself)! Its also used in recommender systems as well, specifically matrix factorization (which usually involves linear systems). Some of these concepts (SVD and matrix factorization) we will explore later on! Just know that linear systems and solving for them are highly relevant in AI/ML!

1.0.2 Gaussian Elimination

Now that we have shown you HOW you can solve linear systems via augmented matrices, let's learn a bit more about these augmented matrices!

There are usually two forms for a solved augmented matrix: **row echelon form** and **reduced row echelon form**.

Reduced row echelon form would be the augmented matrix we are pretty accustomed to, with values only on the left-diagonal (not including the right-most columns).

$$\begin{bmatrix} 1 & 0 & 0 & 22 \\ 0 & 1 & 0 & 47 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

One thing you might have noticed from the augmented matrix above, is that the first nonzero number in each row is 1. These 1s are called **leading 1**s. They're called "leading 1"s because they are the first 1 in their row (from left to right). Leading 1s are a major property of reduced row echelon matrices. There are other properties that are used to identify reduced row echelon matrices:

1. All rows containing only zeroes are grouped together at the bottom of the matrix.
2. Every subsequent row has a leading 1 further to the right than the previous row (i.e., if the previous row has a leading 1 in column two, then the current row will have a leading 1 in column 3, if it has a leading 1).
3. Each column has only one leading 1. All other values in the column **MUST** be 0s.

Row echelon forms are a lot more lenient. Unlike reduced row echelon form, row echelon form can have multiple leading 1s in each column. I.e., the following augmented matrix is a valid row echelon form (but **NOT** a valid reduced row echelon form):

$$\begin{bmatrix} 1 & 0 & 0 & 22 \\ 1 & 1 & 0 & 47 \\ 0 & 2 & 1 & 1 \end{bmatrix}$$

Leading 1s are generally called **leading variables**. All other variables in the same row that are *not on the right most column and are not the leading variables*, are called **free variables**.

Now lets look at what a row echelon form matrix that has infinite solutions look like:

$$\begin{bmatrix} 1 & 20 & 0 & 22 \\ 1 & 1 & 23 & 47 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

As you can see, the equation has a row of just 0s! Now we can write each equation into a set of parametric equations!

$$\begin{aligned} x &= 22 - 20y \\ x &= 47 - y - 23z \end{aligned}$$

We can then replace y and z with any letters to represent them as parameters.

$$\begin{aligned} x &= 22 - 20t \\ x &= 47 - t - 23s \\ y &= t \\ z &= s \end{aligned}$$

Now we have a set of parametric equations! We can generally call a set of parametric equations as a **general solution**.

Okay...now that we have gotten some formal definitions out of the way, we can go over 2 types of **elimination procedures**. There are two ways in which we can solve a linear system using an augmented matrix, via **Gauss-Jordan Elimination** or **Gaussian Elimination**.

- **Gauss-Jordan Elimination**: Solves an augmented matrix by bringing it to reduced row echelon form. This type of elimination method uses both a *forward phase* and a *backward phase*.
- **Gaussian Elimination**: Solves an augmented matrix by bringing it to row echelon form. Typically uses a forward phase followed by *back-substitution*.

In that small bulletin, we introduced 3 new terminologies: forward phase, backward phase, and back-substitution. They may sound complex or new, but they are things you are very familiar with at this point!

- **forward phase**: You can think of this as solving the augmented matrix from top to bottom. The goal of this phase is to remove values below the leading variables.
- **backward phase**: This phase focuses on cleaning up our augmented matrix. In this phase, we move from bottom to top; we are essentially ensuring that, by the end of this phase, only the *left diagonal* contains nonzero numbers (excluding the right-most column, which will contain values).
- **back-substitution**: We learned about back-substitution when we took Algebra decades ago! Let's say you had the following equations: $x+y+z = 10$, $x + 20y + z = 3$, and $z = 10$. Since we know what z is, we can plug it into either of the other 2 equations to solve for y. We can then plug in what we found for y to solve for x! Basically, we are literally "*substituting*" backwards!

More Terminologies...Yay...

Now that we have a good idea on linear systems, augmented matrix, and two ways we might solve them, lets go over some additional terminologies: **homogeneous linear system**, **trivial solutions**, and **nontrivial solutions**.

If you remembered what a homogeneous equation is, you can think of a homogeneous linear system as a set of those homogeneous equations. For example:

$$\begin{bmatrix} 1 & 3 & 10 & 5 & 0 \\ 2 & 7 & 7 & 1 & 0 \\ 3 & 3 & 1 & 1 & 0 \end{bmatrix}$$

Would be a homogeneous linear system! As you can probably guess, its a homogeneous linear system because the constants (right-hand most values) are all 0s!

A **trivial solution** is a solution where all of the unknowns are 0. I.e., if we had a linear system that has the following unknowns (a, b, c, d), then the trivial solution would be $(0, 0, 0, 0)$; if you plugged those 0s into the homogeneous system, you'll end up with $0 = 0$. **Nontrivial solutions** are all other solutions (any solutions except for the trivial solution. I.e., any solution except for $(0, 0, 0, 0)$).

A homogeneous linear system either has 1 solution (the trivial solution) or infinitely many solutions (including the trivial solution). If you recalled, a "no solution" scenario would be when you solve for an unknown and end up with something like $0 = 5$, where 0 is on the unknown side and 5 is on the constant side. In this scenario, the solution would always be false since 0 is never equals to 5.

In a homogeneous linear system, where the constants are always 0, there is never a scenario in which we have no solutions (because the constants are all 0s).

Think about it, in what situation would we get no solution? When the only values that can be plugged into the unknowns are 0s and the constants are non-zeroes, right?

Let's look at an example!

$$0x + 0y + 0z = 10 \tag{1.17}$$

In the equation above, there is literally no situation where a solution for (x, y, z) would result in 10, right? Because anything multipled by 0 is...right! 0!

Now let's look at this equation:

$$0x + 0y + 0z = 0 \tag{1.18}$$

In the equation above, any value for (x, y, z) would result in $0 = 0$!

Recall that in situations where both trivial and nontrivial solutions exist (i.e., infinitely many solutions), we have to construct parametric equations. *When constructing the parametric equations for infinitely many solutions, we can ignore rows with all 0s.* This is because the row of 0s do not provide us with any additional info, because they do not provide any sort of constraint; 0s indicate that, at least for that row, any values can be plugged in for the unknowns and it would be good. This isn't useful in a parametric equation, because when we are creating parametric equations, we are doing so with the expectation that we can use these parametric equations to find potential unknown values that work in our linear system.

Let's look at an example! Say we had the following linear system:

$$\begin{aligned} 3x + 10y - 5z &= 10 \\ 2x + 2y + z &= -11 \\ 0x + 0y + 0z &= 0 \end{aligned} \tag{1.19}$$

The first two rows tells us that, "Whatever (x,y,z) values we find, it must be the case that when we plug in the (x,y,z) values into each equation, it must result in 10 = 10 and -11 = -11, respectively." However, the last row is telling us that, "Any values (x,y,z) will do! Because any values plugged into that equation will result in $0 = 0$"! In this scenario (and all scenarios where we have a row of 0s), the equations with all unknowns multiplied by a 0 coefficient are not useful!

Even More Terminologies...but Also Theorems!

In the last example we provided in the previous subsubsection, we can see that there is one row with all 0s. This row of 0s correspond to the free variable z, meaning z can take any value as long as z–which when plugged into the top two equations–satisfies the constraints outlined. In this case, since z corresponds to a row of 0s, it is classified as a free variable since it is not a leading 1.

There is a neat theorem that defines how we find the number of free variables in a homogeneous system:

Free Variable Theorem for Homogeneous Systems:
Given an augmented matrix that has n unknowns and r nonzero rows, the system has $n - r$ free variables.

This is just a mathematical way of saying, "If a row is all 0s, then one of the unknowns in that row is a free variable".

Let's look at an example. Let's say we had the following matrix with 4 unknowns and 2 nonzero rows, and let's say that the first 4 columns (the 4 unknowns) represent the variables a, b, c, and d.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In this case, we can tell that a and b are properly defined because the first two rows has a leading 1. The last two rows, and subsequently the last two variables c and d, are all zeroes! This means that c and d are free variables, since they can't be properly defined! Let's try to plug into the equation defined in the theorem:

$$n \text{ unknowns} = 4$$
$$r \text{ nonzero rows} = 2$$
$$n - r = 4 - 2 = 2$$

Theorem Two (it doesn't actually have a name so...):
A homogeneous linear system with more unknowns than equations has infinitely many solutions.

Really intuitive once you think about it. Homogeneous linear systems can have one of two outcomes: $0 = 0$ or numbers $= 0$ ¡- all cases has at least 1 solution regardless of how many unknowns we have. In situations where we have **more unknowns than equations**, there's no way for us to find the other unknowns without equations! Meaning we'll end up in an infinite solution anyways!

NOTE: Homogeneous systems are the only type of system that can never have no solutions. **Nonhomogeneous systems can still have no solutions** if there are more unknowns than there are leading 1s.

This is rather intuitive once you see an example.

$$\begin{bmatrix} 1 & 6 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In the example above, we have a nonhomogeneous system that has more unknowns than leading 1s and is inconsistent. Again, it's inconsistent (i.e., has no solution) because $0 = 1$ is never true. There might be other scenarios where a nonhomogeneous system has more unknowns than leading 1s while also being inconsistent, but that is the most obvious one.

HOWEVER! If a nonhomogeneous system has more unknowns than equations **AND** is consistent, then it has infinitely many solutions.

Here are two examples:

$$\begin{bmatrix} 2 & 3 & 4 & 5 & 6 \\ 1 & 1 & 1 & 1 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 2 & 1 & 3 & 4 & 11 & 6 \\ 0 & 0 & 0 & 0 & 12 & 20 & 5 \end{bmatrix}$$

In both scenarios, we will never reach a point where all unknowns are zeroed out. Instead, we'll always have an unknown available that we can use to solve! In these types of situations, infinite solutions are apparent.

Row Row Row Your...

When it comes to bringing a matrix to reduced row echelon form, you will always get the same reduced row echelon form. That is, regardless of the order you perform row operations in, they will all result in the same reduced row echelon form.

This is actually very intuitive, but very tedious to showcase (at least in demonstrating a full example). Given the augmented matrix below:

$$\begin{bmatrix} 1 & 2 & 3 & 6 \\ 2 & 2 & 2 & 2 \\ 1 & 3 & 3 & 1 \end{bmatrix}$$

We have a number of potential steps we can take. We can: swap row 1 with row 3, use row 1 to eliminate the first 2 to the left from row two, etc...

Despite the different values we may see while performing row operations, it will ultimately lead to the same reduced row echelon form:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -5 \\ 0 & 0 & -4 & -20 \end{bmatrix}$$

The reasoning behind this logic is simple: you are performing operations on rows, not individual values. Furthermore, reduced row echelon form contains rows with one leading 1 and all other unknowns as 0s. The form constraint and row-based operations ensure that reduced row echelon form for a matrix– regardless of the order of row operations–will result in the same reduced row echelon form matrix.

We can generalize this idea as the following:

All reduced row echelon forms of a matrix A have the same number of zero rows and all leading 1s are in the same positions.

The location of the leading 1s in each row are known as a **pivot position**. The column that contains the leading 1 is called a **pivot column**.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

So for matrix A, column 1 and column 2 are the **pivot columns of A** and the positions are the **pivot positons of A**.

Row echelon forms, on the other hand, are not unique. In other words, for a given matrix A, different sequences of row operations will lead to different row echelon forms; multiplying row 1 by a scalar and adding it to row 2 will lead to a different matrix than if we just add row 3 to row 2.

But...Why?

At this point, you might be wondering, "Okay...terminologies are cool and all but...Tommy, why do we have **TWO** different elimination procedures? Why don't we just stick to one?". To answer that, we must understand the pros and cons of the Gauss-Jordan and Gaussian Elimination.

There is an obvious reason why we might choose to use Gauss-Jordan over Gaussian Elimination: *It's super easy to see the answer*! Say you had the following augmented matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 7 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 5 & 10 \end{bmatrix}$$

If 1, 8, and 5 represented the coefficients of variables x, y, and z respectively, we can easily see what the values of x, y, and z are simply by dividing the values in the right column by the values in the same row (7 / 1, 0 / 8, and 10 / 5)! Plus, performing forward phase + backward phase is easier than performing forward phase + back-substitution!

...but that only applies for small matrices. Once we get to larger ones– especially ones that are large enough to require computational power–this process becomes arduous. Not only that, but if we look at the way computations are performed in the Gauss-Jordan elimination procedure, we will ultimately realize that Gaussian-Jordan elimination requires more computational power and operations than Gaussian elimination (roughly 50% more)!

To further emphasize, we are iterating through the augmented matrix **TWICE**, while performing multiplications, additions, subtractions on **ALL** rows and variables. This results in essentially 50% more operations since we are performing row operations while moving forward and backwards.

In the realm of AI, this could mean significantly slower inference or training time due to the forward phase + backward phase pipeline in the Gauss-Jordan elimination procedure. Even worse than slow inference time or training time, this 50% more operation will make the results more error-prone.

Errors...Errors Everywhere!

In computers, every operation performed has the potential for rounding errors because computers typically approximate values. These rounding errors could be dangerous, as slight rounding can lead to severely incorrect outputs (in Python, this might mean rounding a very small value to 0 or a very large value to NaN or inf). So as you may have guessed, the more operations that are performed, the more likely it is that we will run into these types of rounding issues. In safety critical systems like aircrafts, rounding can lead to devastating consequences as many operations require precision. Furthermore, the more operations that are performed, the more the number will round until it eventually reaches an unusable value (NaN, inf, -inf, etc). In the world of AI, this would

mean invalid results that throw the model off completely either during training or inference.

These types of rounding issues have been a major debate when converting mathematical concepts/theories into practice, as it brings up a rather philosophical argument many mathematicians (and students) have: Do we really need a long and tedious sequence of complex mathematical operations? This ultimately lead to disparity between mathematical theory and practical implementations. In this sense, algorithms that perform a lot of complex mathematical operations which may ultimately result in rounding errors or others are labeled as **unstable algorithms**. These types of algorithms are considered unstable because they produce **roundoff** errors (where values are rounded).

All in all, Gaussian Elimination seems to be the better choice in solving for larger augmented matrices as it reduces computations leading to: faster inference time and lower roundoff error rates.

Summary

This chapter covers a lot of terminologies, but fundamentally it covers 2 ways of solving for augmented matrices: Gaussian Elimination and Gauss-Jordan Elimination. Both have their pros and cons.

Gauss-Jordan: It is a lot easier for us to understand and solve for, as it directly brings our augmented matrix to reduced row echelon form. This results in that nicely formed matrix where the main diagonal contains non-zero values and all other locations contain 0, effectively providing us with the unknowns in a clean and visible format!

Gaussian Elimination: Gaussian elimination is what we probably were introduced to in middle school/high school but in a non-augmented matrix format. We use the elimination method to remove some variables in some of our equations, solve for those simplified equations, and plug the values we found into the remaining equations so that we can solve for the remaining unknowns. This method can be easier for some, but the solution isn't nearly as obvious as in Gauss-Jordan elimination.

You might have noticed that my description of Gaussian Elimination doesn't really provide any benefit, and that's because Gaussian Elimination works best for very large matrices. This is because Gaussian Elimination requires 50% less work than Gauss-Jordan elimination, resulting in a faster problem solving process and reducing potential roundoff errors in computers! For a more detailed discuss on the tradeoffs between the two, I would read through the previous two subsubsections.

There are obviously other terminologies that are discussed in this chapter (alongside some neat theorems), but those aren't really necessary in understanding the overarching theme of the chapter, which is: Which method should I use for solving linear systems?

Practical Applications

The practical application of this chapter is closely tied to the practical application discussed in chapter 1; it is used almost everywhere in AI! Once again, linear systems are a fundamental aspect of AI models such as linear regression, SVC models, and the neurons in a neural network!

Given an input I, weights W, and biases B, our AI model is fundamentally trained to solve for the following equation:

$$Y = W * I + B \tag{1.20}$$

Where Y is our predicted output. Of course, this is stripped out some additional complex operations such as activation functions or gradients, but this is fundamentally it! At the core, we are finding the weights W and biases B that will lead us closer to the correct Y.

This chapter focuses on the practicality of Gaussian Elimination vs Gauss-Jordan elimination (as well as discussing how some mathematical theorems/processes are so complex that it becomes impractical in practice, which is a neat acknowledgment to some of our fears and complaints)!

We can conclude from this chapter that Gaussian Elimination should always be used in practical applications that operate on large matrices in order to reduce potential errors! So if you're ever planning on building an AI model from scratch without any libraries, you should keep this in mind! :))

1.0.3 Matrices & Matrix Operations

In previous sections, we introduced matrices as a way of solving for linear systems (*augmented matrices*). However, any table containing numbers is considered a "matrices" as well. As such, we can give the matrix a general definition:

A **matrix** is any rectangular array of numbers. Each number in this matrix is called an **entry** of that matrix.

This broad definition means that a matrix can be applied to any scenario, not just linear systems. Have a table for the number of hours you played on a particular video game? Matrix. Have a financial spreadsheet containing spending and earnings? Matrix. And so on...

Point is, anything is a matrix if you imagine hard enough! :D

When it comes to matrices, the dimension of the matrix is very important (we'll see why later on). The dimension is often called the **size** of a matrix. Typically, the size of the matrix is written in the following notation: *num of rows* $\boldsymbol{x}$ *num of columns*. For example, a matrix with 4 rows and 6 columns would have a size of 4 x 6.

When part of the matrix size has a value of 1, that means that the matrix is either a **row matrix** or a **column matrix**. You can think of them as a horizontal or vertical collection of elements, respectively. They are also called **row vectors** and **column vectors** as well.

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

A column matrix with size 3 x 1

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

A row matrix with size 1 x 3

All you really need to know is that vectors are 1D matrices, but are often referred to as vectors in papers. It is still worthwhile to know that vectors can sometimes be referred to as matrices!

In general, when the size of the matrix matters, we typically draw out the matrix as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1n} \\ . & & & & \\ . & & & & \\ . & & & & \\ a_{m1} & a_{m2} & a_{m3} & ... & a_{mn} \end{bmatrix}$$

When size really doesn't matter, we can shrink the notation significantly as follows: $[a_{ij}]$ or $[a_{ij}]_{mxn}$ where i and j refers to any index in row and column, respectively. Sometimes (often in AI papers), matrix notations can be simplified even further: A_{ij}, where A is the matrix name and i and j are indices.

For vectors (i.e., row and column matrices), there is a special notation. The matrix/vector is typically a lowercase and bolded letter, for example: $\mathbf{b}$.

For square matrices (matrices whose row and column size are the same), are typically denoted as *square matrix of order n*.

Lastly, the **main diagonal** is the elements in the matrix going from the top left to bottom right.

center $\begin{bmatrix} a_{11} & ... & ... & ... & ... \\ ... & a_{22} & ... & ... & ... \\ ... & ... & a_{33} & ... & ... \\ ... & ... & ... & a_{44} & ... \\ & & & & \end{bmatrix}$

The main diagonal may seem like an unnecessary definition, but it's actually quite useful (as we will see later on)!

Matrix Operations

I think this section will be the most succinct. Let's quickly go over some operations:

1. "=": Matrices are equal if they have the same size (same number of rows and columns) and the same entries. In other word, they are exactly the same.

2. "+" and "-": Addition and subtractions are performed element-wise. This means that when you add, you are adding elements of matrix A to elements of matrix B (and the same applies to subtraction).

    - **NOTE**: Addition/subtraction can only be performed on **matrices with the same size**.
    - There are a few ways addition/subtraction matrix operation may be notated in papers: $(A + B)_{ij} = (A)_{ij} + (B)_{ij} = a_{ij} + b_{ij}$ (where $A = [a_{ij}]$ and $B = [b_{ij}]$). The type of annotation is important in determining what is focused on, but typically annotation isn't really important in academic papers (CS papers).
    - If the matrices are different sizes, then adding/subtracting is undefined. Makes sense, since you'll have some elements that can't be added if the two matrices are of different sizes.

3. scalar product: If you are multiplying a matrix by a single number, you are performing scalar multiplication (or trying to find the scalar product). You are basically multiplying that single number by every element within the matrix.

    - Given a scalar c and a matrix A, the product cA is a scalar multiple of A. Just like how $3 * 5 = 15$ is a scalar multiple of 5.
    - When multiplying a matrix by a negative value, it is typical to write $(-c) * B$ as $-cB$. For example, $(-1) * B$ would be $-B$.

4. Matrix A * Matrix B: We are performing the *dot product* of the rows of matrix A by the columns of matrix B. For example, given the following matrices:

$$\mathrm{B} = \begin{bmatrix} 2 & 6 & 4 \\ 3 & 4 & 1 \end{bmatrix}$$

$$\mathrm{A} = \begin{bmatrix} 1 & 9 \\ 3 & 2 \\ 4 & 4 \end{bmatrix}$$

    The first operation in $A * B$ would be performed on the first row of A and the first column of B: $1 * 2 + 9 * 3$ (then $1 * 6 + 9 * 4$, $1 * 4 + 9 * 1$, etc for subsequent rows). Each row in A will be distributed/multiplied by every column in B. For a more abstract summary: **We are multiplying the rows of our first matrix (A) with every columns in our second matrix (B), and adding up those values that we get from the row-by-column operation**. Since this is a matrix operation, the final output won't be a single value, but a matrix of values. For example, if we multiplied all of our rows by every columns in the multiplication example we gave, the result of our matrix multiplication would be

$$A * B = \begin{bmatrix} 28 & 42 & 13 \\ 12 & 26 & 14 \\ 20 & 40 & 20 \end{bmatrix}$$

You can think of this as the dot product between the rows of A by the columns of B.

- The neat thing about this operation is that we know the dimensions of our output matrix. Given matrix A of size **w** x r and matrix B of size r x **s**, the dimension of output matrix AxB would be w x s.
- We can justify the final dimensions in simple terms: when we multiply row 1 of matrix A with all columns of B, we are getting the first row of matrix AxB.

$$\text{B} = \begin{bmatrix} \mathbf{2} & 6 & 4 \\ \mathbf{3} & 4 & 1 \end{bmatrix}$$

$$\text{A} = \begin{bmatrix} \mathbf{1} & \mathbf{9} \\ 3 & 2 \\ 4 & 4 \end{bmatrix}$$

$$A * B = \begin{bmatrix} \mathbf{28} & 42 & 13 \\ 12 & 26 & 14 \\ 20 & 40 & 20 \end{bmatrix}$$

$$\text{B} = \begin{bmatrix} 2 & \mathbf{6} & 4 \\ 3 & \mathbf{4} & 1 \end{bmatrix}$$

$$\text{A} = \begin{bmatrix} \mathbf{1} & \mathbf{9} \\ 3 & 2 \\ 4 & 4 \end{bmatrix}$$

$$A * B = \begin{bmatrix} 28 & \mathbf{42} & 13 \\ 12 & 26 & 14 \\ 20 & 40 & 20 \end{bmatrix}$$

  As you can probably tell, multiplying row 1 of matrix A by all columns of matrix B will give us our first row! From there, we can see that once row 1 of A is multiplied by all columns of B, AxB now has **3 columns**! From there, we can extrapolate and assume the same thing will happen once we multiple all rows of A by all columns of B (i.e., we will end up with 3 rows, because there are 3 rows in A)!
- If we are multiply A by B, the number of columns in A must match the number of rows in B. We can think about this logically, if we are multiplying the rows of A by the columns of B, then the number of elements we are multiplying must be the same, otherwise there will be extra elements!

$$\text{B} = \begin{bmatrix} 2 & \mathbf{6} & 4 \\ 3 & \mathbf{4} & 1 \end{bmatrix}$$
$$\text{A} = \begin{bmatrix} \mathbf{1} & \mathbf{9} & \mathbf{10} \\ 3 & 2 & \\ 4 & 4 & \end{bmatrix}$$

$$6 * 1 + 4 * 9 + \mathbf{10 \text{ (10 is left over!!)}} \tag{1.21}$$

This wouldn't work since all elements must be multiplied by another element in matrix multiplication! So before you perform any matrix multiplication, check to see if the columns of the first matrix matches the rows of the second! If A was of size a x r and B was of size r x f, since the columns of A is r, and the rows of B is r, matrix multiplication is possible!

5. Row-Column Rule: Matrix multiplication fundamentally follows a row-column rule; multiply the rows of one matrix by the columns of another. This rule is particularly useful insituations where we want to **partition** the product of two matrices.

6. partitioning: Partitioning is just a fancy way of saying, "let's cut up a matrix into smaller matrices/vectors". For example, you can cut a matrix A of shape 4 x 3 into 4 individual row vectors or 3 individual column vectors. We can also cut the matrix into being 1 matrix of shape 2 x 3 as well.

7. Submatrices: a submatrix is a partitioned matrix of a larger matrix. 2 x 3 would be a submatrix of our larger 4 x 3 matrix in the example above!

8. Linear combinations: Linear combinations is matrix multiplication between a matrix A and a column vector x (which is effectively a m by 1 matrix), where x contains scalars.

   - For example, if x containing the elements
   $$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$
   and A is
   $$\begin{bmatrix} 5 & 4 & 3 \\ 2 & 3 & 5 \end{bmatrix}$$
   We can think of Ax as a linear combination between A and x:
   $$1 * \begin{bmatrix} 5 \\ 2 \end{bmatrix} + 2 * \begin{bmatrix} 4 \\ 3 \end{bmatrix} + 3 * \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$
   - In the example above, A would be our **coefficient matrix**, since it holds the coefficients for our own (**2**x + **3**y +...). Distributing vector x into matrix A means multiplying each element of x with the corresponding column of A, based on its position.

- If we were to generalize the equation above by using their letter representation, we will get: $Ax = b$. Notice anything? Correct! We're very familiar with this equation!

9. column-row expansion: As the name implies, we are splitting matrix A into columns and matrix B into rows, for the matrix multiplication operation AB. This is an easier way of seeing how the values are distributed.

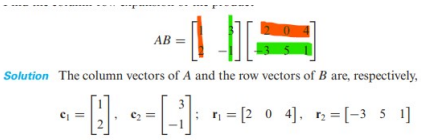

Figure 1.4: Distributing the rows of A into the columns of B means each element in a column of A is multiplied by all elements in the corresponding row of B.

10. Transpose: The transpose of a matrix A, denoted as A^T, is the reflection of A across its main diagonal; The columns of A becomes the rows of A^T and the rows of A becomes the columns of A^T.

11. Trace: The trace of matrix A, denoted as $tr(A)$, is the sum of the main diagonal of A. The trace only exists if A is a square matrix. If A is not a square matrix, the trace does not exist.

Interesting History

One of the interesting things about operations is the history of why some of these operations were produced! Back in the 1800s, a German mathematician named "Gotthold Eisenstein" had introduced matrix multiplication as a way of simplifying substitution in linear systems. His mentor, Gauss (yes, THE Gauss), recognized Eisenstein as being equal to Isaac Newton and Archimedes, but Eisenstein sadly passed at the age of 30 due to poor health.

1.0.4 Summary

Matrix operations are typically performed on a row-by-column or element-by-element basis. Because these operations are matrix operations, there are certain criteria that must be met in order for these operations to be performable.

Long story short, many matrix operations require the matrices to be of the same size (when adding or subtracting two matrices) or for the size of a column of one matrix be the same size as the row of another (multiplication between two matrices).

There are some operations that don't really have restrictions, like scalar product, transpose, partitioning, row-column expansion, etc, but those typically

are single number-to-matrix operations or an operation only performed on 1 matrix.

Why Is Matrix Multiplication The Way It Is?

If you've ever wondered why we perform matrix multiplication the way we do (multiple the rows of one matrix by the column of another matrix), then I've got you covered.

Although many Linear Algebra books cover this, they don't cover it in a way that is obvious to understand so I will try and express it in a digestible way.

Let us say that we had a ton of equations in the form of $2x+3y+4z+...$ (where the numbers can vary across equations). Let's also say that we somehow found multiple values for $x, y, z, ...$ after solving all of these equations. The question now is, "how do we plug them in easily?". We could plug in all values for $x, y, z, ...$ manually into each equation, but that would be very tedious. Further, trying out all of these equations manually would be tedious as well...so how can we solve this while also ensuring our equations, numbers, etc are as compact as possible?...matrices! Let's try forming our equations and values for $x, y, z, ...$ into separate matrices.

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

The above equations have their variables stripped and are restricted to x, y, and z for convenience. As you can tell, we are keeping our equations **horizontally**, this is just standard practice (because writing equations vertically hurts to look at :( ).

In a similar vein, we would write the values we found for x, y, and z as **vertical columns**, because it is standard in mathematics:

$$\begin{bmatrix} 1 & 3 & 3 \\ 2 & 1 & 1 \\ 5 & 5 & 2 \end{bmatrix}$$

Where 1, 2, and 5 of the first column are x, y, and z respectively. So when we are computing row-by-column multiplication, we are effectively plugging in our x, y, and z values into our equations.

But sure, this does somewhat answer the "why" matrix multiplication is performed on a row-by-column basis, but "standard practice" isn't really sufficient of an explanation.

Justifiable...Justification

Searching the Internet, I was able to find a legitimately good justification for matrix multiplication: row-by-column let us calculate the values and **find the position in the new matrix to place the values**. For example, multiplying the **first** row of matrix A by the **first** column of matrix B tells us that in the new matrix, the newly calculated value should be in **the first row and first column of matrix** AB.

This is a more intuitive approach to find the location to place the new value, rather than placing it in an arbitrary location. It also allows us to reverse engineer the matrix AB ("oh, the top left element is probably the multiplication of the first row of A by the first column of B").

There is another fundamental issue with comes with row-by-row matrix multiplication though. When we perform matrix multiplication on a row-by-row basis, many of the useful properties we have, break:

- Associative: When multiplying 3 or more numbers (matrices, vectors, etc), the way we group them (using parenthesis) should result in the same solution. For example $(A*B)*c$ should give the same answer as $A*(B*c)$.
- Identity Matrix: A matrix whose main diagonal has 1s, but all other locations are 0s. When multiplied by another matrix B, we expect the result to be B; identity matrix is effectively the "1" in matrix multiplication (any matrix multiplied by the identity matrix will be itself).
- Not Commutative: $A * B$ is not the same as $B * A$. The row-by-column operation ensures that this is true.

Simply saying "these properties are important" is easy, but that's not a sufficient answer for individuals who want to know "why" these properties are important.

Importance of Properties

Questions to answer/look into:

- WHY is associative important?
- Provide evidence that identity matrix is broken. Explain why identity matrix is crucial.
- Why is commutative important? What purpose does it serve?
- ~~matrix~~
- ~~entry~~
- ~~size~~
- ~~row vector/row matrix~~
- ~~column vector/column matrix~~
- ~~scalars~~
- ~~compact notation for matrices~~
  - ~~Different notation for when size is important vs when size isn't.~~

- Sometimes an entry is denoted by the matrix name followed by the index like: $(A)_{ij}$. Where A is the matrix.
- row/column matrices/vectors have a special notation: lowercase and bold.
- "Square matrix of order n" to denote a square matrix of size n x n.
- main diagonal
- NEW SECTION ON ARITHMETIC
- "equal" matrices are matrices that have the same size and same entries
- "adding" and "subtracting" matrices are element-wise operations. This means that adding and subtracting are performed on each element in the matrices. NOTE: adding and subtracting can only be performed on SAME SIZED matrices, different sizes cannot be added/subtracted from one another.
- Cover the different ways we can write out the notation for addition/subtraction (might be useful when writing academic papers). Use page 42 in the Linear Algebra PDF.
- Addition/Subtraction of matrices with different sizes are undefined.
- If c is a scalar and A is a matrix, then $c * A$ is a scalar **product**, where each element in A is multipled by the scalar c.
- cA is a **scalar multiple** of A.
- It is common notation to write $(-1) * B$ as $-B$.
- Explain how the **product of two matrices** was found through experience by mathmaticians.
  - If A s mxn and B is nxw, then the matrix AB will be of size mxw
  - We are multiplying the rows of A by the columns of B, and summing up the values we get from that row-by-column operation. ¡- this is basically the dot product, which would have been useful if explained earlier.
  - A nice thing about the product of matrices is that we know the dimensions of the resulting matrix, which is just the rows of A and columns of B: m x n, n x w
  - NOTE: The inner the number of columns of A **MUST** match the number of rows of B. If not, we will be missing elements during our multiplication, which shouldn't happen and the product will be undefined.
  - For example, if A was of size r x c and B was of size r x f, we wouldn't be able to perform multiplication between these two matrices.

- ~~row-column rule for matrix multiplication~~
- ~~partitioned - partitioning is useful when you want to see the product of AB for a specific row in AB, specific column in AB, specific set of rows in AB, etc without computing the entire product.~~
- ~~submatrices~~
- ~~linear combinations + coefficients - a different way of thinking about matrix multiplication (can just think of it as a sum of products between matrices A_1, A_2, ... with constants C_1, C_2, ...)~~
- ~~Proves a theorem: If A is a matrix of size m x n, and x is a column vector of size n x 1, then we can express the product between them as a linear combination of the rows in A with the entries in x.~~
- ~~column-row expansion of AB or product of any two matrices.~~
- ~~Ax = b~~
- ~~coefficient matrix~~
- ~~Transpose of a matrix - Denoted as A^T given matrix A. The rows of A become the columns of A^T and the columns A become the rows of A^T~~
  - ~~You can think of this as a matrix "reflect" along its main diagonal.~~
- ~~trace - trace of A (denoted as tr(A)) is the sum of the main diagonal of A.~~
  - ~~Only exists if A is a square matrix. If A is not a square matrix, then the trace is undefined.~~

1.0.5 Practical Applications

Matrices and matrix operations are the foundations of any large-scale numerical operations performed by machines. If you have a machine that is performing operations on large quantities of data, it is likely performing some sort of matrix operations. In the world of programming, this is the equivalent of performing operations on large arrays (1D and beyond)! In the world of AI/ML, this is essentially every operation performed by a model (training, calculating weights, backpropagation, etc).