Lecture Notes for MIT Course 10.34: Numerical Methods in Chemical Engineering

Instructors: Prof. Connor Coley and Prof. Sungho Shin

Notes by: Matthew Cox

Fall 2024

Department of Chemical Engineering
Massachusetts Institute of Technology

Contents

0 Introduction

This document is based on the Fall 2024 offering of 10.34 (Numerical Methods Applied to Chemical Engineering) at MIT taught by Profs. Sungho Shin and Connor Coley; it is essentially a transcription of the course slides/lectures into document form. In fact, this document began as Matthew Cox's notes for the course. This book should be fairly self-contained, but at times it may be helpful to refer to external resources for more in-depth explanations.

Warning: This document is still a work in progress, and is not yet completely polished. It is recommended to always refer to the course slides/lectures for the most up-to-date information. If you spot any errors, feel free to email Matthew at mcox340@mit.edu (or directly submit an issue/PR on the corresponding GitHub repository if you know how to use GitHub).

1 Computational Preliminaries

1.1 Numerical Error

Most computer problem solving is done approximately (numerically) and not analytically/symbolically, either because no analytic solution exists or because the analytic solution is too complex to compute. Consequently, **numerical error** is introduced into our calculations. Mathematically, we can define numerical error as absolute or relative error:

$$\epsilon_{\text{abs}} = |x_{\text{exact}} - x_{\text{approx}}| \quad \text{(absolute error)}$$
$$\epsilon_{\text{rel}} = \frac{|x_{\text{exact}} - x_{\text{approx}}|}{|x_{\text{exact}}|} \quad \text{(relative error)}$$

Finite-Precision Arithmetic One major source of numerical error comes from the fact that numbers are represented with "floating point" or "finite precision." In other words, we can only represent a finite number of digits in a computer. The most common floating-point formats are single-precision (float32) and double-precision (float64), which represent numbers with 32 and 64 bits, respectively (Figure 1.1). The number of bits determines the precision of the number. For example, float32 has 23 explicitly stored bits of precision, which is roughly equivalent to 7 decimal digits. Meanwhile, float64 has 52 explicitly stored bits of precision, which is roughly equivalent to 16 decimal digits. The precise details of how numbers are represented in floating-point format are not important for this course, but it is important to understand that there is a tradeoff between precision and storage.

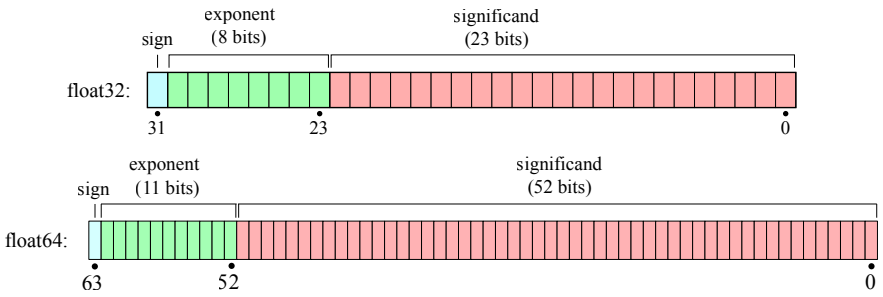


Figure 1.1: Floating-point representation of numbers.

IEEE-754 Bit Layout We consider this briefly for concreteness' sake; the IEEE-754 standard is a very common format for representing floating-point numbers in computer hardware. For a floating-point number in this format, bits are divided into sign (s), exponent (e), and fraction/significand (f). For normal numbers (i.e., not zero, NaN, infinity, or subnormal),

$$x = (-1)^s \times (1.f)_2 \times 2^{e-\text{bias}}$$

where the notation $(1.f)_2$ means the binary representation of the fraction f with a leading 1 (implicit).In float32, $s = 1$ bit, $e = 8$ bits (bias 127), and $f = 23$ bits. In float64, $s = 1$, $e = 11$ bits (bias 1023), and $f = 52$ bits. The bit indices for float32 are 31 (sign) | 30–23 (exponent) | 22–0 (fraction) (Figure 1.1). As an example (float32),

$$\underbrace{0}_{s}\,\underbrace{01111100}_{e}\,\underbrace{01000000000000000000000}_{f} \implies (-1)^0 \times (1.01)_2 \times 2^{-3} = 1 \times 1.25 \times \frac{1}{8} = 0.15625$$

Binary expansions of most real numbers are non-terminating and must be truncated, e.g.,

$$\pi = 1.10010010000011111101101010\ldots \times 2^1$$

so the significand is truncated, introducing representation error.

Precision and Hardware There is a practical tradeoff between accuracy and speed/memory. On most CPUs, double precision (float64) is the default and well optimized (e.g., MATLAB uses float64 by default). Many GPUs deliver highest throughput with float32 or even float16; low-precision arithmetic is common in machine learning. Choose the lowest precision that maintains the needed accuracy, and be mindful of mixed-precision casts when moving data between CPU and GPU. (In 10.34, this is not a significant consideration, but it is important to be aware of this fact.)

Round-off and Machine Epsilon One specific type of numerical error arising from finite precision is round-off error. Every time an arithmetic operation is performed, the result must be rounded to the nearest representable number. Machine epsilon (ϵ_{mach}) is an upper bound on the relative error due to rounding:

$$\epsilon_{\text{mach}} = \begin{cases} 1.192 \cdot 10^{-7} & \text{for float32} \\ 2.2204 \cdot 10^{-16} & \text{for float64} \end{cases}$$

Simply put, ϵ_{mach} is the difference between 1 and the next largest floating point number for a given representation. For example, if we were to check the condition $1 + \epsilon_{\text{mach}}/2 \stackrel{?}{=} 1$ using single- or double-precision in some programming language, we would be told that indeed $1 + \epsilon_{\text{mach}}/2 = 1$, which is obviously not correct mathematically. However, adding anything larger than ϵ_{mach} to 1 will give the expected result.

Ordering Calculations Because floating-point numbers are only approximations, the way we arrange computations can create or avoid error. The most common manifestation of this is known as "catastrophic cancellation": subtracting nearly equal quantities wipes out correct digits and magnifies rounding noise. For example, the algebraically equivalent expressions $1 - 3(4/3 - 1)$ and $1 - (3/3 \cdot 4 - 3)$ can result in 2.22×10^{-16} versus 0 in double precision; this is because the first forces a small difference subtraction while the second avoids it. Similar issues show up when computing $1 - \cos x$ for small x (use the fact that $1 - \cos x \approx 2\sin^2(x/2)$ at small x to compute this instead), summing terms with very different magnitudes (sum smaller terms first or use compensated summation), or applying the quadratic formula when $b^2 \gg 4ac$ (use $2c/(-b \mp \sqrt{b^2 - 4ac})$ for the small root). The practical rule you should know for 10.34 is simple: choose algebraic forms that avoid subtracting nearly equal numbers, keep intermediate values on similar scales (rescale when helpful), and use numerically stable library routines (i.e., no need to reinvent the wheel if MATLAB has something you need). Otherwise, tiny rounding errors can accumulate over many operations into visible inaccuracies.

Overflow and Underflow Overflow/underflow describes the situation where numerical calculation exceeds the limits of floating-point formats. It occurs when we try to store numbers that are too large (overflow) or too small (underflow) for our chosen representation. The finite precision and finite exponent limit the range: approximately $10^{-38} \leq |x| \leq 10^{38}$ for float32 and approximately $10^{-308} \leq |x| \leq 10^{308}$ for float64. Thus, we may need to rescale our numbers to avoid overflow/underflow. Generally, though, this is not something we need to worry much about.

> **Warning: Intermediate Steps Can Overflow!** An expression can overflow during an intermediate step even if the final answer is perfectly representable.
>
> For example, calculating the hypotenuse $\sqrt{a^2 + b^2}$ with $a = 10^{200}$ and $b = 10^{200}$ will fail because a^2 overflows. A robust function like MATLAB's `hypot(A, B)` avoids this by rescaling first.

Truncation Error and Convergence Even with infinite precision, numerical errors arise due to the approximate nature of many algorithms. Iterative methods and series expansions approximate mathematical functions, introducing errors that depend on the number of terms or iterations used. To illustrate this, let's compare two different algorithms for approximating the same mathematical constant: π. The Leibniz series is a simple infinite series that approximates π as

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

This series converges very slowly. To achieve just two decimal places of accuracy, we need to sum over several hundred terms. The slow convergence means more iterations are required for high precision, increasing

computational effort and the potential for cumulative numerical errors. In contrast, we can consider one of Ramanujan's series for π, which converges extremely rapidly:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!(1103 + 26390n)}{(n!)^4 396^{4n}}$$

This series is remarkable because each term adds approximately eight additional correct decimal places to the approximation of π. Therefore, summing just a few terms yields a highly accurate value. The convergence of these two series to the true value of π is shown in Figure 1.2. The Ramanujan series converges to within machine epsilon after just 3 terms!

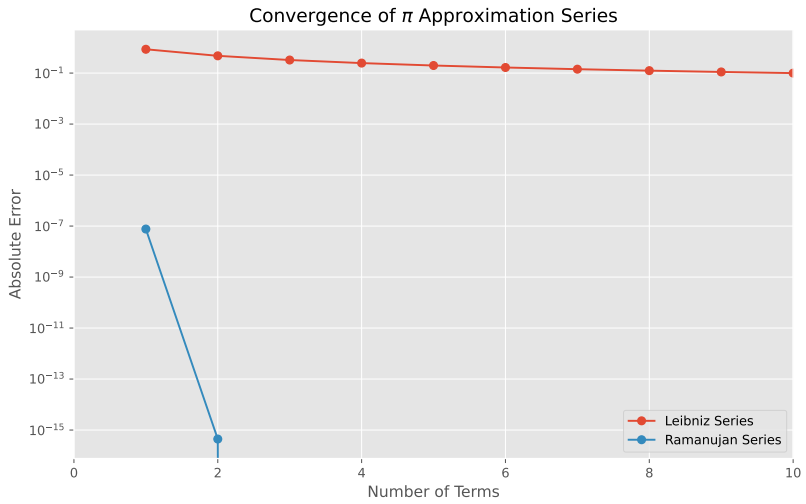


Figure 1.2: Convergence of two series approximating π. Machine epsilon is approximately $2.22 \cdot 10^{-16}$ here, so values within this range are essentially as converged as numerically possible.

1.2 Computational Complexity

As suggested earlier when discussing numerical error, we need to understand not just how to solve a problem, but also how efficiently it can be solved as the problem size increases. This efficiency is quantified using the concept of **computational complexity**. Specifically, we focus on **time complexity**, which measures the number of elementary operations an algorithm requires relative to the size of its input (the size of the input is typically denoted by N).

To express computational complexity, we generally use **big-O notation**. This mathematical notation describes an upper bound on the growth rate of a function so that we can focus on the most significant factors that affect an algorithm's performance as the input size becomes large. Mathematically, we say that a function $f(N)$ is $O(g(N))$ if there exist positive constants c and N_0 such that

$$|f(N)| \leq c \cdot |g(N)| \quad \text{for all } N \geq N_0$$

This definition means that for sufficiently large values of N, the function $f(N)$ does not grow faster than a constant multiple of $g(N)$. Again, we stress that big-O notation captures the dominant term of $f(N)$ as N approaches infinity, ignoring lower-order terms and constant factors that become insignificant at large scales.

Example: We consider two algorithms designed to solve the same problem with input size N. Algorithm A has a time complexity of $f_A(N) = 5N^2 + 10000$. Algorithm B has a time complexity of $f_B(N) =$

$2N^3 + N$. We can express the time complexity of each algorithm using big-O notation:

$$f_A(N) = O(N^2)$$
$$f_B(N) = O(N^3)$$

To see the practical impact, let's compare the number of operations required as N increases:

Problem Size (N)	**Operations in Algorithm A**	**Operations in Algorithm B**
10	$1.05 \cdot 10^4$	$2.01 \cdot 10^3$
100	$6 \cdot 10^4$	$2.0001 \cdot 10^6$
1000	$5.01 \cdot 10^6$	$2.000001 \cdot 10^9$
10000	$5.000001 \cdot 10^{10}$	$2.0000000001 \cdot 10^{15}$

As the table shows, Algorithm B requires significantly more operations than Algorithm A as N grows (but notice this is NOT true at small N!), making it less practical for large problem sizes. This also follows directly from the big O notation, which tells us that $f_B(N)$ grows faster than $f_A(N)$.

To be even more concrete, let's assume that each operation takes 10^{-8} seconds, which is reasonable for a modern CPU. Then, we can construct the table below:

Problem Size (N)	**Algorithm A Runtime**	**Algorithm B Runtime**
10	0.105 ms	0.0201 ms
100	0.6 ms	20.001 ms
1000	50.1 ms	20 s
10000	8.33 min	231 days

The above example shows how algorithms with lower computational complexity dramatically reduce computation time, especially for large N. The difference between $O(N^3)$ and $O(N^2)$ algorithms (and more generally, any two algorithms with different associated time complexities) can be the difference between a project being feasible or not.

Finally, we briefly mention the concept of **space complexity**, which measures the amount of memory an algorithm requires relative to the size of its input. Space complexity is also expressed using big O notation, but we will not discuss it further in this course. However, it is important to remember that memory usage can be just as important as time complexity, especially when working with large datasets or on memory-constrained devices.

2 Linear Algebra (Lectures 1-6)

We will first review some of the main results of linear algebra that will be useful for 10.34. Then, we will move on to some of the applications of linear algebra to numerical methods, and how to numerically solve linear systems. Feel free to skip directly to subsection 2.6 if you are comfortable with the basics of linear algebra.

2.1 Scalars and Fields

Scalars are elements of a field, such as the real numbers $\mathbb{R}$ or complex numbers $\mathbb{C}$. The field $\mathbb{C}$ is an extension of $\mathbb{R}$ using the imaginary unit i, defined by $i^2 = -1$.

Any complex number $z \in \mathbb{C}$ can be written as

$$z = a + bi, \quad a, b \in \mathbb{R}$$

where $a = \text{Re}(z)$ and $b = \text{Im}(z)$. The **complex conjugate** of z is

$$\overline{z} = a - bi$$

The **magnitude** (or modulus) of z is

$$|z| = \sqrt{a^2 + b^2} = \sqrt{z\overline{z}}$$

Example: For $z = 3 + 4i$, the conjugate is $\overline{z} = 3 - 4i$, and the magnitude is $|z| = \sqrt{3^2 + 4^2} = 5$.

We can visualize complex numbers as points in the complex plane, with the real part on the horizontal axis and the imaginary part on the vertical axis (Figure 2.1).

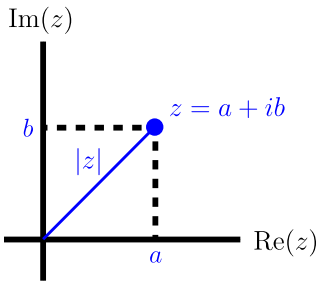


Figure 2.1: The complex plane.

2.2 Vectors and Vector Spaces

A **vector** is an ordered list of scalars; it has magnitude and direction. In n-dimensional space, a vector $\mathbf{v}$ over the field $\mathbb{R}$ or $\mathbb{C}$ is written as

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \quad v_i \in \mathbb{R} \text{ or } \mathbb{C}$$

It is also common to write this as $\mathbf{v} = (v_1, v_2, \ldots, v_n)$. The set of all such vectors forms a **vector space** $\mathbb{R}^n$ or $\mathbb{C}^n$, equipped with two operations: vector addition and scalar multiplication.

2.2.1 Vector Addition

Given two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ (or $\mathbb{C}^n$), their sum is defined component-wise:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 + v_1 \\ u_2 + v_2 \\ \vdots \\ u_n + v_n \end{bmatrix}$$

This operation is commutative and associative. Geometrically, vector addition corresponds to a "tip-to-tail" construction (see Figure 2.2).

2.2.2 Scalar Multiplication

For a scalar $c \in \mathbb{R}$ (or $\mathbb{C}$) and a vector $\mathbf{v}$, the scalar multiplication is

$$c\mathbf{v} = \begin{bmatrix} cv_1 \\ cv_2 \\ \vdots \\ cv_n \end{bmatrix}$$

This operation is distributive over vector addition and scalar addition. Geometrically, scalar multiplication stretches or shrinks the vector by a factor of c (see Figure 2.2).

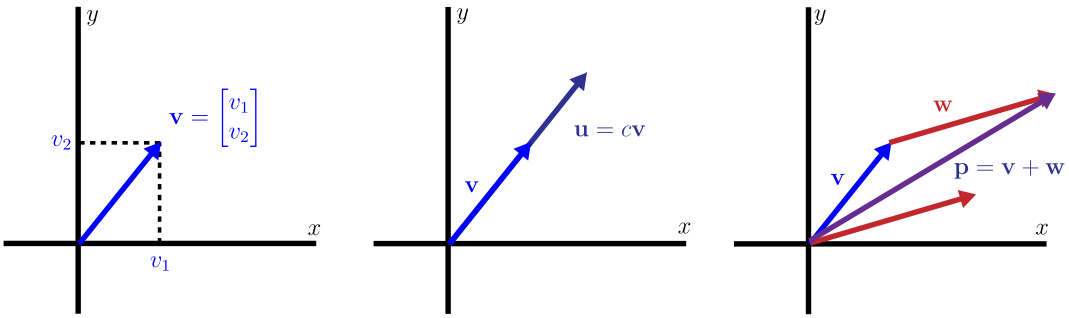


Figure 2.2: Vector addition and scalar multiplication in $\mathbb{R}^2$.

2.2.3 Linear Combinations and Span

A **linear combination** of vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ is an expression of the form:

$$\mathbf{w} = c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_k\mathbf{v}_k$$

where $c_1, c_2, \ldots, c_k$ are scalars. The set of all possible linear combinations of $\mathbf{v}_1, \ldots, \mathbf{v}_k$ is called their **span**:

$$\text{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\} = \left\{ \sum_{i=1}^{k} c_i\mathbf{v}_i \mid c_i \in \mathbb{R} \text{ or } \mathbb{C} \right\}$$

Example: Consider vectors $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ in $\mathbb{R}^2$. Their span is the entire $\mathbb{R}^2$ plane since any vector $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ can be written as $w_1\mathbf{v}_1 + w_2\mathbf{v}_2$

2.2.4 Linear Independence and Basis

Vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ are **linearly independent** if the only solution to

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \ldots + c_k\mathbf{v}_k = \mathbf{0}$$

is $c_1 = c_2 = \ldots = c_k = 0$.

A set of vectors that is linearly independent and spans the vector space is called a **basis**. The number of vectors in a basis equals the **dimension** of the space.

2.2.5 The Zero Vector

The **zero vector** $\mathbf{0}$ in $\mathbb{R}^n$ is:

$$\mathbf{0} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

It acts as the additive identity: $\mathbf{v} + \mathbf{0} = \mathbf{v}$.

2.2.6 The Unit Vectors

The **unit vectors** in $\mathbb{R}^n$ are:

$$\mathbf{e}_i = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \text{where } (\mathbf{e}_i)_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The unit vectors form a basis for $\mathbb{R}^n$, meaning any vector $\mathbf{v}$ can be written as a linear combination of the unit vectors.

Example: In $\mathbb{R}^3$, the unit vectors are $\mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, $\mathbf{e}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, and $\mathbf{e}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

2.2.7 Properties of Vector Spaces

For all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ and scalars $c, d \in \mathbb{R}$, the following hold:

1. $\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$ (commutativity)
2. $(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$ (associativity)
3. $\mathbf{v} + \mathbf{0} = \mathbf{v}$ (additive identity)
4. $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$ (additive inverse)
5. $c(d\mathbf{v}) = (cd)\mathbf{v}$ (associativity of scalar multiplication)
6. $(c + d)\mathbf{v} = c\mathbf{v} + d\mathbf{v}$ (distributivity over scalars)
7. $c(\mathbf{v} + \mathbf{w}) = c\mathbf{v} + c\mathbf{w}$ (distributivity over vectors)
8. $1\mathbf{v} = \mathbf{v}$ (multiplicative identity)

2.3 Matrices

A **matrix** is a rectangular array of scalars arranged in rows and columns. An $m \times n$ matrix $\mathbf{A}$ has m rows and n columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}.$$

Matrices can represent linear transformations from $\mathbb{R}^n$ to $\mathbb{R}^m$. The notation $\mathbf{A} \in \mathbb{R}^{m \times n}$ indicates the dimensions of $\mathbf{A}$ (and the fact that the entries are drawn from $\mathbb{R}$), and the notation a_{ij} denotes the element in the i-th row and j-th column.

Additionally, we can think of vectors as matrices with a single column—that is, as $n \times 1$ matrices. This perspective allows us to treat vectors and matrices uniformly.

2.3.1 Matrix Addition and Scalar Multiplication

Like vectors, matrices can be added and scaled entry-wise:

- **Addition**: $(\mathbf{A} + \mathbf{B})_{ij} = a_{ij} + b_{ij}$
- **Scalar Multiplication**: $(c\mathbf{A})_{ij} = c \cdot a_{ij}$

2.3.2 Matrix Multiplication

The product of an $m \times n$ matrix $\mathbf{A}$ and an $n \times p$ matrix $\mathbf{B}$ is an $m \times p$ matrix $\mathbf{C}$:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Matrix multiplication is associative and distributive but generally not commutative ($\mathbf{AB} \neq \mathbf{BA}$ in general). Additionally, the number of columns in $\mathbf{A}$ must equal the number of rows in $\mathbf{B}$ for the product to be defined.

Example: Multiply $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$:

$$\mathbf{AB} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}.$$

The product of two matrices, $\mathbf{AB}$, represents the composition of two linear transformations. If matrix $\mathbf{B}$ transforms a vector $\mathbf{x}$ into a new vector $\mathbf{y} = \mathbf{Bx}$, and matrix $\mathbf{A}$ then transforms $\mathbf{y}$ into $\mathbf{z} = \mathbf{Ay}$, the product matrix $\mathbf{C} = \mathbf{AB}$ is the single matrix that represents the entire transformation from $\mathbf{x}$ to $\mathbf{z}$ directly ($\mathbf{z} = \mathbf{Cx}$). This is why the order of multiplication matters; applying transformation $\mathbf{A}$ then $\mathbf{B}$ is generally not the same as applying $\mathbf{B}$ then $\mathbf{A}$.

2.3.3 Matrix-Vector Products

The product of a matrix $\mathbf{A}$ and a vector $\mathbf{x}$ is a vector:

$$(\mathbf{Ax})_i = \sum_{j=1}^{n} a_{ij} x_j$$

The number of columns in $\mathbf{A}$ must equal the number of elements in $\mathbf{x}$ for the product to be defined. Since vectors can be thought of as matrices with one column, matrix-vector multiplication is just a special case of matrix multiplication.

Example: For $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $\mathbf{x} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$:

$$\mathbf{Ax} = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 6 \\ 3 \cdot 5 + 4 \cdot 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

A matrix-vector product $\mathbf{Ax}$ can be viewed as applying a linear transformation to the vector $\mathbf{x}$. The matrix $\mathbf{A}$ acts as an operator that maps the vector $\mathbf{x}$ to a new vector, potentially rotating, scaling, or shearing it in the process. Equivalently, the resulting vector is a linear combination of the columns of $\mathbf{A}$, with the components of $\mathbf{x}$ serving as the weights.

2.3.4 Outer Product

The **outer product** of vectors $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^n$ is an $m \times n$ matrix:

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^\top = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \dots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \dots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \dots & u_m v_n \end{bmatrix}$$

The outer product of vectors satisfies the following properties:

- $(\mathbf{u} \otimes \mathbf{v})^\top = (\mathbf{v} \otimes \mathbf{u})$
- $(\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} = \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}$
- $\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = \mathbf{u} \otimes \mathbf{v} + \mathbf{u} \otimes \mathbf{w}$
- $c(\mathbf{v} \otimes \mathbf{u}) = (c\mathbf{v}) \otimes \mathbf{u} = \mathbf{v} \otimes (c\mathbf{u})$

If $\mathbf{u} \neq \mathbf{0}$ and $\mathbf{v} \neq \mathbf{0}$, then $\mathbf{u} \otimes \mathbf{v}$ always has matrix rank 1 since the columns of the outer product are all proportional to $\mathbf{u}$. As we will see later, this property has important implications in constructing low-rank approximations of matrices (Sherman-Morrison formula).

To emphasize, the outer product constructs a matrix from the product of two vectors, while the inner product (dot product) constructs a scalar from the product of two vectors.

2.3.5 Transpose of a Matrix

The **transpose** of $\mathbf{A}$, denoted $\mathbf{A}^\top$, is obtained by swapping rows with columns:

$$(\mathbf{A}^\top)_{ij} = a_{ji}$$

The transpose satisfies $(\mathbf{A}^\top)^\top = \mathbf{A}$ and $(\mathbf{A} + \mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top$.

In addition, the transpose of a product reverses order: $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$ whenever the product is defined.

Example: For $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $\mathbf{A}^\top = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

2.3.6 Special Matrices

- **Zero Matrix**: All entries are zero; acts as additive identity ($\mathbf{A} + \mathbf{0} = \mathbf{A}$).

$$\mathbf{0} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

- **Identity Matrix I**: Square matrix with ones on the diagonal and zeros elsewhere; acts as multiplicative identity ($\mathbf{AI} = \mathbf{IA} = \mathbf{A}$).

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

- **Symmetric Matrix**: $\mathbf{A} = \mathbf{A}^\top$.
- **Positive Semi-Definite Matrix**: Symmetric and satisfies $\mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x} \neq \mathbf{0}$. This is denoted $\mathbf{A} \succeq 0$. Also common is the notion of a positive definite matrix, which requires strict positivity ($\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$, denoted $\mathbf{A} \succ 0$).
- **Diagonal Matrix**: All off-diagonal entries are zero.
- **Triangular Matrix**: All entries above or below the main diagonal are zero.

2.3.7 Trace of a Matrix

For a square matrix $\mathbf{A}$, the **trace** is the sum of its diagonal elements:

$$\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} a_{ii}$$

The trace is linear: $\text{tr}(\mathbf{A} + \mathbf{B}) = \text{tr}(\mathbf{A}) + \text{tr}(\mathbf{B})$. Additionally, the trace of an $n \times n$ matrix is equal to the sum of its eigenvalues:

$$\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i$$

where the eigenvalues with algebraic multiplicity are counted multiple times. We will discuss eigenvalues and eigenvectors in more detail later. The trace also satisfies the following properties:

- $\text{tr}(\mathbf{A}^\top) = \text{tr}(\mathbf{A})$
- $\text{tr}(\mathbf{A} + \mathbf{B}) = \text{tr}(\mathbf{A}) + \text{tr}(\mathbf{B})$
- $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$

Finally, as a generalization of the last property mentioned above, we note that the trace is invariant under **cyclic permutations**, meaning that for any square matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ such that the product $\mathbf{ABC}$ is defined, we have $\text{tr}(\mathbf{ABC}) = \text{tr}(\mathbf{CAB}) = \text{tr}(\mathbf{BCA})$. However, note that this property does not hold for arbitrary permutations (e.g., $\text{tr}(\mathbf{ACB}) \neq \text{tr}(\mathbf{ABC})$).

Example: For $\mathbf{A} = \begin{bmatrix} 2 & -1 \\ 4 & 3 \end{bmatrix}$, $\text{tr}(\mathbf{A}) = 2 + 3 = 5$.

2.3.8 Determinant

For a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, the **determinant** $\det(\mathbf{A})$ is a scalar value that can tell us a lot about the matrix's properties. Geometrically, it represents the signed volume-scaling factor of the linear transformation described by $\mathbf{A}$. If $|\det(\mathbf{A})| = 2$, the transformation doubles the volume of a region, while if $|\det(\mathbf{A})| = 0.5$, it halves the volume. The sign of the determinant tells us whether the transformation preserves or reverses orientation. A positive determinant means the orientation is preserved, while a negative determinant means it's flipped (think of it as turning a shape "inside-out" like a reflection). If $\det(\mathbf{A}) = 0$, the transformation collapses the space into a lower dimension (e.g., squashing a 2D square into a line segment). When this happens, information is lost, and you can't reverse the transformation. This is why a matrix is **invertible if and only if its determinant is non-zero**.

The determinant has several important properties, including $\det(\mathbf{A}^\top) = \det(\mathbf{A})$, the multiplicative property $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$, and the fact that for an invertible matrix, $\det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$.

For a 2×2 matrix, the formula is

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc$$

For a 3×3 matrix, one common method is cofactor expansion along the first row:

$$\begin{aligned}\det\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}\right) &= a\det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b\det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c\det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right) \\ &= a(ei - fh) - b(di - fg) + c(dh - eg)\end{aligned}$$

The determinant of a general $n \times n$ matrix can be defined recursively via Laplace expansion, which expresses it as a signed sum of the entries of any chosen row or column times the determinants of their $(n-1) \times (n-1)$ minors. This recursion reduces ultimately to the 2×2 case. However, in practice, Laplace expansion is only used for small matrices; large determinants are almost always computed using Gaussian elimination or LU decomposition instead.

2.3.9 Inverse of a Matrix

Conceptually, the inverse of a matrix, denoted $\mathbf{A}^{-1}$, is an operator that reverses the linear transformation performed by the original matrix $\mathbf{A}$. If $\mathbf{A}$ transforms a vector $\mathbf{x}$ into $\mathbf{y}$ (i.e., $\mathbf{y} = \mathbf{Ax}$), then its inverse $\mathbf{A}^{-1}$ transforms $\mathbf{y}$ back to the original vector $\mathbf{x}$ (i.e., $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$).

Formally, a square matrix $\mathbf{A}$ is **invertible** (or **non-singular**) if there exists a matrix $\mathbf{A}^{-1}$ such that:

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix. This property is analogous to a number and its reciprocal (e.g., $5 \cdot 5^{-1} = 1$).

A matrix is invertible if and only if its determinant is non-zero ($\det(\mathbf{A}) \neq 0$). If the determinant is zero, the matrix is singular, meaning its transformation is irreversible because it collapses the space into a lower dimension (e.g., mapping a 2D plane onto a line), and the original vector cannot be uniquely recovered.

This "undoing" capability is fundamental for solving systems of linear equations. A system represented as $\mathbf{Ax} = \mathbf{b}$ can be solved for the unknown vector $\mathbf{x}$ by applying the inverse: $\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$, which simplifies to $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Important properties of inverses include $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$ (note the reversed order) and $(\mathbf{A}^\top)^{-1} = (\mathbf{A}^{-1})^\top$. For a 2×2 matrix, the inverse can be calculated directly:

$$\text{If } \mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \text{ then } \mathbf{A}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}, \quad \text{provided } ad - bc \neq 0$$

While the 2×2 formula is simple, computing the inverse for a general $n \times n$ matrix is a numerically intensive task. In practice, one typically avoids computing the inverse explicitly. For problems like solving $\mathbf{Ax} = \mathbf{b}$, it is both more efficient and more numerically stable to solve the system directly using matrix factorization or other methods instead, which we will discuss in more detail later.

2.4 Inner Products and Norms

2.4.1 Dot Product

The **dot product** (inner product) of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ is

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \sum_{i=1}^{n} u_i v_i$$

It measures the projection of one vector onto another, which we can geometrically interpret as the product of the magnitudes of the vectors and the cosine of the angle between them (Figure 2.3).

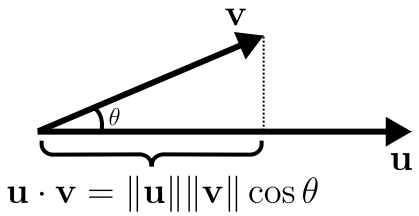


Figure 2.3: Geometric interpretation of the dot product.

The dot product satisfies the following properties:

- $\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$ (commutativity)
- $\mathbf{u} \cdot (\mathbf{v} + \mathbf{w}) = \mathbf{u} \cdot \mathbf{v} + \mathbf{u} \cdot \mathbf{w}$ (distributivity)
- $(\alpha \mathbf{u}) \cdot \mathbf{v} = \alpha(\mathbf{u} \cdot \mathbf{v})$ (homogeneity)
- $\mathbf{u} \cdot \mathbf{u} \geq 0$ and $\mathbf{u} \cdot \mathbf{u} = 0$ if and only if $\mathbf{u} = \mathbf{0}$ (positive definiteness)

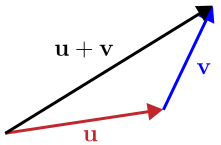
2.4.2 Norms of Vectors

A **norm** is a function that assigns a non-negative length or size to vectors. The most common norms are:

- **1-Norm**:
$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i|$$
- **2-Norm** (Euclidean norm):
$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{\mathbf{v}^\top \mathbf{v}}$$
- **Infinity Norm**:
$$\|\mathbf{v}\|_\infty = \max_{1 \leq i \leq n} |v_i|$$
- **p-Norm**:
$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}$$

We note that the 1-norm, 2-norm, and infinity norm are special cases of the p-norm. The 2-norm is the most commonly used norm in practice. These norms satisfy:

- $\|\mathbf{v}\| \geq 0$ and $\|\mathbf{v}\| = 0$ if and only if $\mathbf{v} = \mathbf{0}$.
- $\|\alpha \mathbf{v}\| = |\alpha| \|\mathbf{v}\|$ for scalar α.
- Triangle inequality: $\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\|$.



Example: For $\mathbf{v} = \begin{bmatrix} 3 \\ -4 \end{bmatrix}$:

$$\|\mathbf{v}\|_1 = |3| + |-4| = 7,$$
$$\|\mathbf{v}\|_2 = \sqrt{3^2 + (-4)^2} = 5,$$
$$\|\mathbf{v}\|_\infty = \max\{|3|, |-4|\} = 4.$$

2.4.3 Orthogonality

Vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ are **orthogonal** if $\mathbf{u}^\top \mathbf{v} = 0$. A set of vectors is **orthonormal** if vectors are mutually orthogonal and each has unit norm. A square matrix $\mathbf{Q}$ is **orthogonal** if $\mathbf{Q}^\top \mathbf{Q} = \mathbf{Q}\mathbf{Q}^\top = \mathbf{I}$, which implies $\mathbf{Q}^{-1} = \mathbf{Q}^\top$ and that $\|\mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ for all $\mathbf{x}$.

2.4.4 Norms of Matrices

Matrix norms extend the concept of vector norms to matrices, providing a measure of the "size" or "magnitude" of a matrix. Just as vector norms quantify the length of vectors, matrix norms help assess various properties of matrices, such as their behavior under linear transformations, stability in numerical computations, and convergence in iterative methods.

2.4.4.1 Induced Matrix Norms One fundamental way to define a matrix norm is through an **induced (operator) norm**, which is derived from a vector norm. Given a vector norm $\|\cdot\|$ on $\mathbb{R}^n$ or $\mathbb{C}^n$, the induced matrix norm for a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is defined as

$$\|\mathbf{A}\| = \max_{\mathbf{x} \neq \mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|}{\|\mathbf{x}\|}$$

The induced matrix norm measures the maximum stretching factor of the matrix $\mathbf{A}$ when it acts as a linear transformation on vectors. Specifically, it identifies the largest possible ratio of the norm of $\mathbf{A}\mathbf{x}$ to the norm of $\mathbf{x}$ across all non-zero vectors $\mathbf{x}$. This tells us how much $\mathbf{A}$ can amplify a vector.

Induced norms satisfy the following properties:

- $\|\mathbf{A}\| \geq 0, \quad \|\mathbf{A}\| = 0$ only if $\mathbf{A} = \mathbf{0}$
- $\|c\mathbf{A}\| = |c|\|\mathbf{A}\|$
- $\|\mathbf{A}\mathbf{x}\| \leq \|\mathbf{A}\|\|\mathbf{x}\|$ (submultiplicative property)
- $\|\mathbf{A}\mathbf{B}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$ (submultiplicative property)
- $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$

The submultiplicative property is a very important property of induced/operator norms! This property allows us to bound the norm of a product of matrices by the product of their norms and will be very useful in the analysis of iterative methods.

2.4.4.2 Common Matrix Norms Several matrix norms are commonly used. Below, we take a look at a few of these norms and their interpretations. The first three are induced norms derived from vector norms (and in particular p-norms), while the last norm is a matrix-specific norm.

- **1-Norm (Maximum Column Sum)**:

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m} |a_{ij}|$$

The 1-norm of a matrix is the largest sum of absolute values of the elements in any single column. It represents the maximum effect that $\mathbf{A}$ can have on the 1-norm of a vector, emphasizing the most "influential" column in terms of contributing to the overall transformation.

- **2-Norm (Spectral Norm)**:
$$\|\mathbf{A}\|_2 = \sigma_{\max}(\mathbf{A})$$
where $\sigma_{\max}(\mathbf{A})$ denotes the largest singular value of $\mathbf{A}$. The 2-norm measures the maximum stretching factor of $\mathbf{A}$ in any direction, corresponding to the largest singular value. It is closely related to the concept of eigenvalues and provides a precise measure of the matrix's action on vectors in terms of scaling.

- **Infinity Norm (Maximum Row Sum)**:
$$\|\mathbf{A}\|_\infty = \max_{1\le i\le m} \sum_{j=1}^{n} |a_{ij}|$$
The infinity norm is the largest sum of absolute values of the elements in any single row. It reflects the maximum effect that $\mathbf{A}$ can have on the infinity norm of a vector, highlighting the most "influential" row in terms of transformation impact.

- **Frobenius Norm**:
$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}$$
Again, this is NOT an induced norm. The Frobenius norm is very similar to the Euclidean norm (which is defined for vectors). It measures the "overall size" of the matrix by taking the square root of the sum of the squares of all its entries. This norm is particularly useful in applications where the collective magnitude of all matrix elements is of interest, such as in least squares problems and matrix approximations.

2.4.4.3 Relationships Between Matrix Norms For any matrix $\mathbf{A}$ of size $m \times n$ and rank r, the following inequalities hold:

$$\|\mathbf{A}\|_2 \le \|\mathbf{A}\|_F \le \sqrt{r}\,\|\mathbf{A}\|_2,$$
$$\|\mathbf{A}\|_2 \le \sqrt{\|\mathbf{A}\|_1\,\|\mathbf{A}\|_\infty}.$$

2.5 Eigenvalues and Eigenvectors

A linear transformation, represented by a matrix $\mathbf{A}$, acts on vectors, typically changing their direction. However, for many square matrices, there exist special non-zero vectors called **eigenvectors** whose direction is unchanged by the transformation. When multiplied by the matrix, these vectors are simply scaled: stretched, shrunk, and/or flipped.

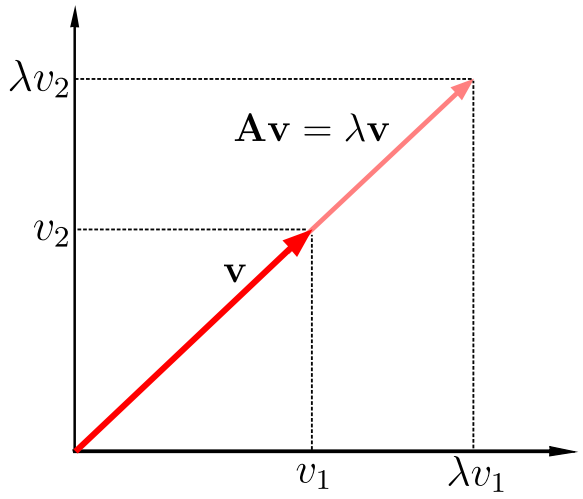


Figure 2.4: Visualization of the eigenvalue equation $\mathbf{Av} = \lambda\mathbf{v}$, where $\mathbf{v} = (v_1, v_2)$.

Formally, for a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, a non-zero vector $\mathbf{v}$ is an eigenvector if it satisfies the equation:

$$\mathbf{Av} = \lambda\mathbf{v}$$

The scalar λ is the corresponding **eigenvalue**, which represents the scaling factor. An $n \times n$ matrix has at most n distinct eigenvalues.

Warning: An eigenvector is not unique. If $\mathbf{v}$ is an eigenvector, then so is any scalar multiple $c\mathbf{v}$ for $c \neq 0$. They both correspond to the same eigenvalue and define the same invariant direction.

The concept is very important for understanding iterative methods. For example, look at the Jacobi iteration error propagation (which we will encounter shortly), $e_{t+1} = \mathbf{S}e_t$, which implies $e_t = \mathbf{S}^t e_0$. Eigenvalues and eigenvectors tell us what happens to a vector when it's multiplied by the same matrix repeatedly.

2.5.1 Finding Eigenvalues and Eigenvectors

To find eigenvalues, we rearrange the defining equation:

$$\mathbf{Av} - \lambda\mathbf{v} = 0 \implies (\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0$$

For this equation to have a non-trivial solution for $\mathbf{v}$ (i.e., $\mathbf{v} \neq 0$), the matrix $(\mathbf{A} - \lambda\mathbf{I})$ must be singular. This means its determinant must be zero.

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

This equation defines the **characteristic polynomial** of $\mathbf{A}$, an n-degree polynomial in λ. Its n roots (which may be real or complex and are counted with algebraic multiplicity) are the eigenvalues of $\mathbf{A}$.

Solving the characteristic polynomial is practical for small, textbook examples. However, this method is generally ill-conditioned and not recommended for numerical work with larger matrices; robust iterative algorithms (e.g., QR/Schur) are used in practice.

Once an eigenvalue λ_i is found, the corresponding eigenvector(s) $\mathbf{v}_i$ are found by solving the linear system $(\mathbf{A} - \lambda_i\mathbf{I})\mathbf{v}_i = 0$, which is equivalent to finding the null space of $(\mathbf{A} - \lambda_i\mathbf{I})$.

Example (finding eigenvalues and eigenvectors): Let $\mathbf{A} = \begin{bmatrix} -2 & 1 \\ 1 & -2 \end{bmatrix}$.

1. Find Eigenvalues: Solve $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$:

$$\det \begin{bmatrix} -2-\lambda & 1 \\ 1 & -2-\lambda \end{bmatrix} = (-2-\lambda)^2 - 1 = \lambda^2 + 4\lambda + 3 = 0$$

Factoring gives $(\lambda + 3)(\lambda + 1) = 0$. The eigenvalues are $\lambda_1 = -3$ and $\lambda_2 = -1$.

2. Find Eigenvectors: For $\lambda_1 = -3$, we solve $(\mathbf{A} - (-3)\mathbf{I})\mathbf{v}_1 = 0$:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \implies v_1 + v_2 = 0$$

An eigenvector is $\mathbf{v}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.

For $\lambda_2 = -1$, we solve $(\mathbf{A} - (-1)\mathbf{I})\mathbf{v}_2 = 0$:

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \implies -v_1 + v_2 = 0$$

An eigenvector is $\mathbf{v}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

2.5.2 Multiplicity and Diagonalizability

The number of times an eigenvalue appears as a root of the characteristic polynomial is its **algebraic multiplicity**. The number of linearly independent eigenvectors for that eigenvalue is its **geometric multiplicity**. For any eigenvalue, $1 \leq$ geometric multiplicity $\leq$ algebraic multiplicity. A matrix is called **defective** if for some eigenvalue its geometric multiplicity is less than its algebraic multiplicity; equivalently, it does not have n linearly independent eigenvectors. A matrix with a "complete set" of eigenvectors (geometric = algebraic multiplicity for all eigenvalues) is **diagonalizable**.

2.5.3 Eigendecomposition

If an $n \times n$ matrix $\mathbf{A}$ is diagonalizable, it has n linearly independent eigenvectors $\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$. We can bundle these into a matrix $\mathbf{W} = [\mathbf{v}_1 | \ldots | \mathbf{v}_n]$ and the corresponding eigenvalues into a diagonal matrix $\mathbf{\Lambda}$. The eigenvalue equation for all eigenvectors can be written compactly as:

$$\mathbf{AW} = \mathbf{W\Lambda}$$

This leads to the **eigendecomposition** of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{W\Lambda W}^{-1} \quad \text{and} \quad \mathbf{\Lambda} = \mathbf{W}^{-1}\mathbf{AW}$$

This decomposition reveals the fundamental behavior of $\mathbf{A}$ as a simple scaling along the eigenvector axes.

Properties and Special Cases
For a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_1, \ldots, \lambda_n$:

- The determinant is the product of its eigenvalues: $\det(\mathbf{A}) = \prod_{i=1}^{n} \lambda_i$.
- The trace is the sum of its eigenvalues: $\text{tr}(\mathbf{A}) = \sum_{i=1}^{n} \lambda_i$.
- The eigenvalues of a triangular matrix are its diagonal entries.

If $\mathbf{A}$ is a real symmetric matrix ($\mathbf{A} = \mathbf{A}^T$):

- It is always diagonalizable and has a complete set of orthonormal eigenvectors.
- All its eigenvalues are real.
- Its eigendecomposition is $\mathbf{A} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^T$, where $\mathbf{W}$ is an orthogonal matrix ($\mathbf{W}^{-1} = \mathbf{W}^T$).
- If $\mathbf{A}$ is positive definite, all its eigenvalues are positive ($\lambda_i > 0$).

2.6 Computational Complexity of Matrix and Vector Operations

We must understand the computational costs of basic matrix and vector operations in order to develop efficient numerical algorithms. Let's explore three fundamental operations—dot product, matrix-vector multiplication, and matrix-matrix multiplication—and analyze their computational complexities by counting the number of required multiplications and additions.

Dot Product For two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$, the dot product $\mathbf{y}^\top \mathbf{x}$ is calculated by multiplying corresponding elements and summing the results:

$$\mathbf{y}^\top \mathbf{x} = \sum_{i=1}^{N} y_i x_i$$

This operation involves N multiplications (one for each pair $y_i x_i$) and $N-1$ additions (to sum the N products). The computational complexity is therefore $O(N + N - 1) = O(N)$.

Matrix-Vector Multiplication Given a matrix $\mathbf{A} \in \mathbb{R}^{M \times N}$ and a vector $\mathbf{x} \in \mathbb{R}^N$, the product $\mathbf{Ax}$ results in a vector $\mathbf{b} \in \mathbb{R}^M$:

$$b_i = \sum_{j=1}^{N} A_{ij} x_j, \quad \text{for } i = 1, 2, \ldots, M$$

For each of the M rows, there are N multiplications (each element A_{ij} is multiplied by x_j) and $N-1$ additions (to sum the products for each b_i). Therefore, the total number of operations is $O(MN + M(N-1)) = O(MN)$.

Matrix-Matrix Multiplication When multiplying two matrices $\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times P}$, the resulting matrix $\mathbf{C} = \mathbf{AB}$ has dimensions $M \times P$:

$$C_{ik} = \sum_{j=1}^{N} A_{ij} B_{jk}, \quad \text{for } i = 1, 2, \ldots, M \text{ and } k = 1, 2, \ldots, P$$

For each element C_{ik} (there are $M \times P$ such elements), there are N multiplications (each A_{ij} multiplied by B_{jk}) and $N-1$ additions (to sum the products for each C_{ik}). So we get MNP multiplications and $MP(N-1)$ additions and hence an overall complexity of $O(MNP)$.

Example: To make these concepts more tangible, let's consider multiplying two small matrices:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}, \quad \mathbf{C} = \mathbf{AB}$$

The resulting matrix $\mathbf{C} \in \mathbb{R}^{3 \times 3}$ will have elements:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ c_{13} &= a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ &\vdots \end{aligned}$$

We see that each element of $\mathbf{C}$ requires 3 multiplications and 2 additions. And since $\mathbf{C}$ has 9 elements, the total number of operations is $9 \cdot (3 + 2) = 45$.

But wait, isn't the complexity of matrix-matrix multiplication $O(MNP)$? And in this case $M = N = P = 3$, so shouldn't we have $3^3 = 27$ operations? No! That is not how big-O notation works. Big-O notation is NOT a precise count of operations but rather an upper bound on the growth rate of the number of operations as the size of the matrices grows.

It is worth mentioning (this is not necessary to know for 10.34) that, in reality, the computational complexity of matrix-matrix multiplication is not exactly cubic. A lot of research has been done to optimize matrix-matrix multiplication due to its extreme importance in the field of optimization and machine learning. For square $N \times N$ matrices, the fastest known algorithms run in $O(N^\omega)$ time, where, as of 2024, the record is $\omega < 2.371339$ (this upper bound is from this paper). Very roughly, these bounds come from the Coppersmith-Winograd line of ideas: view matrix multiply as a tensor and look for low-rank decompositions. The recent twist is to allow more asymmetry between the three dimensions (don't force two to behave identically); similar tricks help with rectangular cases, too. But these bounds are galactic algorithms, meaning the constants (for big O) are huge, so real code sticks to near-cubic methods with blocking, cache-friendly layouts, vectorization, and parallelism. Strassen's method ($O(N^{\log_2 7}) \approx O(N^{2.8074})$) sometimes shows up for very large N, but most production matrix multiplication is effectively $O(N^3)$, so you should stick with $O(N^3)$ as a rule of thumb.

2.7 Optimizing Computations in Complex Expressions

When working with expressions involving multiple matrix and vector operations, the order in which you perform these operations can greatly affect the computational cost. By strategically choosing the sequence of calculations, you can minimize the number of required operations and optimize performance.

Matrix multiplication is associative but generally not commutative; matrix addition is both associative and commutative. The dot product is commutative, and scalars commute with everything. Some structured matrices (e.g., diagonal) commute with each other, but in general do *not* assume $\mathbf{AB} = \mathbf{BA}$. These properties allow some flexibility in how calculations are grouped and ordered.

Matrix-Vector Multiplication in Computation Chains As noted previously, the computational complexity of matrix-vector multiplication is "quadratic" while the complexity of matrix-matrix multiplication is "cubic." Therefore, when given a chain of matrix-matrix multiplications that has at least one matrix-vector multiplication, it is often more efficient to perform the operations in a way that minimizes the number of matrix-matrix multiplications and maximizes the number of matrix-vector multiplications.

It is easiest to see this by example. Consider the expression:

$$\mathbf{y} = \mathbf{ABCx}$$

where $\mathbf{A} \in \mathbb{R}^{3\times3}$, $\mathbf{B} \in \mathbb{R}^{3\times3}$, $\mathbf{C} \in \mathbb{R}^{3\times3}$, and $\mathbf{x} \in \mathbb{R}^3$. To compute $\mathbf{y}$, we can perform the operations in a few different ways by using the associativity property of matrix multiplication. The first way we consider is to rewrite the expression as

$$\mathbf{y} = \mathbf{AB}(\mathbf{Cx}) = \mathbf{A}(\mathbf{Bd}) = \mathbf{Ae}$$

where we compute $\mathbf{d} = \mathbf{Cx}$ first, then $\mathbf{e} = \mathbf{Bd}$, and finally $\mathbf{y} = \mathbf{Ae}$. Importantly, this sequence of operations requires only matrix-vector multiplications (three in total) and no matrix-matrix multiplications. We can compare this to a second way of computing $\mathbf{y}$:

$$\mathbf{y} = (\mathbf{AB})\mathbf{Cx} = (\mathbf{DC})\mathbf{x} = \mathbf{Fx}$$

This requires two matrix-matrix multiplications and one matrix-vector multiplication. Therefore, the first way is more efficient than the second way (concretely, 45 operations vs. 105 operations—check this yourself!).

Warning: The advice to always perform matrix-vector products first is an excellent rule of thumb, but it is not a universal law. The true underlying principle is to **avoid creating large intermediate products**, whether they are matrices or vectors. In cases involving non-square matrices, this can lead to surprising results. Consider the expression $\mathbf{y} = \mathbf{ABx}$ where the dimensions are not square:

- $\mathbf{A} \in \mathbb{R}^{1\times 1000}$
- $\mathbf{B} \in \mathbb{R}^{1000\times 100}$
- $\mathbf{x} \in \mathbb{R}^{100}$

Let's compare the two ways of computing the result based on the number of multiplications:

1. **Matrix-Vector First:** $\mathbf{y} = \mathbf{A}(\mathbf{Bx})$ We first compute $\mathbf{d} = \mathbf{Bx}$. This requires $1000 \times 100 = 100,000$ multiplications and creates a large intermediate vector $\mathbf{d} \in \mathbb{R}^{1000}$. Then, computing $\mathbf{y} = \mathbf{Ad}$ requires $1 \times 1000 = 1,000$ multiplications. Total Cost: $100,000 + 1,000 = 101,000$ multiplications.
2. **Matrix-Matrix First:** $\mathbf{y} = (\mathbf{AB})\mathbf{x}$ We first compute $\mathbf{C} = \mathbf{AB}$. This requires $1 \times 1000 \times 100 = 100,000$ multiplications and creates a small intermediate row vector $\mathbf{C} \in \mathbb{R}^{1\times 100}$. Then, computing $\mathbf{y} = \mathbf{Cx}$ requires $1\times 100 = 100$ multiplications. Total Cost: $100,000+100 = 100,100$ multiplications.

In this case, performing the matrix-matrix multiplication first is slightly more efficient in terms of number of floating point operations required (the amount of additions is the same in either case: 99,999). It avoids creating the huge intermediate vector ($\mathbf{d} \in \mathbb{R}^{1000}$) and instead creates a much smaller intermediate vector ($\mathbf{C} \in \mathbb{R}^{1\times 100}$). So we can indeed see that the optimal order depends entirely on the dimensions of all matrices in the chain. Actually, finding this optimal order is a classic computer science problem known as the matrix chain ordering problem (you can read its Wikipedia page if you're curious!).

Diagonal Matrices Suppose $\mathbf{D} \in \mathbb{R}^{N\times N}$ is a diagonal matrix and $\mathbf{x} \in \mathbb{R}^N$, and you need to compute $\mathbf{y} = \mathbf{Dx}$. Using standard matrix multiplication, this operation would be $O(N^2)$. But since only the diagonal elements D_{ii} are non-zero, we only need to perform N multiplications (one multiplication per diagonal element) and zero additions, meaning the computational complexity can be reduced to $O(N)$.

Sparse Matrices Sparse matrices (which are *very* common in scientific applications!) are matrices for which most elements are zero. They are stored in a way that only the non-zero elements are explicitly represented (in MATLAB, this is never automatically done for you—you must use the `sparse` function yourself to convert from dense to sparse format). Commonly, we must multiply a sparse matrix with a dense vector, which means that the result is the dot product of each sparse row of the matrix with the dense vector. This requires significantly fewer operations than a full matrix-vector multiplication (since we do not need to actually multiply anything with zero).

Symmetric Matrices Symmetric matrices are matrices that are equal to their own transpose: $\mathbf{A} = \mathbf{A}^\top$. For a symmetric matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$, a natural question to ask is: can we halve the work in $\mathbf{y} = \mathbf{Ax}$ by only touching one triangle? You *can* store just one triangle (saving about half the memory), but you *cannot* cut the arithmetic in half. To see why, write out the updates if you traverse only the upper triangle, and you'll find that the number of operations needed still comes out to $O(N^2)$. Where symmetry really shines is in factorizations/solves.

2.8 Numerical Solution of Linear Systems

Linear systems arise in many scientific and engineering applications (even in seemingly nonlinear cases, we can Taylor expand to a locally linear system), so we should develop tools to numerically solve them. The general form of a linear system is

$$\mathbf{Ax} = \mathbf{b} \tag{2.1}$$

where $\mathbf{A} \in \mathbb{R}^{M\times N}$ is a known coefficient matrix, $\mathbf{x} \in \mathbb{R}^N$ is the vector of unknowns, and $\mathbf{b} \in \mathbb{R}^M$ is a known vector. The goal is to find $\mathbf{x}$ that satisfies this equation. We know from linear algebra that this linear system has a unique solution for a given $\mathbf{b}$ if and only if $\mathbf{A}$ is square ($M = N$) and has full rank (i.e., $\text{rank}(\mathbf{A}) = N$).

Otherwise, the system may have no solution or infinitely many solutions. If $M \neq N$, we typically seek a least-squares ($M > N$) or minimum-norm ($M < N$) solution.

2.8.1 Existence and Uniqueness of Solutions

We next formalize when a linear system has a solution and when that solution is unique. When needed, we denote by $\mathbf{A}_i^c$ the i-th column and by $\mathbf{A}_i^r$ the i-th row.

2.8.1.1 The four fundamental subspaces

For a matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ there are four canonical subspaces:

1. **Column space (range):** $\mathcal{R}(\mathbf{A}) = \text{span}\{\mathbf{A}_1^c, \ldots, \mathbf{A}_M^c\} \subseteq \mathbb{R}^N$
2. **Null space (kernel):** $\mathcal{N}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^M : \mathbf{A}\mathbf{x} = \mathbf{0}\}$;
3. **Row space:** $\mathcal{R}(\mathbf{A}^\top) = \text{span}\{\mathbf{A}_1^r, \ldots, \mathbf{A}_N^r\} \subseteq \mathbb{R}^M$
4. **Left null space:** $\mathcal{N}(\mathbf{A}^\top) = \{\mathbf{y} \in \mathbb{R}^N : \mathbf{A}^\top\mathbf{y} = \mathbf{0}\}$

These subspaces organize the two core questions for a system $\mathbf{A}\mathbf{x} = \mathbf{b}$: *existence* (is $\mathbf{b}$ achievable?) and *uniqueness* (if achievable, is the achiever $\mathbf{x}$ unique?).

Think of $\mathbf{A} : \mathbb{R}^M \to \mathbb{R}^N$ as a machine that takes an input vector $\mathbf{x}$ and produces an output $\mathbf{b} = \mathbf{A}\mathbf{x}$. The set of all outputs the machine can possibly produce is the *column space* $\mathcal{R}(\mathbf{A})$. Directions in input space that the machine completely ignores (they map to $\mathbf{0}$) form the *null space* $\mathcal{N}(\mathbf{A})$. Thus:

$$\text{existence is about } \mathbf{b} \in \mathcal{R}(\mathbf{A}), \qquad \text{uniqueness is about } \mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$$

The four subspaces come in orthogonal pairs:

$$\mathcal{R}(\mathbf{A}) = \mathcal{N}(\mathbf{A}^\top)^\perp \quad \text{and} \quad \mathcal{R}(\mathbf{A}^\top) = \mathcal{N}(\mathbf{A})^\perp$$

Consequently,

$$\mathbb{R}^N = \mathcal{R}(\mathbf{A}) \oplus \mathcal{N}(\mathbf{A}^\top), \qquad \mathbb{R}^M = \mathcal{R}(\mathbf{A}^\top) \oplus \mathcal{N}(\mathbf{A})$$

In words, this means that for an $N \times M$ matrix $\mathbf{A}$, every vector in $\mathbb{R}^N$ can be written uniquely as a sum of something in the column space $\mathcal{R}(\mathbf{A})$ and something in the left null space $\mathcal{N}(\mathbf{A}^\top)$. Likewise, every vector in $\mathbb{R}^M$ splits uniquely into a part in the row space $\mathcal{R}(\mathbf{A}^\top)$ and a part in the null space $\mathcal{N}(\mathbf{A})$. As a dimension check this yields

$$\dim \mathcal{R}(\mathbf{A}) + \dim \mathcal{N}(\mathbf{A}^\top) = N \qquad \dim \mathcal{R}(\mathbf{A}^\top) + \dim \mathcal{N}(\mathbf{A}) = M$$

It is also a basic fact that row rank equals column rank:

$$\text{rank}(\mathbf{A}) = \dim \mathcal{R}(\mathbf{A}) = \dim \mathcal{R}(\mathbf{A}^\top) = \text{rank}(\mathbf{A}^\top)$$

Theorem (existence via the column space): A solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ exists for a given $\mathbf{b}$ if and only if $\mathbf{b} \in \mathcal{R}(\mathbf{A})$.

Proof. Using the column view,

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{M} x_i \, \mathbf{A}_i^c$$

If $\mathbf{b} \in \mathcal{R}(\mathbf{A})$, then by definition there exist coefficients x_i so that the sum equals $\mathbf{b}$, hence a solution exists. Conversely, if $\mathbf{A}\mathbf{x} = \mathbf{b}$ for some $\mathbf{x}$, then $\mathbf{b}$ is a linear combination of the columns of $\mathbf{A}$, so $\mathbf{b} \in \mathcal{R}(\mathbf{A})$.

We can also look at things geometrically. The column space is a subspace (a line/plane/higher-dimensional flat) inside $\mathbb{R}^N$. Asking for a solution is asking whether $\mathbf{b}$ lies *on* that flat. If not, there is no exact solution; if yes, there is at least one.

Example (existence). Let

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{b}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{b}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}.$$

The column space is $\mathcal{R}(\mathbf{A}) = \text{span}\{(1,0,0)^\top\}$, a 1-dimensional line in $\mathbb{R}^3$. Thus $\mathbf{b}_1$ lies on that line, so solutions exist (e.g., $x_1 = 1$ and x_2, x_3 arbitrary). The vector $\mathbf{b}_2$ lies off the line, so no solution exists.

2.8.1.2 Consistency via the left null space (Fredholm alternative) The system $\mathbf{Ax} = \mathbf{b}$ is solvable if and only if $\mathbf{b} \perp \mathcal{N}(\mathbf{A}^\top)$; equivalently, $\mathbf{b} \in \mathcal{R}(\mathbf{A})$. In practice this gives a test: if $\{\mathbf{y}_1, \ldots, \mathbf{y}_k\}$ is a basis of $\mathcal{N}(\mathbf{A}^\top)$, then $\mathbf{Ax} = \mathbf{b}$ is consistent precisely when $\mathbf{y}_j^\top \mathbf{b} = 0$ for all j.

Theorem (Uniqueness via the null space): If a solution to $\mathbf{Ax} = \mathbf{b}$ exists, it is unique if and only if $\dim \mathcal{N}(\mathbf{A}) = 0$ (equivalently, $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$).

Proof ($\Leftarrow$). If $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$ and $\mathbf{Ax} = \mathbf{Ay} = \mathbf{b}$, then $\mathbf{A}(\mathbf{x} - \mathbf{y}) = \mathbf{0}$, hence $\mathbf{x} - \mathbf{y} = \mathbf{0}$ and $\mathbf{x} = \mathbf{y}$.

Proof ($\Rightarrow$). Suppose a solution exists and is unique. If there were a nonzero $\mathbf{x}^H \in \mathcal{N}(\mathbf{A})$, then for any particular solution $\mathbf{x}^P$, the vector $\mathbf{x}^P + \mathbf{x}^H$ would be a distinct solution, a contradiction.

Example (uniqueness with $M < N$): Consider

$$\mathbf{A} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times 2}.$$

The columns are $\mathbf{c}_1 = (1,1,1,0)^\top$ and $\mathbf{c}_2 = (0,0,0,1)^\top$, which are linearly independent. Hence $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$ and any consistent system has a unique solution. Concretely, a right-hand side $\mathbf{b} = (b_1, b_2, b_3, b_4)^\top$ is in $\mathcal{R}(\mathbf{A})$ iff $b_1 = b_2 = b_3$; when this holds, the unique solution is $x_1 = b_1$ and $x_2 = b_4$.

2.8.1.3 Rank and nullity The rank and nullity of $\mathbf{A} \in \mathbb{R}^{N\times M}$ are

$$\text{rank}(\mathbf{A}) = \dim \mathcal{R}(\mathbf{A}), \qquad \text{nullity}(\mathbf{A}) = \dim \mathcal{N}(\mathbf{A}).$$

Always, $\text{rank}(\mathbf{A}) \le \min\{M, N\}$. We say that $\mathbf{A}$ has full column rank if $\text{rank}(\mathbf{A}) = M$ and full row rank if $\text{rank}(\mathbf{A}) = N$.

Rank–nullity theorem: For every $\mathbf{A} \in \mathbb{R}^{N\times M}$,

$$\text{rank}(\mathbf{A}) + \text{nullity}(\mathbf{A}) = M$$

An immediate consequence of the rank-nullity theorem is that if $M > N$ (more variables than equations), then $\text{rank}(\mathbf{A}) \le N$ so $\text{nullity}(\mathbf{A}) \ge M - N > 0$. Thus, whenever a solution exists, it cannot be unique. Conversely, if $N > M$ (more equations than variables), uniqueness (when consistent) is possible exactly when $\mathbf{A}$ has full column rank M, but solvability for *every* $\mathbf{b} \in \mathbb{R}^N$ is impossible since that would require rank $N > M$.

2.8.1.4 Square case and invertibility When $\mathbf{A} \in \mathbb{R}^{N\times N}$, the following are equivalent for a square system:

- for every $\mathbf{b} \in \mathbb{R}^N$ there exists a *unique* solution of $\mathbf{Ax} = \mathbf{b}$
- $\mathbf{A}$ is invertible
- $\det(\mathbf{A}) \neq 0$
- $\text{rank}(\mathbf{A}) = N$
- $\dim \mathcal{N}(\mathbf{A}) = 0$

Each statement enforces both existence (full row rank) and uniqueness (full column rank), which coincide in the square case.

Now, we return to the problem of finding the solution of Eq. 2.1. At first glance, it might seem straightforward to solve for $\mathbf{x}$ by simply computing the inverse of $\mathbf{A}$, since theoretically, if $\mathbf{A}$ is invertible, the solution is given by

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

However, in practice, computing $\mathbf{A}^{-1}$ directly is generally avoided for several important reasons: (1) computational cost, (2) numerical stability, and (3) memory requirements. For these reasons, numerical methods focus on solving linear systems without explicitly computing the inverse of $\mathbf{A}$. There are two primary classes of numerical methods for solving linear systems: **direct methods** and **iterative methods**. Each has its own advantages and is suitable for different types of problems.

2.8.2 Condition Number

Before looking at specific direct and iterative methods, we introduce the notion of the **condition number** of a matrix $\mathbf{A}$, denoted $\kappa(\mathbf{A})$, which helps us understand the sensitivity of a linear system to perturbations and numerical errors. We hope that for our linear system $\mathbf{Ax} = \mathbf{b}$, small perturbations in $\mathbf{b}$ will lead to small changes in $\mathbf{x}$. Such a system is said to be well-conditioned. However, if small perturbations in $\mathbf{b}$ lead to large changes in $\mathbf{x}$, the system is said to be ill-conditioned. The meaning of "small" and "large" changes depending on the context (e.g., it could be relative or absolute). The condition number quantifies this sensitivity.

To define the condition number, we consider a linear system $\mathbf{Ax} = \mathbf{b}$ with an exact solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. If we add a small perturbation (error) $\delta\mathbf{b}$ to $\mathbf{b}$, then the solution vector $\mathbf{x}$ will also change by some amount $\delta\mathbf{x}$ (which we think of as the error in $\mathbf{x}$ since the original perturbation is assumed to represent an error) so that the linear system remains satisfied:

$$\mathbf{A}\left(\mathbf{x} + \delta\mathbf{x}\right) = \mathbf{b} + \delta\mathbf{b} \implies \delta\mathbf{x} = \mathbf{A}^{-1}\delta\mathbf{b}$$

Taking the p-norm of both sides, we have

$$\|\delta\mathbf{x}\|_p = \|\mathbf{A}^{-1}\delta\mathbf{b}\|_p \leq \|\mathbf{A}^{-1}\|_p \|\delta\mathbf{b}\|_p$$

where the inequality follows from the submultiplicative property of matrix norms. We also have $\|\mathbf{b}\|_p = \|\mathbf{Ax}\|_p \leq \|\mathbf{A}\|_p \|\mathbf{x}\|_p$, hence $\|\mathbf{x}\|_p \geq \|\mathbf{b}\|_p / \|\mathbf{A}\|_p$. Dividing the two inequalities, we arrive at an expression for the relative error in the solution:

$$\frac{\|\delta\mathbf{x}\|_p}{\|\mathbf{x}\|_p} \leq \|\mathbf{A}\|_p \|\mathbf{A}^{-1}\|_p \frac{\|\delta\mathbf{b}\|_p}{\|\mathbf{b}\|_p}$$

This motivates the definition of the condition number of $\mathbf{A}$ as

$$\kappa_p(\mathbf{A}) = \|\mathbf{A}\|_p \|\mathbf{A}^{-1}\|_p$$

where we add a subscript p on κ to denote the norm used and to emphasize that its value depends on the choice of norm (but the qualitative behavior is the same across norms). If $\mathbf{A}$ is singular, $\kappa_p(\mathbf{A}) = \infty$. We note that clearly $\kappa_p(\mathbf{A}) \geq 1$ since $\|\mathbf{A}\|_p \|\mathbf{A}^{-1}\|_p \geq \|\mathbf{A}\mathbf{A}^{-1}\|_p = \|\mathbf{I}\|_p = 1$. Moreover, a matrix with a large condition number (an "ill-conditioned" matrix) indicates that the linear system is more sensitive to perturbations and numerical errors, while a matrix with a small condition number (a "well-conditioned" matrix) indicates greater numerical robustness. As a practical rule of thumb for backward-stable solvers (e.g., Gaussian elimination

with partial pivoting), the number of correct digits $\approx$ (working-precision digits) $-\log_{10}\kappa(\mathbf{A})$, and the relative forward error is typically $O(\kappa(\mathbf{A})\,\varepsilon_{\text{mach}})$. A small residual $\|\mathbf{r}\| = \|\mathbf{b} - \mathbf{A}\hat{\mathbf{x}}\|$ does not guarantee a small error when $\kappa(\mathbf{A})$ is large; roughly,

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \lesssim \kappa(\mathbf{A})\,\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}.$$

Let's take a look at the condition number for the 2-norm specifically. Here, the condition number is

$$\kappa_2(\mathbf{A}) = \|\mathbf{A}\|_2\|\mathbf{A}^{-1}\|_2 = \frac{\sigma_{\max}(\mathbf{A})}{\sigma_{\min}(\mathbf{A})}$$

where $\sigma_{\max}(\mathbf{A})$ and $\sigma_{\min}(\mathbf{A})$ are the largest and smallest singular values of $\mathbf{A}$, respectively. For $\mathbf{A} \succ 0$,[1] this is equivalent to

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$$

where $\lambda_{\max}(\mathbf{A})$ and $\lambda_{\min}(\mathbf{A})$ are the largest and smallest eigenvalues of $\mathbf{A}$, respectively.

Preconditioning When solving linear systems numerically, we often encounter ill-conditioned matrices. One way to mitigate the effects of ill-conditioning is through **preconditioning**. Preconditioning transforms $\mathbf{Ax} = \mathbf{b}$ into a system with more favorable spectral properties. Common forms are left $(\mathbf{P}^{-1}\mathbf{A})\mathbf{x} = \mathbf{P}^{-1}\mathbf{b}$, right $\mathbf{A}(\mathbf{P}^{-1}\mathbf{y}) = \mathbf{b}$ with $\mathbf{x} = \mathbf{P}^{-1}\mathbf{y}$, and symmetric $\mathbf{P}^{-1/2}\mathbf{A}\mathbf{P}^{-1/2}\mathbf{z} = \mathbf{P}^{-1/2}\mathbf{b}$ with $\mathbf{x} = \mathbf{P}^{-1/2}\mathbf{z}$. Examples include Jacobi (diagonal), incomplete LU/Cholesky (ILU/IC), and algebraic multigrid (AMG). The goal is to cluster eigenvalues and reduce the effective condition number to accelerate and stabilize iterative methods.

Even simpler than preconditioning is **scaling** (often called equilibration), which rescales rows and/or columns to reduce element-wise dynamic range and improve pivoting. A common heuristic is to equalize row/column ∞-norms. This helps numerics and pivoting but does not magically fix ill-conditioning rooted in the model; think nondimensionalization and consistent units (ChemE context: Pa vs bar, mol vs kmol).

Example: Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 10^{14} & 0 \\ 0 & 10^{-9} \end{bmatrix}$$

The condition number of this matrix with respect to the 2-norm is

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})} = \frac{10^{14}}{10^{-9}} = 10^{23}$$

which means that $\mathbf{A}$ is severely ill-conditioned. Such a high condition number implies that even tiny perturbations in $\mathbf{b}$ can lead to enormous errors in the computed solution $\mathbf{x}$.

To mitigate this issue, we can apply scaling by rescaling the rows of $\mathbf{A}$ to reduce the condition number. Specifically, we divide each row by the maximum absolute value in that row. For matrix $\mathbf{A}$, the scaling factors for the rows are 10^{14} and 10^{-9}, respectively. We define a diagonal scaling matrix $\mathbf{D}$ as

$$\mathbf{D} = \begin{bmatrix} 10^{-14} & 0 \\ 0 & 10^{9} \end{bmatrix}$$

Applying the scaling, we obtain the scaled matrix $\mathbf{A}_{\text{scaled}}$:

$$\mathbf{A}_{\text{scaled}} = \mathbf{DA} = \begin{bmatrix} 10^{-14} & 0 \\ 0 & 10^{9} \end{bmatrix}\begin{bmatrix} 10^{14} & 0 \\ 0 & 10^{-9} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The left-scaling forms the equivalent system $\mathbf{DAx} = \mathbf{Db}$ (same solution $\mathbf{x}$). Here, because $\mathbf{A}$ is diagonal, $\mathbf{DA} = \mathbf{I}$. In general, equilibration reduces dynamic range and can reduce κ, but need not drive it to 1.

[1] If $\mathbf{A}$ is normal ($\mathbf{AA}^* = \mathbf{A}^*\mathbf{A}$) over $\mathbb{C}$, then $\sigma_i(\mathbf{A}) = |\lambda_i(\mathbf{A})|$ and $\kappa_2(\mathbf{A}) = \max_i|\lambda_i(\mathbf{A})| / \min_i|\lambda_i(\mathbf{A})|$.

The condition number of the scaled matrix is now

$$\kappa_2(\mathbf{A}_{\text{scaled}}) = \frac{\lambda_{\max}(\mathbf{A}_{\text{scaled}})}{\lambda_{\min}(\mathbf{A}_{\text{scaled}})} = \frac{1}{1} = 1$$

which means that $\mathbf{A}_{\text{scaled}}$ is perfectly conditioned. This dramatic reduction shows how scaling can improve numerical behavior in favorable cases (in this case, $\mathbf{A}$ is diagonal, so $\mathbf{D} = \mathbf{A}^{-1}$ is an exact preconditioner, hence the perfect condition number). In general, scaling is often helpful for stability and pivoting, but it does not eliminate all ill-conditioning.

Warning: Preconditioning/scaling improves the matrix the algorithm sees (e.g., reduces $\kappa(P^{-1}A)$), which speeds and stabilizes iterations, but it *doesn't change the intrinsic sensitivity* of the original mapping $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ to perturbations in b. So it helps the solver's numerics, not the problem's conditioning.

2.9 Direct Methods

Direct methods are named for their ability to (given no round-off errors) determine the exact solution to a linear system within a finite number of steps. In practical applications, however, finite precision limits these methods to producing only approximate solutions.

Direct methods take advantage of the fact that we can easily solve the system Eq. 2.1 when $\mathbf{A}$ is upper triangular by using back substitution. Direct methods typically involve some variant of Gaussian elimination, which is a systematic procedure for transforming the coefficient matrix $\mathbf{A}$ into an upper triangular form in order to be able to solve the system by back substitution.

2.9.1 Back Substitution

Back substitution is a method for solving linear systems of equations that are in upper triangular form. The idea is to start with the last equation and solve for the last unknown, then use that result to solve for the second-to-last unknown, and so on (working backwards, hence the name). In mathematical terms, this is expressed as

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) \quad \text{for } i = n, n-1, \ldots, 1$$

where a_{ij} are the elements of the upper triangular matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, b_i are the elements of the vector $\mathbf{b}$, and x_i are the unknowns we are solving for. Notice that the formula for x_k for some $k < n$ relies on the values of $x_{k+1}, x_{k+2}, \ldots, x_n$, which is why we must work backwards.

Example: Consider the following linear system:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 13 \\ 6 \end{bmatrix}$$

Let's work backwards naively to gain some intuition. The last equation is $6x_3 = 6$, so $x_3 = 6/6 = 1$. The second-to-last equation is $4x_2 + 5(1) = 13$, so $x_2 = (13 - 5)/4 = 2$. Finally, the first equation is $x_1 + 2(2) + 3(1) = 1$, so $x_1 = 1 - 4 - 3 = -6$. Therefore, the solution to the system is $x_1 = -6$, $x_2 = 2$, and $x_3 = 1$.

We can also just directly apply the formula for back substitution (note that the formula is the exact same as the naive/intuitive approach given above, just written in a way that generalizes to larger systems):

$$x_3 = \frac{1}{6}(6 - 0) = 1$$
$$x_2 = \frac{1}{4}(13 - 5(1)) = 2$$
$$x_1 = \frac{1}{1}(1 - 2(2) - 3(1)) = -6$$

2.9.2 Gaussian Elimination

We've seen how easy it is to solve a linear system when the coefficient matrix is upper triangular, so how can we get **A** into this form? One way is via Gaussian elimination. This involves a series of elementary row operations (interchanging rows, multiplying a row by a scalar, and adding a multiple of one row to another) to eliminate the elements below the diagonal. The key linear algebraic fact that makes this possible is that elementary row operations do not change the solution to the linear system.

In general, the Gaussian elimination process begins by selecting the first pivot element a_{11} in the first row. If a_{11} is zero, a row swap is performed with a lower row that has a non-zero entry in the first column to avoid division by zero. Once the pivot is established, elementary row operations are applied to eliminate all entries below a_{11} in the first column. Specifically, for each row $k > 1$, the operation $\text{Row}_k \leftarrow \text{Row}_k - \lambda_{k1} \cdot \text{Row}_1$ is performed, where the multiplier $\lambda_{k1} = a_{k1}/a_{11}$.

After eliminating the first column below the pivot, the process moves to the next pivot a'_{22} in the second row and second column. Similar row operations are applied to eliminate all entries below a'_{22} in the second column, using multipliers $\lambda_{k2} = a'_{k2}/a'_{22}$ for $k > 2$. This procedure is repeated for each subsequent pivot until the augmented matrix is in upper triangular form:

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ 0 & a'_{22} & \cdots & a'_{2n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{nn} & b'_n \end{array}\right]$$

Once the matrix is upper triangular, we can use back substitution to solve for the unknowns.

Example: Consider the following linear system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \implies \left[\begin{array}{ccc|c} 0 & 2 & 1 & 4 \\ 1 & 3 & 2 & 5 \\ 2 & 1 & 3 & 6 \end{array}\right]$$

The first pivot element is $a_{11} = 0$. Since $a_{11} = 0$, we perform a row swap with a lower row that has a non-zero entry in the first column. Here, $a_{21} = 1 \neq 0$, so we swap Row 1 (R_1) with Row 2 (R_2):

$$R_1 \leftrightarrow R_2$$

The augmented matrix becomes

$$\left[\begin{array}{ccc|c} 1 & 3 & 2 & 5 \\ 0 & 2 & 1 & 4 \\ 2 & 1 & 3 & 6 \end{array}\right]$$

The first pivot is now $a_{11} = 1$. We eliminate the entries below a_{11} in the first column using the following

row operations:

$$\lambda_{31} = \frac{a_{31}}{a_{11}} = \frac{2}{1} = 2$$
$$R_3 \leftarrow R_3 - \lambda_{31} \cdot R_1 = R_3 - 2R_1$$

Performing the operation,

$$R_3 = \begin{bmatrix} 2 & 1 & 3 \mid 6 \end{bmatrix} - 2 \cdot \begin{bmatrix} 1 & 3 & 2 \mid 5 \end{bmatrix} = \begin{bmatrix} 0 & -5 & -1 \mid -4 \end{bmatrix}$$

The augmented matrix is now

$$\left[\begin{array}{ccc|c} 1 & 3 & 2 & 5 \\ 0 & 2 & 1 & 4 \\ 0 & -5 & -1 & -4 \end{array}\right]$$

The second pivot element is $a'_{22} = 2$. Since $a'_{22} \neq 0$, no row swap is needed. We proceed to eliminate the entry below a'_{22} in the second column:

$$\lambda_{32} = \frac{a'_{32}}{a'_{22}} = \frac{-5}{2} R_3 \leftarrow R_3 - \lambda_{32} \cdot R_2 = R_3 - \left(\frac{-5}{2}\right) R_2 = R_3 + \frac{5}{2} R_2$$

Performing the operation,

$$R_3 = \begin{bmatrix} 0 & -5 & -1 \mid -4 \end{bmatrix} + \frac{5}{2} \cdot \begin{bmatrix} 0 & 2 & 1 \mid 4 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.5 \mid 6 \end{bmatrix}$$

The augmented matrix is now

$$\left[\begin{array}{ccc|c} 1 & 3 & 2 & 5 \\ 0 & 2 & 1 & 4 \\ 0 & 0 & 1.5 & 6 \end{array}\right]$$

which is in upper triangular form. We can now use back substitution to solve for the unknowns.

Hopefully the above example elucidates the fact that Gaussian elimination is a very intuitive process, even if the formalism can seem a bit confusing at first. It is a very powerful tool for solving linear systems, and it is the foundation of many direct methods for solving linear systems.

Additionally, to increase numerical stability, **partial pivoting** is often used in practice. This involves selecting the pivot element as the largest element in the column (in magnitude) to avoid division by very small numbers, which can lead to numerical instability.

Gaussian Elimination Applications Gaussian elimination can be used for more than just solving linear systems. For example, it can be used to compute the inverse of a matrix by augmenting the matrix with the identity matrix and performing Gaussian elimination to transform the matrix into the identity matrix:

$$[\mathbf{A} \mid \mathbf{I}] \xrightarrow{\text{Gauss. elim.}} [\mathbf{I} \mid \mathbf{A}^{-1}]$$

Another application is determining the rank of a matrix. After reducing $\mathbf{A}$ to its row-echelon form using Gaussian elimination, the rank is equal to the number of non-zero rows.

Moreover, for a square matrix $\mathbf{A}$, Gaussian elimination can transform $\mathbf{A}$ into an upper triangular matrix $\mathbf{U}$. The determinant of $\mathbf{A}$ is then the product of the diagonal entries of $\mathbf{U}$ adjusted by the number of row swaps performed:

$$\det(\mathbf{A}) = (-1)^{\text{swaps}} \prod_{i=1}^{n} u_{ii}$$

Computational Complexity of Gaussian Elimination and Back Substitution The computational complexity of Gaussian elimination for an $n \times n$ matrix is $O(n^3)$. We can understand this as follows: We

have to perform one division, $n-i+1$ multiplications, and $n-i+1$ subtractions to form the j-th row at the i-th stage. There are $n-i$ rows in total. Therefore, at the i-th stage, we must perform $n-i$ divisions, $(n-i)(n-i+1)$ multiplications, and $(n-i)(n-i+1)$ subtractions. Summing these operations over all stages, we have to perform in total

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

divisions and

$$\sum_{i=1}^{n-1}(n-i)(n-i+1) = \frac{n(n^2-1)}{3}$$

multiplications and subtractions (each). This results in an overall computational complexity of $O(n^3)$ operations.

For back substitution, for each of the n variables, we perform one division and $n-i$ multiplications and subtractions to solve for x_i. Consequently, we perform n divisions and

$$\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$$

multiplications and subtractions (each). This results in an overall computational complexity of $O(n^2)$ operations.

Therefore, the overall computational complexity of solving a linear system with n equations and n unknowns using Gaussian elimination and back substitution is $O(n^3)$. This is a very important result to keep in mind when working with large linear systems, as the computational cost grows cubically with the size of the system.

2.10 Iterative Methods

Iterative methods attempt to solve linear systems ($\mathbf{Ax} = \mathbf{b}$) by generating a sequence of improving approximate solutions. They do not require Gaussian elimination and generally scale better for large systems. However, even with infinite precision, iterative methods can only obtain an approximate solution in a finite number of steps[2]. We cover Jacobi and Gauss-Seidel iteration, two of the most common iterative methods, in this course. But one more extremely popular method deserves at least a mention: the conjugate gradient method.[3]

2.10.1 Jacobi Iteration

Jacobi iteration is an iterative method for solving linear systems. Let's build some intuition for this method before giving the matrix form of the update formula. Suppose we have the linear system

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n} &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n} &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn} &= b_n \end{aligned}$$

and some initial approximation/guess for the solution, $x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$. How can we improve this guess? Well, rewriting the i-th equation in terms of the other variables, we have

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j\right)$$

[2]Actually, some iterative methods, like CG on SPD, are finite-termination in exact arithmetic, but we aren't going to cover that in this course.

[3]A good resource for this is Prof. Jonathan Shewchuk's guide *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.*[1]

If this equation is satisfied, then the i-th equation of the linear system is also satisfied. So, to improve our guess for x_i, we can use the current values of the other variables to estimate the value of x_i. After doing this for each variable (which we can do in parallel—we only update the values of the variables *after* we have computed the new values for all of them), we have a new approximation for the solution. We can then repeat this process until the solution converges (assuming that it does—more on this later). This is the main idea behind Jacobi iteration.

Warning: One might be tempted to think that after one iteration, we have the exact solution for x_i (after all, we solve each i-th equation exactly, right?). You should make sure you completely understand why this is not the case before moving on. The Jacobi method is an iterative method, meaning that we must repeat this process multiple times to get a better approximation for the solution.

Matrix Formulation Now, let's put this in matrix form, which is much more convenient for computer implementation. We decompose the coefficient matrix $\mathbf{A}$ into its diagonal component $\mathbf{D}$ and the remainder $\mathbf{R}$, such that $\mathbf{A} = \mathbf{D} + \mathbf{R}$. Here, $\mathbf{D}$ consists of the diagonal elements of $\mathbf{A}$, and $\mathbf{R}$ contains the off-diagonal elements. Thus, the linear system can be rewritten as

$$(\mathbf{D} + \mathbf{R})\,\mathbf{x} = \mathbf{b} \implies \mathbf{D}\mathbf{x} = -\mathbf{R}\mathbf{x} + \mathbf{b}$$

This defines the iterative map

$$\mathbf{D}\mathbf{x}^{(i+1)} = -\mathbf{R}\mathbf{x}^{(i)} + \mathbf{b}$$

Assuming that $\mathbf{D}$ is nonsingular, this can be equivalently written as

$$\mathbf{x}^{(i+1)} = -\mathbf{D}^{-1}(\mathbf{R}\mathbf{x}^{(i)} - \mathbf{b})$$

This formulation allows us to compute the next approximation $\mathbf{x}^{(i+1)}$ based on the current approximation $\mathbf{x}^{(i)}$. Note how this is the *exact* same formula as the one we derived earlier for the i-th variable in the linear system, just written in matrix form so that we can update all variables simultaneously!

As a quick note on notation, many sources commonly split the matrix $\mathbf{A}$ into the sum of a lower triangular matrix $\mathbf{L}$, a diagonal matrix $\mathbf{D}$, and an upper triangular matrix $\mathbf{U}$ (i.e., $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$). The Jacobi method is then written as

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

This is mathematically equivalent to the formulation we have presented here; don't be confused if you see this alternate notation.

Computational Complexity Each iteration of the Jacobi method involves diagonal matrix inversion, matrix-vector multiplications, and vector updates. Specifically, computing $\mathbf{R}\mathbf{x}^{(i)}$ requires $O(n^2)$ operations for an $n \times n$ matrix $\mathbf{A}$, and the subsequent vector operations also scale quadratically. Inverting a diagonal matrix is trivial and requires only n divisions. Therefore, the computational complexity of each Jacobi iteration is $O(n^2)$.

2.10.2 Convergence of Jacobi Iteration

We have seen that we can generate a sequence of improving approximations $\{\mathbf{x}^{(i)}\}$ to the solution of a linear system using the Jacobi method. But how do we know that this sequence converges at all, and if it converges, that it converges to the true solution $\mathbf{x}^\star$? Convergence is not generally guaranteed for iterative methods, let alone convergence to the true solution.

Convergence to the Correct Solution We claim that if the Jacobi sequence $\{\mathbf{x}^{(i)}\}$ converges, it indeed converges to the correct solution. To see this, we take the limit on both sides of the iterative equation:

$$\lim_{i\to\infty} \mathbf{x}^{(i+1)} = \lim_{i\to\infty} -\mathbf{D}^{-1}(\mathbf{R}\mathbf{x}^{(i)} - \mathbf{b})$$
$$\mathbf{x}^\star = -\mathbf{D}^{-1}(\mathbf{R}\mathbf{x}^\star - \mathbf{b})$$

Rearranging terms gives

$$\mathbf{D}\mathbf{x}^\star = -\mathbf{R}\mathbf{x}^\star + \mathbf{b} \quad \implies \quad \mathbf{A}\mathbf{x}^\star = \mathbf{b}$$

Thus, the limit $\mathbf{x}^\star$ satisfies the original system, so convergence implies correctness.

Conditions for Convergence So if we converge, we converge to the correct solution. But how do we know if we converge at all? Let's derive the conditions for convergence. We consider the exact solution satisfying

$$\mathbf{A}\mathbf{x}^\star = \mathbf{b} \iff \mathbf{x}^\star = -\mathbf{D}^{-1}\left(\mathbf{R}\mathbf{x}^\star - \mathbf{b}\right)$$

which means that the error at the $(i+1)$-th iteration is given by (note: the symbol $\stackrel{\text{def}}{=}$ means "defined as")

$$\begin{aligned}\mathbf{e}^{(i+1)} \stackrel{\text{def}}{=} \mathbf{x}^{(i+1)} - \mathbf{x}^* &= -\mathbf{D}^{-1}\mathbf{R}\left(\mathbf{x}^{(i)} - \mathbf{x}^\star\right) \\ &= -\mathbf{D}^{-1}\mathbf{R}\mathbf{e}^{(i)}\end{aligned}$$

Notice how this equation is set up; the error at the $(i+1)$-th iteration is a scaled version of the error at the preceding iteration. So if our initial guess is $\mathbf{x}^{(0)}$, we can keep applying this formula up to the i-th iteration to get

$$\mathbf{e}^{(i)} = \left(-\mathbf{D}^{-1}\mathbf{R}\right)^i \mathbf{e}^{(0)}$$

This is a very powerful result. To see why, let's use the eigendecomposition[4] of $-\mathbf{D}^{-1}\mathbf{R}$ to write

$$\begin{aligned}\mathbf{e}^{(i)} &= \left(-\mathbf{D}^{-1}\mathbf{R}\right)^i \mathbf{e}^{(0)} \\ &= \left(\mathbf{W}\mathbf{\Lambda}\mathbf{W}^{-1}\right)^i \mathbf{e}^{(0)} && \text{(eigendecomposition)} \\ &= \left(\mathbf{W}\mathbf{\Lambda}\mathbf{W}^{-1}\right)\left(\mathbf{W}\mathbf{\Lambda}\mathbf{W}^{-1}\right)\cdots\left(\mathbf{W}\mathbf{\Lambda}\mathbf{W}^{-1}\right)\mathbf{e}^{(0)} && \text{(expand power)} \\ &= \mathbf{W}\mathbf{\Lambda}\left(\mathbf{W}^{-1}\mathbf{W}\right)\cdots\mathbf{\Lambda}\left(\mathbf{W}^{-1}\mathbf{W}\right)\mathbf{\Lambda}\mathbf{W}^{-1}\mathbf{e}^{(0)} && \text{(exploit associativity)} \\ &= \mathbf{W}\mathbf{\Lambda}\left(\mathbf{I}\right)\cdots\mathbf{\Lambda}\left(\mathbf{I}\right)\mathbf{\Lambda}\mathbf{W}^{-1}\mathbf{e}^{(0)} && \text{(definition of inverse)} \\ &= \mathbf{W}\mathbf{\Lambda}^i\mathbf{W}^{-1}\mathbf{e}^{(0)}\end{aligned}$$

Thus, if any of the eigenvalues of $-\mathbf{D}^{-1}\mathbf{R}$ have magnitude greater than 1, the error will grow with i, and the Jacobi method will not converge. On the other hand, if all the eigenvalues have magnitude less than 1, the error will decay with i, and the Jacobi method will converge. And if any eigenvalues have magnitude equal to 1, the error will either remain constant or oscillate. In linear algebraic terms, this is captured by the notion of the **spectral radius** ρ of $-\mathbf{D}^{-1}\mathbf{R}$, which is the maximum magnitude of the eigenvalues of $-\mathbf{D}^{-1}\mathbf{R}$. Indeed, since we also know that convergence implies correctness, we have that

$$\rho(-\mathbf{D}^{-1}\mathbf{R}) < 1 \implies \|\mathbf{x}^{(i)} - \mathbf{x}^\star\|_p \to 0 \quad \text{as } i \to \infty$$

or, in words, the Jacobi method converges to the correct solution if the spectral radius of $-\mathbf{D}^{-1}\mathbf{R}$ is less than 1.

The most natural question to ask is when the spectral radius of $-\mathbf{D}^{-1}\mathbf{R}$ is less than 1. The answer is that this is true if $\mathbf{A}$ (recall that $\mathbf{A} = \mathbf{D} + \mathbf{R}$) is diagonally dominant. A matrix is diagonally dominant if for every row i, the diagonal element a_{ii} is greater in magnitude than the sum of the magnitudes of the off-diagonal elements in that row. In mathematical terms, this is written as

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Rate of Convergence The speed at which the Jacobi iteration converges to the solution $\mathbf{x}^\star$ is characterized by the spectral radius $\alpha = \rho(-\mathbf{D}^{-1}\mathbf{R})$. The error after i iterations can be approximated as

$$\|\mathbf{x}^{(i)} - \mathbf{x}^\star\|_p \approx O(\alpha^i)\|\mathbf{x}_0 - \mathbf{x}^\star\|_p$$

[4] If the matrix is not diagonalizable, analogous bounds via Jordan form or induced norms still give that $\rho < 1$ is necessary and sufficient for linear convergence.

Here, α is referred to as the **rate of convergence**. To achieve a desired accuracy ϵ, the number of iterations required satisfies

$$\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^\star\|_p}{\|\mathbf{x}_0 - \mathbf{x}^\star\|_p} \leq \epsilon \quad \Longrightarrow \quad i \geq \frac{\log(\epsilon)}{\log(\alpha)}$$

A smaller α indicates faster convergence. However, the Jacobi method can be relatively slow compared to other iterative methods like Gauss-Seidel or successive over-relaxation (SOR, not covered in this course), especially when α is close to 1.

2.10.3 Gauss-Seidel Iteration

Gauss-Seidel iteration is another iterative method for solving linear systems, similar to the Jacobi method but often exhibiting faster convergence. Like Jacobi iteration, it builds a sequence of approximations to the solution, but it uses the most recently updated values within each iteration to enhance convergence speed.

Consider the same linear system as before:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n} &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n} &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn} &= b_n \end{aligned}$$

and an initial guess for the solution, $x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$. To improve this guess, Gauss-Seidel iteration updates each variable sequentially, immediately using the latest available values. Specifically, the update rule for the i-th variable is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right)$$

for $i = 1, 2, \ldots, n$, where k denotes the current iteration. This means that when computing $x_i^{(k+1)}$, the most recent values $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are already updated, whereas $x_{i+1}^{(k)}, \ldots, x_n^{(k)}$ remain from the previous iteration. You should convince yourself that this update rule makes sense and compare it to the Jacobi update rule before proceeding.

> **Warning**: Unlike the Jacobi method, Gauss-Seidel updates variables sequentially and relies on the latest available estimates within the same iteration. This sequential dependency can lead to faster convergence but may complicate parallel implementations.

Matrix Formulation To express Gauss-Seidel iteration in matrix form, we decompose the coefficient matrix $\mathbf{A}$ into its lower triangular component $\mathbf{L}$, diagonal component $\mathbf{D}$, and upper triangular component $\mathbf{U}$, such that

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

where $\mathbf{L}$ contains the strictly lower triangular elements of $\mathbf{A}$, $\mathbf{D}$ is the diagonal matrix of $\mathbf{A}$, and $\mathbf{U}$ contains the strictly upper triangular elements of $\mathbf{A}$. The linear system can then be rewritten as

$$(\mathbf{D} + \mathbf{L})\mathbf{x} = -\mathbf{U}\mathbf{x} + \mathbf{b}$$

This defines the iterative map for Gauss-Seidel:

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

Assuming $\mathbf{D} + \mathbf{L}$ is nonsingular, we can solve for $\mathbf{x}^{(k+1)}$ to get

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

This formulation allows for sequential updates of the solution vector, utilizing the latest available estimates within each iteration. This is exactly the same as the update rule we derived earlier, just written in matrix form for convenience.

Computational Complexity Each Gauss-Seidel iteration involves solving a lower triangular system due to the $\mathbf{L} + \mathbf{D}$ term. Luckily, we can do this with forward substitution with $O(n^2)$ operations, so the computational complexity remains similar to Jacobi. That is, it is $O(n^2)$ operations per iteration for an $n \times n$ matrix $\mathbf{A}$.

2.10.4 Convergence of Gauss-Seidel Iteration

As with the Jacobi method, the convergence of Gauss-Seidel iteration is not universally guaranteed. However, under certain conditions, convergence can be assured, and when it does converge, it does so to the correct solution.

Convergence to the Correct Solution Assuming the Gauss-Seidel sequence $\{\mathbf{x}^{(k)}\}$ converges to a limit $\mathbf{x}^\star$, we can take the limit on both sides of the iterative equation:

$$\begin{aligned}\lim_{k\to\infty}(\mathbf{D}+\mathbf{L})\mathbf{x}^{(k+1)} &= \lim_{k\to\infty}\left(-\mathbf{U}\mathbf{x}^{(k)}+\mathbf{b}\right)\\ (\mathbf{D}+\mathbf{L})\mathbf{x}^\star &= -\mathbf{U}\mathbf{x}^\star+\mathbf{b}\\ \mathbf{A}\mathbf{x}^\star &= \mathbf{b}\end{aligned}$$

Thus, the limit $\mathbf{x}^\star$ satisfies the original linear system, meaning that convergence implies correctness.

Conditions for Convergence To determine when the Gauss-Seidel method converges, we analyze the error propagation similarly to Jacobi iteration. Let $\mathbf{e}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^\star$ denote the error at the k-th iteration. The iterative relation for the error becomes

$$\begin{aligned}(\mathbf{D}+\mathbf{L})\mathbf{e}^{(k+1)} &= -\mathbf{U}\mathbf{e}^{(k)}\\ \mathbf{e}^{(k+1)} &= -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}\mathbf{e}^{(k)}\end{aligned}$$

Letting $\mathbf{G} = -(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}$, we have

$$\mathbf{e}^{(k+1)} = \mathbf{G}\mathbf{e}^{(k)}$$

By iterating, the error after k iterations is

$$\mathbf{e}^{(k)} = \mathbf{G}^k\mathbf{e}^{(0)}$$

Using the eigendecomposition of $\mathbf{G}$, similar to the Jacobi method, we find that the spectral radius $\rho(\mathbf{G})$ governs convergence:

$$\rho(\mathbf{G}) < 1 \implies \|\mathbf{x}^{(k)} - \mathbf{x}^\star\|_p \to 0 \quad \text{as } k \to \infty$$

When is $\rho(\mathbf{G}) < 1$? If the matrix $\mathbf{A}$ is strictly diagonally dominant, i.e.,

$$|a_{ii}| > \sum_{j\neq i}|a_{ij}| \quad \forall i$$

then the spectral radius $\rho(\mathbf{G}) < 1$, ensuring convergence of the Gauss-Seidel method. Additionally, if $\mathbf{A}$ is symmetric positive definite, convergence is also guaranteed.

Rate of Convergence The rate at which Gauss-Seidel iteration converges is determined by the spectral radius $\alpha = \rho(\mathbf{G})$. The error after k iterations can be approximated by

$$\|\mathbf{x}^{(k)} - \mathbf{x}^\star\|_p \approx O(\alpha^k)\|\mathbf{x}^{(0)} - \mathbf{x}^\star\|_p$$

Since $\alpha < 1$ for convergent systems, a smaller α results in faster convergence. Often, it is empirically true that Gauss-Seidel converges faster than Jacobi because it effectively incorporates more up-to-date information within each iteration. However, like Jacobi, the convergence rate depends on the properties of $\mathbf{A}$, and for systems with a spectral radius close to 1, convergence can still be slow.

Comparison with Jacobi Iteration While both methods are simple and easy to implement, Gauss-Seidel typically outperforms Jacobi in terms of convergence speed due to its sequential update strategy. However, this comes at the cost of reduced parallelizability, as each update depends on the results of the previous ones within the same iteration. In contrast, Jacobi's parallel updates make it more suitable for parallel computing environments despite its slower convergence.

In practice, Gauss-Seidel is preferred over Jacobi when faster convergence is desired and when the system structure permits sequential updates without significant computational overhead. For large-scale systems where parallel computation is essential, Jacobi may still be advantageous despite its slower convergence rate.

2.10.5 Condition Number and Convergence Rate

To understand the impact of the condition number on the convergence rate of iterative methods, we consider the Richardson method, an iterative method defined by the update rule

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}^{(i)})$$

where ω is a relaxation parameter. The choice of ω strongly affects the convergence behavior of the method. Assuming that $\mathbf{A}$ is symmetric positive definite, the optimal convergence rate for the Richardson method is achieved with the choice

$$\omega_{\text{opt}} = \frac{2}{\lambda_{\min}(\mathbf{A}) + \lambda_{\max}(\mathbf{A})}$$

With this optimal ω, the spectral radius of the iteration matrix $\mathbf{I} - \omega_{\text{opt}}\mathbf{A}$ becomes

$$\rho(\mathbf{I} - \omega_{\text{opt}}\mathbf{A}) = \frac{\kappa(\mathbf{A}) - 1}{\kappa(\mathbf{A}) + 1}$$

This expression shows that the spectral radius—and consequently the convergence rate—is directly related to the condition number $\kappa(\mathbf{A})$. Specifically, a smaller $\kappa(\mathbf{A})$ leads to a smaller spectral radius, resulting in faster convergence of the iterative method. Conversely, a large condition number implies that the spectral radius is closer to one, which can significantly slow down convergence.

This relationship (not the literal equation, but the idea we just talked about) between the condition number and the convergence rate is not unique to the Richardson method but holds generally for a wide class of iterative methods, including Jacobi and Gauss-Seidel iterations. Therefore, we should make sure that the coefficient matrix $\mathbf{A}$ is well-conditioned in order to achieve efficiency and reliability of these iterative solvers. If $\kappa(\mathbf{A})$ is large, it may be beneficial to precondition the system to reduce the condition number and improve the convergence rate of the iterative method.

Warning: While a lower condition number generally promotes faster convergence, it is important to recognize that not all matrices can be easily preconditioned to achieve a desirable condition number. Additionally, some iterative methods may exhibit different sensitivities to the condition number, so the choice of method should consider both the properties of $\mathbf{A}$ and the specific requirements of the problem at hand.

References

(1) Shewchuk, J. R. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*; tech. rep.; USA, 1994.