The Little Book of Linear Algebra

Version 0.3.0

Duc-Tam Nguyen

2025-09-26

Table of contents

Content

26. Detecting inconsistency (when no solution exists)
27. Gaussian elimination by hand (a disciplined procedure)
28. Back substitution and solution sets (finishing cleanly)
29. Rank and its first meaning (pivots as information)
30. LU factorization (elimination captured as L and U)

Chapter 4. Vector Spaces and Subspaces

31. Axioms of vector spaces (what "space" really means)
32. Subspaces, column space, and null space (where solutions live)
33. Span and generating sets (coverage of a space)
34. Linear independence and dependence (no redundancy vs. redundancy)
35. Basis and coordinates (naming every vector uniquely)
36. Dimension (how many directions)
37. Rank–nullity theorem (dimensions that add up)
38. Coordinates relative to a basis (changing the "ruler")
39. Change-of-basis matrices (moving between coordinate systems)
40. Affine subspaces (lines and planes not through the origin)

Chapter 5. Linear Transformations and Structure

41. Linear transformations (preserving lines and sums)
42. Matrix representation of a linear map (choosing a basis)
43. Kernel and image (inputs that vanish; outputs we can reach)
44. Invertibility and isomorphisms (perfectly reversible maps)
45. Composition, powers, and iteration (doing it again and again)
46. Similarity and conjugation (same action, different basis)
47. Projections and reflections (idempotent and involutive maps)
48. Rotations and shear (geometric intuition)
49. Rank and operator viewpoint (rank beyond elimination)
50. Block matrices and block maps (divide and conquer structure)

Chapter 6. Determinants and Volume

51. Areas, volumes, and signed scale factors (geometric entry point)
52. Determinant via linear rules (multilinearity, sign, normalization)
53. Determinant and row operations (how each move changes det)
54. Triangular matrices and product of diagonals (fast wins)
55. $\det(AB) = \det(A)\det(B)$ (multiplicative magic)
56. Invertibility and zero determinant (flat vs. full volume)
57. Cofactor expansion (Laplace's method)

58. Permutations and sign (the combinatorial core)
59. Cramer's rule (solving with determinants, and when not to use it)
60. Computing determinants in practice (use LU, mind stability)

Chapter 7. Eigenvalues, Eigenvectors, and Dynamics

61. Eigenvalues and eigenvectors (directions that stay put)
62. Characteristic polynomial (where eigenvalues come from)
63. Algebraic vs. geometric multiplicity (how many and how independent)
64. Diagonalization (when a matrix becomes simple)
65. Powers of a matrix (long-term behavior via eigenvalues)
66. Real vs. complex spectra (rotations and oscillations)
67. Defective matrices and a peek at Jordan form (when diagonalization fails)
68. Stability and spectral radius (grow, decay, or oscillate)
69. Markov chains and steady states (probabilities as linear algebra)
70. Linear differential systems (solutions via eigen-decomposition)

Chapter 8. Orthogonality, Least Squares, and QR

71. Inner products beyond dot product (custom notions of angle)
72. Orthogonality and orthonormal bases (perpendicular power)
73. Gram–Schmidt process (constructing orthonormal bases)
74. Orthogonal projections onto subspaces (closest point principle)
75. Least-squares problems (fit when exact solve is impossible)
76. Normal equations and geometry of residuals (why it works)
77. QR factorization (stable least squares via orthogonality)
78. Orthogonal matrices (length-preserving transforms)
79. Fourier viewpoint (expanding in orthogonal waves)
80. Polynomial and multifeature least squares (fitting more flexibly)

Chapter 9. SVD, PCA, and Conditioning

81. Singular values and SVD (universal factorization)
82. Geometry of SVD (rotations + stretching)
83. Relation to eigen-decompositions (ATA and AAT)
84. Low-rank approximation (best small models)
85. Principal component analysis (variance and directions)
86. Pseudoinverse (Moore–Penrose) and solving ill-posed systems
87. Conditioning and sensitivity (how errors amplify)
88. Matrix norms and singular values (measuring size properly)
89. Regularization (ridge/Tikhonov to tame instability)

90. Rank-revealing QR and practical diagnostics (what rank really is)

Chapter 10. Applications and Computation

91. 2D/3D geometry pipelines (cameras, rotations, and transforms)
92. Computer graphics and robotics (homogeneous tricks in action)
93. Graphs, adjacency, and Laplacians (networks via matrices)
94. Data preprocessing as linear ops (centering, whitening, scaling)
95. Linear regression and classification (from model to matrix)
96. PCA in practice (dimensionality reduction workflow)
97. Recommender systems and low-rank models (fill the missing entries)
98. PageRank and random walks (ranking with eigenvectors)
99. Numerical linear algebra essentials (floating point, BLAS/LAPACK)
100. Capstone problem sets and next steps (a roadmap to mastery)

The Book

Overtune

A soft opening, an invitation into the world of vectors and spaces, where each step begins a journey.

1. Geometry's Dawn

```
Lines cross in silence,
planes awaken with order,
numbers sketch the world.
```

2. Invitation to Learn

```
Steps begin with dots,
arrows stretch into new paths,
the journey unfolds.
```

3. Light and Shadow

```
Shadows fall on grids,
hidden shapes emerge in form,
clarity takes root.
```

4. The Seed of Structure

```
One point, then a line,
spaces blossom out from rules,
infinity grows.
```

5. Whisper of Algebra

Silent rules of space,
woven threads of thought align,
order sings through time.

6. Beginner's Welcome

Empty page awaits,
axes cross like guiding hands,
first steps find their place.

7. Eternal Path

From vectors to stars,
equations trace destiny,
patterns guide our sight.

Chapter 1. Vectors, scalars, and geometry

Opening

```
Arrows in the air,
directions whisper softly-
the plane comes alive.
```

1. Scalars, Vectors, and Coordinate Systems

When we begin learning linear algebra, everything starts with the simplest building blocks: scalars and vectors. A scalar is just a single number, like 3, –7, or . It carries only magnitude and no direction. Scalars are what we use for counting, measuring length, or scaling other objects up and down. A vector, by contrast, is an ordered collection of numbers. You can picture it as an arrow pointing somewhere in space, or simply as a list like (2, 5) in 2D or (1, –3, 4) in 3D. Where scalars measure "how much," vectors measure both "how much" and "which way."

Coordinate Systems

To talk about vectors, we need a coordinate system. Imagine laying down two perpendicular axes on a sheet of paper: the x-axis (left to right) and the y-axis (up and down). Every point on the sheet can be described with two numbers: how far along the x-axis, and how far along the y-axis. This pair of numbers is a vector in 2D. Add a z-axis pointing up from the page, and you have 3D space. Each coordinate system gives us a way to describe vectors numerically, even though the underlying "space" is the same.

Visualizing Scalars vs. Vectors

- A scalar is like a single tick mark on a ruler.
- A vector is like an arrow that starts at the origin (0, 0, …) and ends at the point defined by its components. For example, the vector (3, 4) in 2D points from the origin to the point 3 units along the x-axis and 4 units along the y-axis.

Why Start Here?

Understanding the difference between scalars and vectors is the foundation for everything else in linear algebra. Every concept-matrices, linear transformations, eigenvalues-eventually reduces to how we manipulate vectors and scale them with scalars. Without this distinction, the rest of the subject would have no anchor.

Why It Matters

Nearly every field of science and engineering depends on this idea. Physics uses vectors for velocity, acceleration, and force. Computer graphics uses them to represent points, colors, and transformations. Data science treats entire datasets as high-dimensional vectors. By mastering scalars and vectors early, you unlock the language in which modern science and technology are written.

Try It Yourself

1. Draw an x- and y-axis on a piece of paper. Plot the vector (2, 3).
2. Now draw the vector (–1, 4). Compare their directions and lengths.
3. Think: which of these two vectors points "more upward"? Which is "longer"?

These simple experiments already give you intuition for the operations you'll perform again and again in linear algebra.

2. Vector Notation, Components, and Arrows

Linear algebra gives us powerful ways to describe and manipulate vectors, but before we can do anything with them, we need a precise notation system. Notation is not just cosmetic-it tells us how to read, write, and think about vectors clearly and unambiguously. In this section, we'll explore how vectors are written, how their components are represented, and how we can interpret them visually as arrows.

Writing Vectors

Vectors are usually denoted by lowercase letters in bold (like $\mathbf{v}, \mathbf{w}, \mathbf{x}$)
or with an arrow overhead (like $\vec{v}$).

For instance, the vector $\mathbf{v} = (2, 5)$ is the same as $\vec{v} = (2, 5)$.

The style depends on context: mathematicians often use bold, physicists often use arrows.
In handwritten notes, people sometimes underline vectors (e.g., $\underline{v}$) to avoid confusion with scalars.

The important thing is to distinguish vectors from scalars at a glance.

Components of a Vector

A vector in two dimensions has two components, written as (x, y).
In three dimensions, it has three components: (x, y, z).
More generally, an n-dimensional vector has n components: $(v_1, v_2, \dots, v_n)$.
Each component tells us how far the vector extends along one axis of the coordinate system.

For example:

- $\mathbf{v} = (3, 4)$ means the vector extends 3 units along the x-axis and 4 units along the y-axis.
- $\mathbf{w} = (-2, 0, 5)$ means the vector extends -2 units along the x-axis, 0 along the y-axis, and 5 along the z-axis.

We often refer to the i-th component of a vector $\mathbf{v}$ as v_i.
So, for $\mathbf{v} = (3, 4, 5)$, we have $v_1 = 3$, $v_2 = 4$, $v_3 = 5$.

Column vs. Row Vectors

Vectors can be written in two common ways:

- As a row vector: (v_1, v_2, v_3)

- As a column vector:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Both represent the same abstract object.
Row vectors are convenient for quick writing, while column vectors are essential when we start multiplying by matrices, because the dimensions must align.

Vectors as Arrows

The most intuitive way to picture a vector is as an arrow:

- It starts at the origin (0, 0, …).
- It ends at the point given by its components.

For example, the vector (2, 3) in 2D is drawn as an arrow from (0, 0) to (2, 3). The arrow has both direction (where it points) and magnitude (its length). This geometric picture makes abstract algebraic manipulations much easier to grasp.

Position Vectors vs. Free Vectors

There are two common interpretations of vectors:

1. Position vector - a vector that points from the origin to a specific point in space. Example: (2, 3) is the position vector for the point (2, 3).
2. Free vector - an arrow with length and direction, but not tied to a specific starting point. For instance, an arrow of length 5 pointing northeast can be drawn anywhere, but it still represents the same vector.

In linear algebra, we often treat vectors as free vectors, because their meaning does not depend on where they are drawn.

Example: Reading a Vector

Suppose u = (–3, 2).

- The first component (–3) means move 3 units left along the x-axis.
- The second component (2) means move 2 units up along the y-axis. So the arrow points to the point (–3, 2). Even without a diagram, the components tell us exactly what the arrow would look like.

Why It Matters

Clear notation is the backbone of linear algebra. Without it, equations quickly become unreadable, and intuition about direction and size is lost. The way we write vectors determines how easily we can connect the algebra (numbers and symbols) to the geometry (arrows and spaces). This dual perspective-symbolic and visual-is what makes linear algebra powerful and practical.

Try It Yourself

1. Write down the vector (4, –1). Draw it on graph paper.
2. Rewrite the same vector as a column vector.
3. Translate the vector (4, –1) by moving its starting point to (2, 3) instead of the origin. Notice that the arrow looks the same-it just starts elsewhere.
4. For a harder challenge: draw the 3D vector (2, –1, 3). Even if you can't draw perfectly in 3D, try to show each component along the x, y, and z axes.

By practicing both the notation and the arrow picture, you'll develop fluency in switching between abstract symbols and concrete visualizations. This skill will make every later concept in linear algebra far more intuitive.

3. Vector Addition and Scalar Multiplication

Once we know how to describe vectors with components and arrows, the next step is to learn how to combine them. Two fundamental operations form the backbone of linear algebra: adding vectors together and scaling vectors with numbers (scalars). These two moves, though simple, generate everything else we'll build later. With them, we can describe motion, forces, data transformations, and more.

Vector Addition in Coordinates

Suppose we have two vectors in 2D:
$\mathbf{u} = (u_1, u_2), \quad \mathbf{v} = (v_1, v_2).$

Their sum is defined as:

$$\mathbf{u} + \mathbf{v} = (u_1 + v_1,\ u_2 + v_2).$$

In words, you add corresponding components.
This works in higher dimensions too:

$$(u_1, u_2, \dots, u_n) + (v_1, v_2, \dots, v_n) = (u_1 + v_1,\ u_2 + v_2,\ \dots,\ u_n + v_n).$$

Example:

$$(2, 3) + (-1, 4) = (2 - 1,\ 3 + 4) = (1, 7).$$

Vector Addition as Geometry

The geometric picture is even more illuminating. If you draw vector u as an arrow, then place the tail of v at the head of u, the arrow from the start of u to the head of v is u + v. This is called the tip-to-tail rule. The parallelogram rule is another visualization: place u and v tail-to-tail, form a parallelogram, and the diagonal is their sum.

Example: u = (3, 1), v = (2, 2). Draw both from the origin. Their sum (5, 3) is exactly the diagonal of the parallelogram they span.

Scalar Multiplication in Coordinates

Scalars stretch or shrink vectors.
If $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ and c is a scalar, then:

$$c \cdot \mathbf{u} = (c \cdot u_1,\ c \cdot u_2,\ \ldots,\ c \cdot u_n).$$

Example:

$$2 \cdot (3, 4) = (6, 8).$$

$$(-1) \cdot (3, 4) = (-3, -4).$$

Multiplying by a positive scalar stretches or compresses the arrow while keeping the direction the same. Multiplying by a negative scalar flips the arrow to point the opposite way.

Scalar Multiplication as Geometry

Imagine the vector (1, 2). Draw it on graph paper: it goes right 1, up 2. Now double it: (2, 4). The arrow points in the same direction but is twice as long. Halve it: (0.5, 1). It's the same direction but shorter. Negate it: (–1, –2). Now the arrow points backward.

This geometric picture explains why we call these numbers "scalars": they scale the vector.

Combining Both: Linear Combinations

Vector addition and scalar multiplication are not just separate tricks-they combine to form the heart of linear algebra: linear combinations.

A linear combination of vectors u and v is any vector of the form
$a \cdot u + b \cdot v$, where a and b are scalars.

Example:
If $u = (1, 0)$ and $v = (0, 1)$, then
$3 \cdot u + 2 \cdot v = (3, 2)$.

This shows how any point on the grid can be reached by scaling and adding these two basic vectors.
That's the essence of constructing spaces.

Algebraic Properties

Vector addition and scalar multiplication obey rules that mirror arithmetic with numbers:

- Commutativity: $u + v = v + u$
- Associativity: $(u + v) + w = u + (v + w)$
- Distributivity over scalars: $c \cdot (u + v) = c \cdot u + c \cdot v$
- Distributivity over numbers: $(a + b) \cdot u = a \cdot u + b \cdot u$

These rules are not trivial bookkeeping - they guarantee that linear algebra behaves predictably, which is why it works as the language of science.

Why It Matters

With only these two operations-addition and scaling-you can already describe lines, planes, and entire spaces. Any system that grows by combining influences, like physics, economics, or machine learning, is built on these simple rules. Later, when we define matrix multiplication, dot products, and eigenvalues, they all reduce to repeated patterns of adding and scaling vectors.

Try It Yourself

1. Add (2, 3) and (–1, 4). Draw the result on graph paper.
2. Multiply (1, –2) by 3, and then add (0, 5). What is the final vector?
3. For a deeper challenge: Let u = (1, 2) and v = (2, –1). Sketch all vectors of the form a · u + b · v for integer values of a, b between –2 and 2. Notice the grid of points you create-that's the span of these two vectors.

This simple practice shows you how combining two basic vectors through addition and scaling generates a whole structured space, the first glimpse of linear algebra's real power.

4. Linear Combinations and Span

After learning to add vectors and scale them, the natural next question is: *what can we build from these two operations?* The answer is the concept of linear combinations, which leads directly to one of the most fundamental ideas in linear algebra: the span of a set of vectors. These ideas tell us not only what individual vectors can do, but how groups of vectors can shape entire spaces.

What Is a Linear Combination?

A linear combination is any vector formed by multiplying vectors with scalars and then adding the results together.

Formally, given vectors $v_1, v_2, \ldots, v_k$ and scalars $a_1, a_2, \ldots, a_k$, a linear combination looks like:

$$a_1 \cdot v_1 + a_2 \cdot v_2 + \cdots + a_k \cdot v_k.$$

This is nothing more than repeated addition and scaling, but the idea is powerful because it describes how vectors combine to generate new ones.

Example:
Let $u = (1, 0)$ and $v = (0, 1)$. Then any linear combination $a \cdot u + b \cdot v = (a, b)$.
This shows that every point in the 2D plane can be expressed as a linear combination of these two simple vectors.

Geometric Meaning

Linear combinations are about mixing directions and magnitudes. Each vector acts like a "directional ingredient," and the scalars control how much of each ingredient you use.

- With one vector: You can only reach points on a single line through the origin.
- With two non-parallel vectors in 2D: You can reach every point in the plane.
- With three non-coplanar vectors in 3D: You can reach all of 3D space.

This progression shows that the power of linear combinations depends not just on the vectors themselves but on how they relate to each other.

The Span of a Set of Vectors

The span of a set of vectors is the collection of all possible linear combinations of them.
It answers the question: *"What space do these vectors generate?"*

Notation:

$$\text{Span}\{v_1, v_2, \ldots, v_k\} = \{a_1 v_1 + a_2 v_2 + \cdots + a_k v_k \mid a_i \in \mathbb{R}\}.$$

Examples:

- $\text{Span}\{(1, 0)\}$ = all multiples of $(1, 0)$, which is the x-axis.

- Span$\{(1,0),(0,1)\}$ = all of $\mathbb{R}^2$, the entire plane.
- Span$\{(1,2),(2,4)\}$ = just the line through $(1,2)$, because the second vector is a multiple of the first.

So the span depends heavily on whether the vectors add new directions or just duplicate what's already there.

Parallel and Independent Vectors

If vectors point in the same or opposite directions (one is a scalar multiple of another), then their span is just a line. They don't add any new coverage of space. But if they point in different directions, they open up new dimensions. This leads to the critical idea of linear independence, which we'll explore later: vectors are independent if none of them is a linear combination of the others.

Visualizing Span in Different Dimensions

- In 2D:
  - One vector spans a line.
  - Two independent vectors span the whole plane.
- In 3D:
  - One vector spans a line.
  - Two independent vectors span a plane.
  - Three independent vectors span all of 3D space.
- In higher dimensions: The same pattern continues. A set of k independent vectors spans a k-dimensional subspace inside the larger space.

Algebraic Properties

- The span of vectors always includes the zero vector, because you can choose all scalars = 0.
- The span is always a subspace, meaning it's closed under addition and scalar multiplication. If you add two vectors in the span, the result stays in the span.
- The span grows when you add new independent vectors, but not if the new vector is just a combination of the old ones.

Why It Matters

Linear combinations and span are the foundation for almost everything else in linear algebra:

- They define what it means for vectors to be independent or dependent.
- They form the basis for solving linear systems (solutions are often described as spans).
- They explain how dimensions arise in vector spaces.
- They underpin practical methods like principal component analysis, where data is projected onto the span of a few important vectors.

In short, the span tells us the "reach" of a set of vectors, and linear combinations are the mechanism to explore that reach.

Try It Yourself

1. Take vectors (1, 0) and (0, 1). Write down three different linear combinations and plot them. What shape do you notice?
2. Try vectors (1, 2) and (2, 4). Write down three different linear combinations. Plot them. What's different from the previous case?
3. In 3D, consider (1, 0, 0) and (0, 1, 0). Describe their span. Add (0, 0, 1). How does the span change?
4. Challenge: Pick vectors (1, 2, 3) and (4, 5, 6). Do they span a plane or all of 3D space? How can you tell?

By experimenting with simple examples, you'll see clearly how the idea of span captures the richness or limitations of combining vectors.

5. Length (Norm) and Distance

So far, vectors have been arrows with direction and components. To compare them more meaningfully, we need ways to talk about how long they are and how far apart they are. These notions are formalized through the norm of a vector (its length) and the distance between vectors. These concepts tie together the algebra of components and the geometry of space.

The Length (Norm) of a Vector

The norm of a vector measures its magnitude, or how long the arrow is.
For a vector $v = (v_1, v_2, \ldots, v_n)$ in n-dimensional space, its norm is defined as:

$$\|v\| = \sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}.$$

This formula comes directly from the Pythagorean theorem: the length of the hypotenuse equals the square root of the sum of squares of the legs.
In 2D, this is the familiar distance formula between the origin and a point.

Examples:

- For $v = (3, 4)$:

$$\|v\| = \sqrt{3^2 + 4^2} = \sqrt{9 + 16} = 5.$$

- For $w = (1, -2, 2)$:

$$\|w\| = \sqrt{1^2 + (-2)^2 + 2^2} = \sqrt{1 + 4 + 4} = \sqrt{9} = 3.$$

Unit Vectors

A unit vector is a vector whose length is exactly 1.
These are important because they capture direction without scaling.
To create a unit vector from any nonzero vector, divide by its norm:

$$u = \frac{v}{\|v\|}.$$

Example:
For $v = (3, 4)$, the unit vector is

$$u = \left(\tfrac{3}{5}, \tfrac{4}{5}\right).$$

This points in the same direction as $(3, 4)$ but has length 1.

Unit vectors are like pure directions.
They're especially useful for projections, defining coordinate systems, and normalizing data.

Distance Between Vectors

The distance between two vectors u and v is defined as the length of their difference:

$$\text{dist}(u, v) = \|u - v\|.$$

Example:
Let $u = (2, 1)$ and $v = (5, 5)$. Then

$$u - v = (-3, -4).$$

Its norm is

$$\sqrt{(-3)^2 + (-4)^2} = \sqrt{9 + 16} = 5.$$

So the distance is 5. This matches our intuition: the straight-line distance between points $(2, 1)$ and $(5, 5)$.

Geometric Interpretation

- The norm tells you how far a point is from the origin.
- The distance tells you how far two points are from each other.

Both are computed with the same formula-the square root of sums of squares-but applied in slightly different contexts.

Different Kinds of Norms

The formula above defines the Euclidean norm (or ℓ_2 norm), the most common one.
But in linear algebra, other norms are also useful:

- ℓ_1 norm:

  $$\|v\|_1 = |v_1| + |v_2| + \cdots + |v_n|$$

  (sum of absolute values).

- ℓ_∞ norm:

  $$\|v\|_\infty = \max(|v_1|, |v_2|, \ldots, |v_n|)$$

  (largest component).

These norms change the geometry of "length" and "distance." For example, in the norm, the unit circle is shaped like a diamond; in the ∞ norm, it looks like a square.

Algebraic Properties

Norms and distances satisfy critical properties that make them consistent measures:

- Non-negativity: $\|v\| \geq 0$, and $\|v\| = 0$ only if $v = 0$.
- Homogeneity: $\|c \cdot v\| = |c| \, \|v\|$ (scaling affects length predictably).
- Triangle inequality: $\|u + v\| \leq \|u\| + \|v\|$ (the direct path is shortest).
- Symmetry (for distance): $\text{dist}(u, v) = \text{dist}(v, u)$.

These properties are why norms and distances are robust tools across mathematics.

Why It Matters

Understanding length and distance is the first step toward geometry in higher dimensions. These notions:

- Allow us to compare vectors quantitatively.
- Form the basis of concepts like angles, orthogonality, and projections.
- Underpin optimization problems (e.g., "find the closest vector" is central to machine learning).
- Define the geometry of spaces, which changes dramatically depending on which norm you use.

Try It Yourself

1. Compute the norm of (6, 8). Then divide by the norm to find its unit vector.
2. Find the distance between (1, 1, 1) and (4, 5, 6).
3. Compare the Euclidean and Manhattan ( ) distances between (0, 0) and (3, 4). Which one matches your intuition if you were walking along a city grid?
4. Challenge: For vectors u = (2, –1, 3) and v = (–2, 0, 1), compute ‖u – v‖. Then explain what this distance means geometrically.

By working through these examples, you'll see how norms and distances make abstract vectors feel as real as points and arrows you can measure in everyday life.

6. Dot Product (Algebraic and Geometric Views)

The dot product is one of the most fundamental operations in linear algebra. It looks like a simple formula, but it unlocks the ability to measure angles, detect orthogonality, project one vector onto another, and compute energy or work in physics. Understanding it requires seeing both the algebraic view (a formula on components) and the geometric view (a way to compare directions).

Algebraic Definition

For two vectors of the same dimension, $u = (u_1, u_2, \dots, u_n)$ and $v = (v_1, v_2, \dots, v_n)$, the dot product is defined as:

$$u \cdot v = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n.$$

This is simply multiplying corresponding components and summing the results.

Examples:

- $(2, 3) \cdot (4, 5) = (2 \times 4) + (3 \times 5) = 8 + 15 = 23$
- $(1, -2, 3) \cdot (0, 4, -1) = (1 \times 0) + (-2 \times 4) + (3 \times -1) = 0 - 8 - 3 = -11$

Notice that the dot product is always a scalar, not a vector.

Geometric Definition

The dot product can also be defined in terms of vector length and angle:

$$u \cdot v = \|u\| \, \|v\| \cos(\theta),$$

where θ is the angle between u and v ($0^\circ \leq \theta \leq 180^\circ$).

This formula tells us:

- If the angle is acute (less than 90°), $\cos(\theta) > 0$, so the dot product is positive.
- If the angle is right (exactly 90°), $\cos(\theta) = 0$, so the dot product is 0.
- If the angle is obtuse (greater than 90°), $\cos(\theta) < 0$, so the dot product is negative.

Thus, the sign of the dot product encodes directional alignment.

Connecting the Two Definitions

At first glance, the algebraic sum of products and the geometric length–angle formula seem unrelated. But they are equivalent. To see why, consider the law of cosines applied to a triangle formed by u, v, and u – v. Expanding both sides leads directly to the equivalence between the two formulas. This dual interpretation is what makes the dot product so powerful: it is both a computation rule and a geometric measurement.

Orthogonality

Two vectors are orthogonal (perpendicular) if and only if their dot product is zero:

$$u \cdot v = 0 \iff \theta = 90^\circ.$$

This gives us an algebraic way to check for perpendicularity without drawing diagrams.

Example:
$(2, 1) \cdot (-1, 2) = (2 \times -1) + (1 \times 2) = -2 + 2 = 0,$
so the vectors are orthogonal.

Projections

The dot product also provides a way to project one vector onto another.
The scalar projection of u onto v is:

$$\text{proj}_{\text{scalar}}(u \text{ onto } v) = \frac{u \cdot v}{\|v\|}.$$

The vector projection is then:

$$\text{proj}_{\text{vector}}(u \text{ onto } v) = \frac{u \cdot v}{\|v\|^2}\, v.$$

This allows us to decompose vectors into “parallel” and “perpendicular” components, which is central in geometry, physics, and data analysis.

Examples

1. Compute $u = (3, 4)$ and $v = (4, 3)$.
   - Dot product: $(3 \times 4) + (4 \times 3) = 12 + 12 = 24$.
   - Norms: $\|u\| = 5$, $\|v\| = 5$.
   - $\cos(\theta) = \frac{24}{5 \times 5} = \frac{24}{25} \approx 0.96$, so $\theta \approx 16^\circ$.
     These vectors are nearly parallel.
2. Compute $u = (1, 2, -1)$ and $v = (2, -1, 1)$.
   - Dot product: $(1 \times 2) + (2 \times -1) + (-1 \times 1) = 2 - 2 - 1 = -1$.
   - Norms: $\|u\| = \sqrt{6}$, $\|v\| = \sqrt{6}$.
   - $\cos(\theta) = \frac{-1}{\sqrt{6} \times \sqrt{6}} = -\frac{1}{6}$, so $\theta \approx 99.6^\circ$.
     Slightly obtuse.

Physical Interpretation

In physics, the dot product computes work:

$$\text{Work} = \text{Force} \cdot \text{Displacement} = \|\text{Force}\| \, \|\text{Displacement}\| \cos(\theta).$$

Only the component of the force in the direction of motion contributes. If you push straight down on a box while trying to move it horizontally, the dot product is zero: no work is done in the direction of motion.

Algebraic Properties

- Commutative: $u \cdot v = v \cdot u$
- Distributive: $u \cdot (v + w) = u \cdot v + u \cdot w$
- Scalar compatibility: $(c \cdot u) \cdot v = c\,(u \cdot v)$
- Non-negativity: $v \cdot v = \|v\|^2 \geq 0$

These guarantee that the dot product behaves consistently and meshes with the structure of vector spaces.

Why It Matters

The dot product is the first bridge between algebra and geometry. It:

- Defines angles and orthogonality in higher dimensions.
- Powers projections and decompositions, which underlie least squares, regression, and data fitting.
- Appears in physics as energy, power, and work.
- Serves as the kernel of many machine learning methods (e.g., similarity measures in high-dimensional spaces).

Without the dot product, linear algebra would lack a way to connect numbers with geometry and meaning.

Try It Yourself

1. Compute $(2, -1) \cdot (-3, 4)$. Then find the angle between them.
2. Check if (1, 2, 3) and (2, 4, 6) are orthogonal. What does the dot product tell you?
3. Find the projection of (3, 1) onto (1, 2). Draw the original vector, the projection, and the perpendicular component.
4. In physics terms: Suppose a 10 N force is applied at 60° to the direction of motion, and the displacement is 5 m. How much work is done?

These exercises reveal the dual power of the dot product: as a formula to compute and as a geometric tool to interpret.

7. Angles Between Vectors and Cosine

Having defined the dot product, we are now ready to measure angles between vectors. In everyday life, angles tell us how two lines or directions relate-whether they point the same way, are perpendicular, or are opposed. In linear algebra, the dot product and cosine function give us a precise, generalizable way to define angles in any dimension, not just in 2D or 3D. This section explores how we compute, interpret, and apply vector angles.

The Definition of an Angle Between Vectors

For two nonzero vectors u and v, the angle θ between them is defined by:

$$\cos(\theta) = \frac{u \cdot v}{\|u\| \, \|v\|}.$$

This formula comes directly from the geometric definition of the dot product.
Rearranging gives:

$$\theta = \arccos\left(\frac{u \cdot v}{\|u\| \, \|v\|}\right).$$

Key points:

- θ is always between 0° and 180° (or 0 and π radians).
- The denominator normalizes the dot product by dividing by the product of lengths, so the result is dimensionless and always between -1 and 1.
- The cosine value directly encodes alignment: positive, zero, or negative.

Interpretation of Cosine Values

The cosine tells us about the directional relationship:

- $\cos(\theta) = 1 \;\Rightarrow\; \theta = 0^\circ \rightarrow$ vectors point in exactly the same direction.
- $\cos(\theta) = 0 \;\Rightarrow\; \theta = 90^\circ \rightarrow$ vectors are orthogonal (perpendicular).
- $\cos(\theta) = -1 \;\Rightarrow\; \theta = 180^\circ \rightarrow$ vectors point in exactly opposite directions.
- $\cos(\theta) > 0 \rightarrow$ acute angle $\rightarrow$ vectors point more "together" than apart.
- $\cos(\theta) < 0 \rightarrow$ obtuse angle $\rightarrow$ vectors point more "against" each other.

Thus, the cosine compresses geometric alignment into a single number.

Examples

1. $u = (1, 0),\ v = (0, 1)$
   - Dot product: $1 \times 0 + 0 \times 1 = 0$
   - Norms: 1 and 1
   - $\cos(\theta) = 0 \;\Rightarrow\; \theta = 90^\circ$
     The vectors are perpendicular, as expected.
2. $u = (2, 3),\ v = (4, 6)$

- Dot product: $(2 \times 4) + (3 \times 6) = 8 + 18 = 26$
- Norms: $\sqrt{2^2 + 3^2} = \sqrt{13}$, and $\sqrt{4^2 + 6^2} = \sqrt{52} = 2\sqrt{13}$
- $\cos(\theta) = \frac{26}{\sqrt{13} \cdot 2\sqrt{13}} = \frac{26}{26} = 1$
- $\theta = 0^\circ$
  These vectors are multiples, so they align perfectly.

3. $u = (1, 1),\ v = (-1, 1)$
   - Dot product: $(1 \times -1) + (1 \times 1) = -1 + 1 = 0$
   - $\cos(\theta) = 0 \ \Rightarrow \ \theta = 90^\circ$
     The vectors are perpendicular, forming diagonals of a square.

Angles in Higher Dimensions

The beauty of the formula is that it works in any dimension.
Even in $\mathbb{R}^{100}$ or higher, we can define the angle between two vectors using only their dot product and norms.

While we cannot visualize the geometry directly in high dimensions, the cosine formula still captures how aligned two directions are:

$$\cos(\theta) = \frac{u \cdot v}{\|u\| \, \|v\|}.$$

This is critical in machine learning, where data often lives in very high-dimensional spaces.

Cosine Similarity

The cosine of the angle between two vectors is often called cosine similarity. It is widely used in data analysis and machine learning to measure how similar two data vectors are, independent of their magnitude.

- In text mining, documents are turned into word-frequency vectors. Cosine similarity measures how "close in topic" two documents are, regardless of length.
- In recommendation systems, cosine similarity compares user preference vectors to suggest similar users or items.

This demonstrates how a geometric concept extends far beyond pure math.

Orthogonality Revisited

The angle formula reinforces the special role of orthogonality.
If $\cos(\theta) = 0$, then $u \cdot v = 0$.

This means the dot product not only computes length but also serves as a direct test for perpendicularity.
This algebraic shortcut is far easier than manually checking geometric right angles.

Angles and Projections

Angles are closely tied to projections.
The length of the projection of u onto v is $\|u\| \cos(\theta)$.

If the angle is small, the projection is large — most of u lies in the direction of v.
If the angle is close to $90°$, the projection shrinks toward zero.

Thus, the cosine acts as a scaling factor between directions.

Why It Matters

Angles between vectors provide:

- A way to generalize geometry beyond 2D/3D.
- A measure of similarity in high-dimensional data.
- The foundation for orthogonality, projections, and decomposition of spaces.
- A tool for optimization: in gradient descent, for example, the angle between the gradient and step direction determines how effectively we reduce error.

Without the ability to measure angles, we could not connect algebraic manipulations with geometric intuition or practical applications.

Try It Yourself

1. Compute the angle between (2, 1) and (1, –1). Interpret the result.
2. Find two vectors in 3D that form a 60° angle. Verify using the cosine formula.
3. Consider word vectors for "cat" and "dog" in a machine learning model. Why might cosine similarity be a better measure of similarity than Euclidean distance?
4. Challenge: In $\mathbb{R}^3$, find a vector orthogonal to both (1, 2, 3) and (3, 2, 1). What angle does it make with each of them?

By experimenting with these problems, you will see how angles provide the missing link between algebraic formulas and geometric meaning in linear algebra.

8. Projections and Decompositions

In earlier sections, we saw how the dot product measures alignment and how the cosine formula gives us angles between vectors. The next natural step is to use these tools to project one vector onto another. Projection is a way to "shadow" one vector onto the direction of another, splitting vectors into meaningful parts: one along a given direction and one perpendicular to it. This is the essence of decomposition, and it is everywhere in linear algebra, geometry, physics, and data science.

Scalar Projection

The scalar projection of a vector u onto a vector v measures how much of u lies in the direction of v. It is given by:

$$\text{proj}_{\text{scalar}}(u \text{ onto } v) = \frac{u \cdot v}{\|v\|}.$$

- If this value is positive, u has a component pointing in the same direction as v.
- If it is negative, u points partly in the opposite direction.
- If it is zero, u is completely perpendicular to v.

Example:
$u = (3, 4)$, $v = (1, 0)$.
Dot product: $(3 \times 1 + 4 \times 0) = 3$.
$\|v\| = 1$.
So the scalar projection is 3. This tells us u has a "shadow" of length 3 on the x-axis.

Vector Projection

The vector projection gives the actual arrow in the direction of v that corresponds to this scalar amount:

$$\text{proj}_{\text{vector}}(u \text{ onto } v) = \frac{u \cdot v}{\|v\|^2}\, v.$$

This formula normalizes v into a unit vector, then scales it by the scalar projection.
The result is a new vector lying along v, capturing exactly the "parallel" part of u.

Example:
$u = (3, 4)$, $v = (1, 2)$

- Dot product: $3 \times 1 + 4 \times 2 = 3 + 8 = 11$
- Norm squared of v: $(1^2 + 2^2) = 5$
- Coefficient: $11/5 = 2.2$
- Projection vector: $2.2 \cdot (1, 2) = (2.2, 4.4)$

So the part of $(3, 4)$ in the direction of $(1, 2)$ is $(2.2, 4.4)$.

Perpendicular Component

Once we have the projection, we can find the perpendicular component (often called the rejection) simply by subtracting:

$$u_\perp = u - \text{proj}_{\text{vector}}(u \text{ onto } v).$$

This gives the part of u that is entirely orthogonal to v.

Example continued:
$u_\perp = (3, 4) - (2.2, 4.4) = (0.8, -0.4)$

Check:
$(0.8, -0.4) \cdot (1, 2) = 0.8 \times 1 + (-0.4) \times 2 = 0.8 - 0.8 = 0.$
Indeed, orthogonal.

Geometric Picture

Projection is like dropping a perpendicular from one vector onto another. Imagine shining a light perpendicular to v: the shadow of u on the line spanned by v is the projection. This visualization explains why projections split vectors naturally into two pieces:

- Parallel part: Along the line of v.
- Perpendicular part: Orthogonal to v, forming a right angle.

Together, these two parts reconstruct the original vector exactly.

Decomposition of Vectors

Every vector u can be decomposed relative to another vector v into two parts:

$$u = \text{proj}_{\text{vector}}(u \text{ onto } v) + (u - \text{proj}_{\text{vector}}(u \text{ onto } v)).$$

This decomposition is unique and geometrically meaningful.
It generalizes to subspaces: we can project onto entire planes or higher-dimensional spans, splitting a vector into a "within-subspace" part and a "perpendicular-to-subspace" part.

Applications

1. Physics (Work and Forces): Work is the projection of force onto displacement. Only the part of the force in the direction of motion contributes. Example: Pushing on a sled partly sideways wastes effort-the sideways component projects to zero.
2. Geometry and Engineering: Projections are used in CAD (computer-aided design) to flatten 3D objects onto 2D surfaces, like blueprints or shadows.
3. Computer Graphics: Rendering 3D scenes onto a 2D screen is fundamentally a projection process.
4. Data Science: Projecting high-dimensional data onto a lower-dimensional subspace (like the first two principal components in PCA) makes patterns visible while preserving as much information as possible.
5. Signal Processing: Decomposition into projections onto sine and cosine waves forms the basis of Fourier analysis, which powers audio, image, and video compression.

Algebraic Properties

- Projections are linear: proj(u + w) = proj(u) + proj(w).
- The perpendicular part is always orthogonal to the direction of projection.
- The decomposition is unique: no other pair of parallel and perpendicular vectors will reconstruct u.
- The projection operator onto a unit vector $\hat{v}$ satisfies: proj(u) = ($\hat{v}$ $\hat{v}$)u, showing how projection can be expressed in matrix form.

Why It Matters

Projection is not just a geometric trick; it is the core of many advanced topics:

- Least squares regression is finding the projection of a data vector onto the span of predictor vectors.
- Orthogonal decompositions like Gram–Schmidt and QR factorization rely on projections to build orthogonal bases.
- Optimization methods often involve projecting guesses back onto feasible sets.
- Machine learning uses projections constantly to reduce dimensions, compare vectors, and align features.

Without projection, we could not cleanly separate influence along directions or reduce complexity in structured ways.

Try It Yourself

1. Project (2, 3) onto (1, 0). What does the perpendicular component look like?
2. Project (3, 1) onto (2, 2). Verify the perpendicular part is orthogonal.
3. Decompose (5, 5, 0) into parallel and perpendicular parts relative to (1, 0, 0).
4. Challenge: Write the projection matrix for projecting onto (1, 2). Apply it to (3, 4). Does it match the formula?

Through these exercises, you will see that projection is more than an operation-it is a lens through which we decompose, interpret, and simplify vectors and spaces.

9. Cauchy–Schwarz and Triangle Inequalities

Linear algebra is not only about operations with vectors-it also involves understanding the fundamental relationships between them. Two of the most important results in this regard are the Cauchy–Schwarz inequality and the triangle inequality. These are cornerstones of vector spaces because they establish precise boundaries for lengths, angles, and inner products. Without them, the geometry of linear algebra would fall apart.

The Cauchy–Schwarz Inequality

For any two vectors u and v in $\mathbb{R}^n$, the Cauchy–Schwarz inequality states:

$$|u \cdot v| \leq \|u\| \, \|v\|.$$

This means that the absolute value of the dot product of two vectors is always less than or equal to the product of their lengths.

Equality holds if and only if u and v are linearly dependent (i.e., one is a scalar multiple of the other).

Why It Is True

Recall the geometric formula for the dot product:

$$u \cdot v = \|u\| \, \|v\| \cos(\theta).$$

Since $-1 \leq \cos(\theta) \leq 1$, the magnitude of the dot product cannot exceed $\|u\| \, \|v\|$.
This is exactly the inequality.

Example

Let $u = (3, 4)$ and $v = (-4, 3)$.

- Dot product: $(3 \times -4) + (4 \times 3) = -12 + 12 = 0$
- Norms: $\|u\| = 5$, $\|v\| = 5$
- Product of norms: 25
- $|u \cdot v| = 0 \leq 25$, which satisfies the inequality

Equality does not hold since they are not multiples - they are perpendicular.

Intuition

The inequality tells us that two vectors can never "overlap" more strongly than the product of their magnitudes. If they align perfectly, the overlap is maximum (equality). If they're perpendicular, the overlap is zero.

Think of it as: "the shadow of one vector on another can never be longer than the vector itself."

The Triangle Inequality

For any vectors u and v, the triangle inequality states:

$$\|u + v\| \leq \|u\| + \|v\|.$$

This mirrors the geometric fact that in a triangle, any side is at most as long as the sum of the other two sides.

Example

Let $u = (1, 2)$ and $v = (3, 4)$.

- $\|u + v\| = \|(4, 6)\| = \sqrt{16 + 36} = \sqrt{52} \approx 7.21$
- $\|u\| + \|v\| = \sqrt{5} + 5 \approx 2.24 + 5 = 7.24$

Indeed, $7.21 \leq 7.24$, very close in this case.

Equality Case

The triangle inequality becomes equality when the vectors point in exactly the same direction (or are scalar multiples with nonnegative coefficients). For example, (1, 1) and (2, 2) produce equality because adding them gives a vector whose length equals the sum of their lengths.

Extensions

- These inequalities hold in all inner product spaces, not just . This means they apply to functions, sequences, and more abstract mathematical objects.
- In Hilbert spaces (infinite-dimensional generalizations), they remain just as essential.

Why They Matter

1. They guarantee that the dot product and norm are well-behaved and geometrically meaningful.
2. They ensure that the norm satisfies the requirements of a distance measure: nonnegativity, symmetry, and triangle inequality.
3. They underpin the validity of projections, orthogonality, and least squares methods.
4. They are essential in proving convergence of algorithms, error bounds, and stability in numerical linear algebra.

Without these inequalities, we could not trust that the geometry of vector spaces behaves consistently.

Try It Yourself

1. Verify Cauchy–Schwarz for (2, –1, 3) and (–1, 4, 0). Compute both sides.
2. Try the triangle inequality for (–3, 4) and (5, –12). Does equality hold?
3. Find two vectors where Cauchy–Schwarz is an equality. Explain why.
4. Challenge: Prove the triangle inequality in $\mathbb{R}^2$ using only the Pythagorean theorem and algebra, without relying on dot products.

Working through these problems will show you why these inequalities are not abstract curiosities but the structural glue of linear algebra's geometry.

10. Orthonormal sets in $\mathbb{R}^2$ and $\mathbb{R}^3$

Up to now, we've discussed vectors, their lengths, angles, and how to project one onto another. A natural culmination of these ideas is the concept of orthonormal sets. These are collections of vectors that are not only orthogonal (mutually perpendicular) but also normalized (each of length 1). Orthonormal sets form the cleanest, most efficient coordinate systems in linear algebra. They are the mathematical equivalent of having rulers at right angles, perfectly calibrated to unit length.

Orthogonal and Normalized

Let's break the term "orthonormal" into two parts:

- Orthogonal: Two vectors u and v are orthogonal if $u \cdot v = 0$.
  In $\mathbb{R}^2$, this means the vectors meet at a right angle.
  In $\mathbb{R}^3$, it means they form perpendicular directions.
- Normalized: A vector v is normalized if its length is 1, i.e., $\|v\| = 1$.
  Such vectors are called unit vectors.

When we combine both conditions, we get orthonormal vectors: vectors that are both perpendicular to each other and have unit length.

Orthonormal Sets in $\mathbb{R}^2$

In two dimensions, an orthonormal set typically consists of two vectors.
A classic example is:

$e_1 = (1, 0), \quad e_2 = (0, 1)$

- Dot product: $e_1 \cdot e_2 = (1 \times 0 + 0 \times 1) = 0 \quad \Rightarrow \quad$ orthogonal
- Lengths: $\|e_1\| = 1$, $\|e_2\| = 1 \quad \Rightarrow \quad$ normalized

Thus, $\{e_1, e_2\}$ is an orthonormal set.
In fact, this is the standard basis for $\mathbb{R}^2$.
Any vector (x, y) can be written as $xe_1 + ye_2$.
This is the simplest coordinate system.

Orthonormal Sets in $\mathbb{R}^3$

In three dimensions, an orthonormal set usually has three vectors.
The standard basis is:

$e_1 = (1, 0, 0), \quad e_2 = (0, 1, 0), \quad e_3 = (0, 0, 1)$

- Each pair has dot product zero, so they are orthogonal
- Each has length 1, so they are normalized
- Together, they span all of $\mathbb{R}^3$

Geometrically, they correspond to the x-, y-, and z-axes in 3D space.
Any vector (x, y, z) can be written as a linear combination $xe_1 + ye_2 + ze_3$.

Beyond the Standard Basis

The standard basis is not the only orthonormal set. For example:

$u = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right), \quad v = \left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$

- Dot product: $(\frac{1}{\sqrt{2}})(-\frac{1}{\sqrt{2}}) + (\frac{1}{\sqrt{2}})(\frac{1}{\sqrt{2}}) = -\frac{1}{2} + \frac{1}{2} = 0$
- Lengths: $\sqrt{(\frac{1}{\sqrt{2}})^2 + (\frac{1}{\sqrt{2}})^2} = \sqrt{\frac{1}{2} + \frac{1}{2}} = 1$

So $\{u, v\}$ is also orthonormal in $\mathbb{R}^2$.
These vectors are rotated 45° relative to the standard axes.

Similarly, in $\mathbb{R}^3$, you can construct rotated orthonormal sets (such as unit vectors along diagonals), as long as the conditions of perpendicularity and unit length hold.

Properties of Orthonormal Sets

1. Simplified coordinates: If $\{v_1, \ldots, v_k\}$ is an orthonormal set, then for any vector u in their span, the coefficients are easy to compute:

$$c_i = u \cdot v_i$$

   This is much simpler than solving systems of equations.

2. Pythagorean theorem generalized: If vectors are orthonormal, the squared length of their sum is the sum of the squares of their coefficients.
   For example, if $u = av_1 + bv_2$, then

$$\|u\|^2 = a^2 + b^2$$

3. Projection is easy: Projecting onto an orthonormal set is straightforward — just take dot products.

4. Matrices become nice: When vectors form the columns of a matrix, orthonormality makes that matrix an orthogonal matrix, which has special properties: its transpose equals its inverse, and it preserves lengths and angles.

Importance in $\mathbb{R}^2$ and $\mathbb{R}^3$

- In geometry, orthonormal bases correspond to coordinate axes.
- In physics, they represent independent directions of motion or force.
- In computer graphics, orthonormal sets define camera axes and object rotations.
- In engineering, they simplify stress, strain, and rotation analysis.

Even though $\mathbb{R}^2$ and $\mathbb{R}^3$ are relatively simple, the same ideas extend naturally to higher dimensions, where visualization is impossible but the algebra is identical.

Why Orthonormal Sets Matter

Orthonormality is the gold standard for building bases in linear algebra:

- It makes calculations fast and simple.
- It ensures numerical stability in computations (important in algorithms and simulations).
- It underpins key decompositions like QR factorization, singular value decomposition (SVD), and spectral theorems.
- It provides the cleanest way to think about space: orthogonal, independent directions scaled to unit length.

Whenever possible, mathematicians and engineers prefer orthonormal bases over arbitrary ones.

Try It Yourself

1. Verify that (3/5, 4/5) and (–4/5, 3/5) form an orthonormal set in $\mathbb{R}^2$.
2. Construct three orthonormal vectors in $\mathbb{R}^3$ that are not the standard basis. Hint: start with $(1/\sqrt{2}, 1/\sqrt{2}, 0)$ and build perpendiculars.
3. For u = (2, 1), compute its coordinates relative to the orthonormal set $\{(1/\sqrt{2}, 1/\sqrt{2}), (-1/\sqrt{2}, 1/\sqrt{2})\}$.
4. Challenge: Prove that if $\{v_1, \ldots, v_n\}$ is orthonormal, then the matrix with these as columns is orthogonal, i.e., $Q^T Q = I$.

Through these exercises, you will see how orthonormal sets make every aspect of linear algebra-from projections to decompositions-simpler, cleaner, and more powerful.

Closing

```
Lengths, angles revealed,
projections trace hidden lines,
clarity takes shape.
```

Chapter 2. Matrices and basic operations

Opening

```
Rows and columns meet,
woven grids of silent rules,
machines of order.
```

11. Matrices as Tables and as Machines

The next stage in our journey is to move from vectors to matrices. A matrix may look like just a rectangular array of numbers, but in linear algebra it plays two distinct and equally important roles:

1. As a table of numbers, storing data, coefficients, or geometric patterns in a compact form.
2. As a machine that transforms vectors into other vectors, capturing the essence of linear transformations.

Both views are valid, and learning to switch between them is crucial to building intuition.

Matrices as Tables

At the most basic level, a matrix is a grid of numbers arranged into rows and columns.

- A 2×2 matrix has 2 rows and 2 columns:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

- A 3×2 matrix has 3 rows and 2 columns:

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Each entry a_{ij} or b_{ij} tells us the number in the i-th row and j-th column. The rows of a matrix can represent constraints, equations, or observations; the columns can represent features, variables, or directions.

In this sense, matrices are data containers, organizing information efficiently. That's why matrices show up in spreadsheets, statistics, computer graphics, and scientific computing.

Matrices as Machines

The deeper view of a matrix is as a function from vectors to vectors. If x is a column vector, then multiplying A · x produces a new vector.

For example:

$$A = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}.$$

Multiplying:

$$A\mathbf{x} = \begin{bmatrix} 2 \times 4 + 0 \times 5 \\ 1 \times 4 + 3 \times 5 \end{bmatrix} = \begin{bmatrix} 8 \\ 19 \end{bmatrix}.$$

Here, the matrix is acting as a machine that takes input (4, 5) and outputs (8, 19). The "machine rules" are encoded in the rows of A.

Column View of Matrix Multiplication

Another way to see it: multiplying A · x is the same as taking a linear combination of A's columns.

If

$$A = \begin{bmatrix} a_1 & a_2 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

then:

$$A\mathbf{x} = x_1 a_1 + x_2 a_2.$$

So the vector x tells the machine "how much" of each column to mix together. This column view is critical-it connects matrices to span, dimension, and basis ideas we saw earlier.

The Duality of Tables and Machines

- As a table, a matrix is a static object: numbers written in rows and columns.
- As a machine, the same numbers become instructions for transforming vectors.

This duality is not just conceptual-it's the key to understanding why linear algebra is so powerful. A dataset, once stored as a table, can be interpreted as a transformation. Likewise, a transformation, once understood, can be encoded as a table.

Examples in Practice

1. Physics: A stress–strain matrix is a table of coefficients. But it also acts as a machine that transforms applied forces into deformations.
2. Computer Graphics: A 2D rotation matrix is a machine that spins vectors, but it can be stored in a simple 2×2 table.
3. Economics: Input–output models use matrices as tables of production coefficients. Applying them to demand vectors transforms them into resource requirements.

Geometric Intuition

Every 2×2 or 3×3 matrix corresponds to some linear transformation in the plane or space. Examples:

- Scaling: $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ doubles lengths.
- Reflection: $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ flips across the x-axis.
- Rotation: $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ rotates vectors by .

These are not just tables of numbers-they are precise, reusable machines.

Why This Matters

This section sets the stage for all matrix theory:

- Thinking of matrices as tables helps in data interpretation and organization.
- Thinking of matrices as machines helps in understanding linear transformations, eigenvalues, and decompositions.
- Most importantly, learning to switch between the two perspectives makes linear algebra both concrete and abstract-bridging computation with geometry.

Try It Yourself

1. Write a 2×3 matrix and identify its rows and columns. What might they represent in a real-world dataset?
2. Multiply $\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$ by $\begin{bmatrix}2\\-1\end{bmatrix}$. Interpret the result using both the row and column views.
3. Construct a matrix that scales vectors by 2 along the x-axis and reflects them across the y-axis. Test it on (1, 1).
4. Challenge: Show how the same 3×3 rotation matrix can be viewed as a data table of cosines/sines and as a machine that turns input vectors.

By mastering both perspectives, you'll see matrices not just as numbers but as dynamic objects that encode and execute transformations.

12. Matrix Shapes, Indexing, and Block Views

Matrices come in many shapes and sizes, and the way we label their entries matters. This section is about learning how to read and write matrices carefully, how to work with rows and columns, and how to use block structure to simplify problems. These seemingly simple ideas are what allow us to manipulate large systems with precision and efficiency.

Shapes of Matrices

The shape of a matrix is given by its number of rows and columns:

- A m×n matrix has m rows and n columns.
- Rows run horizontally, columns run vertically.
- Square matrices have m = n; rectangular matrices have m n.

Examples:

- A 2×3 matrix:

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

- A 3×2 matrix:

$$\begin{bmatrix}7 & 8\\9 & 10\\11 & 12\end{bmatrix}$$

Shape matters because it determines whether certain operations (like multiplication) are possible.

Indexing: The Language of Entries

Each entry in a matrix has two indices: one for its row, one for its column.

- a_{ij} = entry in row i, column j.
- The first index always refers to the row, the second to the column.

For example, in

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix},$$

we have:

- $a_{11} = 1$, $a_{23} = 8$, $a_{32} = 6$.

Indexing is the grammar of matrix language. Without it, we can't specify positions or write formulas clearly.

Rows and Columns as Vectors

Every row and every column of a matrix is itself a vector.

- The i-th row is written as $A_{i,*}$.
- The j-th column is written as $A_{*,j}$.

Example: From the matrix above,

- First row: (1, 4, 7).
- Second column: (4, 5, 6).

This duality is powerful: rows often represent constraints or equations, while columns represent directions or features. Later, when we interpret matrix–vector products, we'll see that multiplying A · x means combining columns, while multiplying y · A means combining rows.

Submatrices

Sometimes we want just part of a matrix. A submatrix is formed by selecting certain rows and columns.

Example: From

$$B = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \\ 7 & 8 & 9 \end{bmatrix},$$

the submatrix of the first two rows and last two columns is:

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

Submatrices allow us to zoom in and isolate parts of a problem.

Block Matrices: Dividing to Conquer

Large matrices can often be broken into blocks, which are smaller submatrices arranged inside. This is like dividing a spreadsheet into quadrants.

For example:

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where each A_{ij} is itself a smaller matrix.

This structure is useful in:

- Computation: Algorithms often process blocks instead of individual entries.
- Theory: Many proofs and factorizations rely on viewing a matrix in blocks (e.g., LU, QR, Schur decomposition).
- Applications: Partitioning data tables into logical sections.

Example: Splitting a 4×4 matrix into four 2×2 blocks helps us treat it as a "matrix of matrices."

Special Shapes

Some shapes of matrices are so common they deserve names:

- Row vector: 1×n matrix.
- Column vector: n×1 matrix.
- Diagonal matrix: Nonzero entries only on the diagonal.
- Identity matrix: Square diagonal matrix with 1's on the diagonal.
- Zero matrix: All entries are 0.

Recognizing these shapes saves time and clarifies reasoning.

Why It Matters

Careful attention to matrix shapes, indexing, and block views ensures:

1. Precision: We can describe positions unambiguously.
2. Structure awareness: Recognizing patterns (diagonal, triangular, block) leads to more efficient computations.
3. Scalability: Block partitioning is the foundation of modern numerical linear algebra libraries, where matrices are too large to handle entry by entry.
4. Geometry: Rows and columns as vectors connect matrix structure to span, basis, and dimension.

These basic tools prepare us for multiplication, transformations, and factorization.

Try It Yourself

1. Write a 3×4 matrix and label the entry in row 2, column 3.
2. Extract a 2×2 submatrix from the corners of a 4×4 matrix of your choice.
3. Break a 6×6 matrix into four 3×3 blocks. How would you represent it compactly?
4. Challenge: Given

$$D = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix},$$

   write it as a block matrix with a 2×2 block in the top-left, a 2×2 block in the top-right, and a 1×4 block in the bottom row.

By practicing with shapes, indexing, and blocks, you'll develop the ability to navigate matrices not just as raw grids of numbers but as structured objects ready for deeper algebraic and geometric insights.

13. Matrix Addition and Scalar Multiplication

Before exploring matrix–vector and matrix–matrix multiplication, it is essential to understand the simplest operations we can perform with matrices: addition and scalar multiplication. These operations extend the rules we learned for vectors, but now applied to entire grids of numbers. Although straightforward, they are the foundation for more complex algebraic manipulations and help establish the idea of matrices as elements of a vector space.

Matrix Addition: Entry by Entry

If two matrices A and B have the same shape (same number of rows and columns), we can add them by adding corresponding entries.

Formally: If

$$A = [a_{ij}], \quad B = [b_{ij}],$$

then

$$A + B = [a_{ij} + b_{ij}].$$

Example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}.$$

Key point: Addition is only defined if the matrices are the same shape. A 2×3 matrix cannot be added to a 3×2 matrix.

Scalar Multiplication: Scaling Every Entry

A scalar multiplies every entry of a matrix.

Formally: For scalar c and matrix $A = [a_{ij}]$,

$$cA = [c \cdot a_{ij}].$$

Example:

$$3 \cdot \begin{bmatrix} 2 & -1 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 0 & 12 \end{bmatrix}.$$

This mirrors vector scaling: stretching or shrinking the whole matrix by a constant factor.

Properties of Addition and Scalar Multiplication

These two operations satisfy familiar algebraic properties that make the set of all m×n matrices into a vector space:

1. Commutativity: $A + B = B + A$.
2. Associativity: $(A + B) + C = A + (B + C)$.
3. Additive identity: $A + 0 = A$, where 0 is the zero matrix.
4. Additive inverse: For every A, there exists $-A$ such that $A + (-A) = 0$.
5. Distributivity: $c(A + B) = cA + cB$.
6. Compatibility: $(c + d)A = cA + dA$.
7. Scalar associativity: $(cd)A = c(dA)$.
8. Unit scalar: $1A = A$.

These guarantee that working with matrices feels like working with numbers and vectors, only in a higher-level setting.

Matrix Arithmetic as Table Operations

From the table view, addition and scalar multiplication are just simple bookkeeping: line up two tables of the same shape and add entry by entry; multiply the whole table by a constant.

Example: Imagine two spreadsheets of monthly expenses. Adding them gives combined totals. Multiplying by 12 converts a monthly table into a yearly estimate.

Matrix Arithmetic as Machine Operations

From the machine view, these operations adjust the behavior of linear transformations:

- Adding matrices corresponds to adding their effects when applied to vectors.
- Scaling a matrix scales the effect of the transformation.

Example: Let A rotate vectors slightly, and B stretch vectors. The matrix $A + B$ represents a transformation that applies both influences together. Scaling by 2 doubles the effect of the transformation.

Special Case: Zero and Identity

- Zero matrix: All entries are 0. Adding it to any matrix changes nothing.
- Scalar multiples of the identity: cI scales every vector by c when applied. For example, $2I$ doubles every vector's length.

These act as neutral or scaling elements in matrix arithmetic.

Geometric Intuition

1. In $\mathbb{R}^2$ or $\mathbb{R}^3$, adding transformation matrices is like superimposing geometric effects: e.g., one matrix shears, another rotates, their sum mixes both.
2. Scaling a transformation makes its action stronger or weaker. Doubling a shear makes it twice as pronounced.

This shows that even before multiplication, addition and scaling already have geometric meaning.

Why It Matters

Though simple, these operations:

- Define matrices as elements of vector spaces.
- Lay the groundwork for linear combinations of matrices, critical in eigenvalue problems, optimization, and control theory.
- Enable modular problem-solving: break big transformations into smaller ones and recombine them.
- Appear everywhere in practice, from combining datasets to scaling transformations.

Without addition and scalar multiplication, we could not treat matrices systematically as algebraic objects.

Try It Yourself

1. Add

$$\begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -2 & 5 \\ 4 & -3 \end{bmatrix}.$$

2. Multiply

$$\begin{bmatrix} 1 & -1 & 2 \\ 0 & 3 & 4 \end{bmatrix}$$

by –2.

3. Show that (A + B) + C = A + (B + C) with explicit 2×2 matrices.
4. Challenge: Construct two 3×3 matrices A and B such that A + B = 0. What does that tell you about B?

By practicing these fundamentals, you will see that even the most basic operations on matrices already build the algebraic backbone for deeper results like matrix multiplication, transformations, and factorization.

14. Matrix–Vector Product (Linear Combinations of Columns)

We now arrive at one of the most important operations in all of linear algebra: the matrix–vector product. This operation takes a matrix A and a vector x, and produces a new vector. While the computation is straightforward, its interpretations are deep: it can be seen as combining rows, as combining columns, or as applying a linear transformation. This is the operation that connects matrices to the geometry of vector spaces.

The Algebraic Rule

Suppose A is an $m \times n$ matrix, and x is a vector in $\mathbb{R}^n$. The product $A\mathbf{x}$ is a vector in $\mathbb{R}^m$, defined as:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Then:

$$A\mathbf{x} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Each entry of the output is a dot product between one row of A and the vector x.

Row View: Dot Products

From the row perspective, $A\mathbf{x}$ is computed row by row:

- Take each row of A.
- Dot it with x.
- That result becomes one entry of the output.

Example:

$$A = \begin{bmatrix} 2 & 1 \\ 3 & 4 \\ -1 & 2 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 5 \\ -1 \end{bmatrix}.$$

- First row dot x: $2(5) + 1(-1) = 9$.
- Second row dot x: $3(5) + 4(-1) = 11$.
- Third row dot x: $(-1)(5) + 2(-1) = -7$.

So:

$$A\mathbf{x} = \begin{bmatrix} 9 \\ 11 \\ -7 \end{bmatrix}.$$

Column View: Linear Combinations

From the column perspective, $A\mathbf{x}$ is a linear combination of the columns of A.

If

$$A = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

then:

$$A\mathbf{x} = x_1 a_1 + x_2 a_2 + \cdots + x_n a_n.$$

That is: multiply each column of A by the corresponding entry in x, then add them up.

This interpretation connects directly to the idea of span: the set of all vectors $A\mathbf{x}$ as x varies is exactly the span of the columns of A.

The Machine View: Linear Transformations

The machine view ties everything together: multiplying a vector by a matrix means applying the linear transformation represented by the matrix.

- If A is a 2×2 rotation matrix, then $A\mathbf{x}$ rotates the vector x.
- If A is a scaling matrix, then $A\mathbf{x}$ stretches or shrinks x.
- If A is a projection matrix, then $A\mathbf{x}$ projects x onto a line or plane.

Thus, the algebraic definition encodes geometric and functional meaning.

Examples of Geometric Action

1. Scaling:

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}.$$

Then $A\mathbf{x}$ doubles the length of any vector x.

2. Reflection:

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

This flips vectors across the x-axis.

3. Rotation by :

$$A = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

This rotates vectors counterclockwise by in the plane.

Why It Matters

The matrix–vector product is the building block of everything in linear algebra:

1. It defines the action of a matrix as a linear map.
2. It connects directly to span and dimension (columns generate all possible outputs).
3. It underpins solving linear systems, eigenvalue problems, and decompositions.
4. It is the engine of computation in applied mathematics, from computer graphics to machine learning (e.g., neural networks compute billions of matrix–vector products).

Try It Yourself

1. Compute

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}.$$

2. Express the result of the above product as a linear combination of the columns of the matrix.
3. Construct a 2×2 matrix that reflects vectors across the line $y = x$. Test it on (1, 0) and (0, 1).
4. Challenge: For a 3×3 matrix, show that the set of all possible $A\mathbf{x}$ (as x varies) is exactly the column space of A.

By mastering both the computational rules and the interpretations of the matrix–vector product, you will gain the most important insight in linear algebra: matrices are not just tables-they are engines that transform space.

15. Matrix–Matrix Product (Composition of Linear Steps)

Having understood how a matrix acts on a vector, the next natural step is to understand how one matrix can act on another. This leads us to the matrix–matrix product, a rule for combining two matrices into a single new matrix. Though the arithmetic looks complicated at first, the underlying idea is elegant: multiplying two matrices represents composing two linear transformations.

The Algebraic Rule

Suppose A is an $m \times n$ matrix and B is an $n \times p$ matrix. Their product $C = AB$ is an $m \times p$ matrix defined by:

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

That is: each entry of C is the dot product of the i-th row of A with the j-th column of B.

Example: A 2×3 times a 3×2

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}.$$

Product: $C = AB$ will be 2×2.

- $c_{11} = 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 = 58.$
- $c_{12} = 1 \cdot 8 + 2 \cdot 10 + 3 \cdot 12 = 64.$
- $c_{21} = 4 \cdot 7 + 5 \cdot 9 + 6 \cdot 11 = 139.$
- $c_{22} = 4 \cdot 8 + 5 \cdot 10 + 6 \cdot 12 = 154.$

So:

$$C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}.$$

Column View: Linear Combinations of Columns

From the column perspective, AB is computed by applying A to each column of B.

If $B = [b_1 \; b_2 \; \cdots \; b_p]$, then:

$$AB = [Ab_1 \; Ab_2 \; \cdots \; Ab_p].$$

That is: multiply A by each column of B. This is often the simplest way to think of the product.

Row View: Linear Combinations of Rows

From the row perspective, each row of AB is formed by combining rows of B using coefficients from a row of A. This dual view is less common but equally useful, especially in proofs and algorithms.

The Machine View: Composition of Transformations

The most important interpretation is the machine view: multiplying matrices corresponds to composing transformations.

- If A maps $\mathbb{R}^n \to \mathbb{R}^m$ and B maps $\mathbb{R}^p \to \mathbb{R}^n$, then AB maps $\mathbb{R}^p \to \mathbb{R}^m$.
- In words: do B first, then A.

Example:

- Let B rotate vectors by 90°.
- Let A scale vectors by 2.
- Then AB rotates and then scales-both steps combined into a single transformation.

Geometric Examples

1. Scaling then rotation:

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

Then AB scales vectors by 2 after rotating them 90°.

2. Projection then reflection: If B projects onto the x-axis and A reflects across the y-axis, then AB represents "project then reflect."

Properties of Matrix Multiplication

1. Associative: $(AB)C = A(BC)$.
2. Distributive: $A(B + C) = AB + AC$.
3. Not commutative: In general, $AB \neq BA$. Order matters!
4. Identity: $AI = IA = A$.

These properties highlight that while multiplication is structured, it is not symmetric. The order encodes the order of operations in transformations.

Why It Matters

Matrix multiplication is the core of linear algebra because:

1. It encodes function composition in algebraic form.
2. It provides a way to capture multiple transformations in a single matrix.
3. It underpins algorithms in computer graphics, robotics, statistics, and machine learning.
4. It reveals deeper structure, like commutativity failing, which reflects real-world order of operations.

Almost every application of linear algebra-solving equations, computing eigenvalues, training neural networks-relies on efficient matrix multiplication.

Try It Yourself

1. Compute

$$\begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix}.$$

2. Show that $AB \neq BA$ for the matrices

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

3. Construct two 2×2 matrices where $AB = BA$. Why does commutativity happen here?
4. Challenge: If A is a projection and B is a rotation, compute AB and BA. Do they represent the same geometric operation?

Through these perspectives, the matrix–matrix product shifts from being a mechanical formula to being a language for combining linear steps-each product telling the story of "do this, then that."

16. Identity, Inverse, and Transpose

With addition, scalar multiplication, and matrix multiplication in place, we now introduce three special operations and objects that form the backbone of matrix algebra: the identity matrix, the inverse of a matrix, and the transpose of a matrix. Each captures a fundamental principle-neutrality, reversibility, and symmetry-and together they provide the algebraic structure that makes linear algebra so powerful.

The Identity Matrix

The identity matrix is the matrix equivalent of the number 1 in multiplication.

- Definition: The identity matrix I_n is the $n \times n$ matrix with 1's on the diagonal and 0's everywhere else.

Example (3×3):

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- Property: For any $n \times n$ matrix A,

$$AI_n = I_n A = A.$$

- Machine view: I does nothing-it maps every vector to itself.

The Inverse of a Matrix

The inverse is the matrix equivalent of the reciprocal of a number.

- Definition: For a square matrix A, its inverse A^{-1} is the matrix such that

$$AA^{-1} = A^{-1}A = I.$$

- Not all matrices have inverses. A matrix is invertible if and only if it is square and its determinant is nonzero.

Example:

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}, \quad A^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}.$$

Check:

$$AA^{-1} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I.$$

- Machine view: Applying A transforms a vector. Applying A^{-1} undoes that transformation, restoring the original input.

Non-Invertible Matrices

Some matrices cannot be inverted. These are called singular.

- Example:

$$B = \begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix}.$$

  Here, the second column is a multiple of the first. The transformation squashes vectors into a line, losing information-so it cannot be reversed.

This ties invertibility to geometry: a transformation that collapses dimensions cannot be undone.

The Transpose of a Matrix

The transpose reflects a matrix across its diagonal.

- Definition: For $A = [a_{ij}]$,

$$A^T = [a_{ji}].$$

- In words: rows become columns, columns become rows.

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

- Properties:
  - $(A^T)^T = A$.
  - $(A + B)^T = A^T + B^T$.
  - $(cA)^T = cA^T$.
  - $(AB)^T = B^T A^T$ (note the reversed order!).

Symmetric and Orthogonal Matrices

Two important classes emerge from the transpose:

- Symmetric matrices: $A = A^T$. Example:

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

  These have beautiful properties: real eigenvalues and orthogonal eigenvectors.

- Orthogonal matrices: $Q^TQ = I$. Their columns form an orthonormal set, and they represent pure rotations/reflections.

Why It Matters

1. The identity guarantees a neutral element for multiplication.
2. The inverse provides a way to solve equations $A\mathbf{x} = \mathbf{b}$ via $\mathbf{x} = A^{-1}\mathbf{b}$.
3. The transpose ties matrices to geometry, inner products, and symmetry.
4. Together, they form the algebraic foundation for deeper topics: determinants, eigenvalues, factorizations, and numerical methods.

Without these tools, matrix algebra would lack structure and reversibility.

Try It Yourself

1. Compute the transpose of

$$\begin{bmatrix} 1 & 0 & 2 \\ -3 & 4 & 5 \end{bmatrix}.$$

2. Verify that $(AB)^T = B^TA^T$ for

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 0 \\ 5 & 6 \end{bmatrix}.$$

3. Find the inverse of

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

4. Challenge: Show that if Q is orthogonal, then $Q^{-1} = Q^T$. Interpret this geometrically as saying "rotations can be undone by transposing."

Through these exercises, you'll see how identity, inverse, and transpose anchor the structure of linear algebra, providing neutrality, reversibility, and symmetry in every calculation.

17. Symmetric, Diagonal, Triangular, and Permutation Matrices

Not all matrices are created equal-some have special shapes or patterns that give them unique properties. These structured matrices are the workhorses of linear algebra: they simplify computation, reveal geometry, and form the building blocks for algorithms. In this section, we study four especially important classes: symmetric, diagonal, triangular, and permutation matrices.

Symmetric Matrices

A matrix is symmetric if it equals its transpose:

$$A = A^T.$$

Example:

$$\begin{bmatrix} 2 & 3 & 4 \\ 3 & 5 & 6 \\ 4 & 6 & 9 \end{bmatrix}.$$

- Geometric meaning: Symmetric matrices represent linear transformations that have no "handedness." They often arise in physics (energy, covariance, stiffness).
- Algebraic fact: Symmetric matrices have real eigenvalues and an orthonormal basis of eigenvectors. This property underpins the spectral theorem, one of the pillars of linear algebra.

Diagonal Matrices

A matrix is diagonal if all non-diagonal entries are zero.

$$D = \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix}.$$

- Multiplying by D scales each coordinate separately.

- Computations with diagonals are lightning fast:
  - Adding: add diagonal entries.
  - Multiplying: multiply diagonal entries.
  - Inverting: invert each diagonal entry (if nonzero).

Example:

$$\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2x \\ 3y \end{bmatrix}.$$

This is why diagonalization is so valuable: turning a general matrix into a diagonal one simplifies everything.

Triangular Matrices

A matrix is upper triangular if all entries below the main diagonal are zero, and lower triangular if all entries above the diagonal are zero.

- Upper triangular example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}.$$

- Lower triangular example:

$$\begin{bmatrix} 7 & 0 & 0 \\ 8 & 9 & 0 \\ 10 & 11 & 12 \end{bmatrix}.$$

Why they matter:

- Determinant = product of diagonal entries.
- Easy to solve systems by substitution (forward or backward).
- Every square matrix can be factored into triangular matrices (LU decomposition).

Permutation Matrices

A permutation matrix is obtained by permuting the rows (or columns) of an identity matrix.

Example:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Multiplying by P:

- On the left, permutes the rows of a matrix.
- On the right, permutes the columns of a matrix.

Permutation matrices are used in pivoting strategies in elimination, ensuring numerical stability in solving systems. They are also orthogonal: $P^{-1} = P^T$.

Connections Between Them

- A diagonal matrix is a special case of triangular (both upper and lower).
- Symmetric matrices often become diagonal under orthogonal transformations.
- Permutation matrices help reorder triangular or diagonal matrices without breaking their structure.

Together, these classes show that structure leads to simplicity-many computational algorithms exploit these patterns for speed and stability.

Why It Matters

1. Symmetric matrices guarantee stable and interpretable eigen-decompositions.
2. Diagonal matrices make computation effortless.
3. Triangular matrices are the backbone of elimination and factorization methods.
4. Permutation matrices preserve structure while reordering, critical for algorithms.

Almost every advanced method in numerical linear algebra relies on reducing general matrices into one of these structured forms.

Try It Yourself

1. Verify that

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

is symmetric. Find its transpose.

2. Compute the determinant of

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 5 \end{bmatrix}.$$

3. Solve

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 5 & 2 \\ 0 & 0 & 4 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

using back substitution.

4. Construct a 4×4 permutation matrix that swaps the first and last rows. Apply it to a 4×1 vector of your choice.

By exploring these four structured families, you'll start to see that not all matrices are messy-many have order hidden in their arrangement, and exploiting that order is the key to both theoretical understanding and efficient computation.

18. Trace and Basic Matrix Properties

So far we have studied shapes, multiplication rules, and special classes of matrices. In this section we introduce a simple but surprisingly powerful quantity: the trace of a matrix. Along with it, we review a set of basic matrix properties that provide shortcuts, invariants, and insights into how matrices behave.

Definition of the Trace

For a square matrix $A = [a_{ij}]$ of size $n \times n$, the trace is the sum of the diagonal entries:

$$\operatorname{tr}(A) = a_{11} + a_{22} + \cdots + a_{nn}.$$

Example:

$$A = \begin{bmatrix} 2 & 5 & 7 \\ 0 & 3 & 1 \\ 4 & 6 & 8 \end{bmatrix}, \quad \operatorname{tr}(A) = 2 + 3 + 8 = 13.$$

The trace extracts a single number summarizing the "diagonal content" of a matrix.

Properties of the Trace

The trace is linear and interacts nicely with multiplication and transposition:

1. Linearity:
   - $\operatorname{tr}(A + B) = \operatorname{tr}(A) + \operatorname{tr}(B)$.
   - $\operatorname{tr}(cA) = c \cdot \operatorname{tr}(A)$.
2. Cyclic Property:
   - $\operatorname{tr}(AB) = \operatorname{tr}(BA)$, as long as the products are defined.
   - More generally, $\operatorname{tr}(ABC) = \operatorname{tr}(BCA) = \operatorname{tr}(CAB)$.
   - But in general, $\operatorname{tr}(AB) \neq \operatorname{tr}(A)\operatorname{tr}(B)$.
3. Transpose Invariance:
   - $\operatorname{tr}(A^T) = \operatorname{tr}(A)$.
4. Similarity Invariance:
   - If $B = P^{-1}AP$, then $\operatorname{tr}(B) = \operatorname{tr}(A)$.
   - This means the trace is a similarity invariant, depending only on the linear transformation, not the basis.

Trace and Eigenvalues

One of the most important connections is between the trace and eigenvalues:

$$\operatorname{tr}(A) = \lambda_1 + \lambda_2 + \cdots + \lambda_n,$$

where λ_i are the eigenvalues of A (counting multiplicity).

This links the simple diagonal sum to the deep spectral properties of the matrix.

Example:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}, \quad \operatorname{tr}(A) = 4, \quad \lambda_1 = 1,\ \lambda_2 = 3, \quad \lambda_1 + \lambda_2 = 4.$$

Other Basic Matrix Properties

Alongside the trace, here are some important algebraic facts that every student of linear algebra must know:

1. Determinant vs. Trace:
   - For 2×2 matrices, $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, $\operatorname{tr}(A) = a + d$, $\det(A) = ad - bc$.
   - Together, trace and determinant encode the eigenvalues: roots of $x^2 - \operatorname{tr}(A)x + \det(A) = 0$.
2. Norms and Inner Products:
   - The Frobenius norm is defined using the trace: $\|A\|_F = \sqrt{\operatorname{tr}(A^T A)}$.
3. Orthogonal Invariance:
   - For any orthogonal matrix Q, $\operatorname{tr}(Q^T A Q) = \operatorname{tr}(A)$.

Geometric and Practical Meaning

- The trace of a transformation can be seen as the sum of its action along the coordinate axes.
- In physics, the trace of the stress tensor measures pressure.
- In probability, the trace of a covariance matrix is the total variance of a system.
- In statistics and machine learning, the trace is often used as a measure of overall "size" or complexity of a model.

Why It Matters

The trace is deceptively simple but incredibly powerful:

1. It connects directly to eigenvalues, forming a bridge between raw matrix entries and spectral theory.
2. It is invariant under similarity, making it a reliable measure of a transformation independent of basis.
3. It shows up in optimization, physics, statistics, and quantum mechanics.
4. It simplifies computations: many proofs in linear algebra reduce to trace properties.

Try It Yourself

1. Compute the trace of

$$\begin{bmatrix} 4 & 2 & 0 \\ -1 & 3 & 5 \\ 7 & 6 & 1 \end{bmatrix}.$$

2. Verify that $\operatorname{tr}(AB) = \operatorname{tr}(BA)$ for

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 0 \\ 5 & 6 \end{bmatrix}.$$

3. For the 2×2 matrix

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

compute its eigenvalues and check that their sum equals the trace.

4. Challenge: Show that the total variance of a dataset with covariance matrix Σ is equal to $\operatorname{tr}(\Sigma)$.

Mastering the trace and its properties will prepare you for the next leap: understanding how matrices interact with volume, orientation, and determinants.

19. Affine Transforms and Homogeneous Coordinates

Up to now, matrices have been used to describe linear transformations: scaling, rotating, reflecting, projecting. But real-world geometry often involves more than just linear effects-it includes translations (shifts) as well. A pure linear map cannot move the origin, so to handle translations (and combinations of them with rotations, scalings, and shears), we extend our toolkit to affine transformations. The secret weapon that makes this work is the idea of homogeneous coordinates.

What is an Affine Transformation?

An affine transformation is any map of the form:

$$f(\mathbf{x}) = A\mathbf{x} + \mathbf{b},$$

where A is a matrix (linear part) and $\mathbf{b}$ is a vector (translation part).

- A handles scaling, rotation, reflection, shear, or projection.
- $\mathbf{b}$ shifts everything by a constant amount.

Examples in 2D:

1. Rotate by 90° and then shift right by 2.
2. Stretch vertically by 3 and shift upward by 1.

Affine maps preserve parallel lines and ratios of distances along lines, but not necessarily angles or lengths.

Why Linear Maps Alone Aren't Enough

If we only use a 2×2 matrix in 2D or 3×3 in 3D, the origin always stays fixed. That's a limitation: real-world movements (like moving a shape from one place to another) require shifting the origin too. To capture both linear and translational effects uniformly, we need a clever trick.

Homogeneous Coordinates

The trick is to add one extra coordinate.

- In 2D, a point (x, y) becomes $(x, y, 1)$.
- In 3D, a point (x, y, z) becomes $(x, y, z, 1)$.

This new representation is called homogeneous coordinates. It allows us to fold translations into matrix multiplication.

Affine Transform as a Matrix in Homogeneous Form

In 2D:

$$\begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + t_x \\ cx + dy + t_y \\ 1 \end{bmatrix}.$$

Here,

- The 2×2 block $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ is the linear part.
- The last column $\begin{bmatrix} t_x \\ t_y \end{bmatrix}$ is the translation.

So with one unified matrix, we can handle both linear transformations and shifts.

Examples in 2D

1. Translation by (2, 3):

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}.$$

2. Scaling by 2 in x and 3 in y, then shifting by (–1, 4):

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 3 & 4 \\ 0 & 0 & 1 \end{bmatrix}.$$

3. Rotation by 90° and shift right by 5:

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

Homogeneous Coordinates in 3D

In 3D, affine transformations use 4×4 matrices. The upper-left 3×3 block handles rotation, scaling, or shear; the last column encodes translation.

Example: translation by (2, –1, 4):

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

This formulation is universal in computer graphics and robotics.

Why It Matters

1. Unified representation: Using homogeneous coordinates, we can treat translations as matrices, enabling consistent matrix multiplication for all transformations.
2. Practicality: This approach underpins 3D graphics pipelines, animation, CAD, robotics, and computer vision.
3. Composability: Multiple affine transformations can be combined into a single homogeneous matrix by multiplying them.
4. Geometry preserved: Affine maps preserve straight lines and parallelism, essential in engineering and design.

Try It Yourself

1. Write the homogeneous matrix that reflects across the x-axis and then shifts up by 3. Apply it to $(2, 1)$.
2. Construct a 4×4 homogeneous matrix that rotates around the z-axis by 90° and translates by (1, 2, 0).
3. Show that multiplying two 3×3 homogeneous matrices in 2D yields another valid affine transform.
4. Challenge: Prove that affine maps preserve parallel lines by applying a general affine matrix to two parallel lines and checking their slopes.

Mastering affine transformations and homogeneous coordinates bridges the gap between pure linear algebra and real-world geometry, giving you the mathematical foundation behind computer graphics, robotics, and spatial modeling.

20. Computing with Matrices (Cost Counts and Simple Speedups)

Thus far, we have studied what matrices are and what they represent. But in practice, working with matrices also means thinking about computation-how much work operations take, how algorithms can be sped up, and why structure matters. This section introduces the basic ideas of computational cost in matrix operations, simple strategies for efficiency, and why these considerations are crucial in modern applications.

Counting Operations: The Cost Model

The simplest way to measure the cost of a matrix operation is to count the basic arithmetic operations (additions and multiplications).

- Matrix–vector product: For an $m \times n$ matrix and an $n \times 1$ vector:
  - Each of the m output entries requires n multiplications and $n - 1$ additions.
  - Total cost $2mn$ operations.
- Matrix–matrix product: For an $m \times n$ matrix times an $n \times p$ matrix:
  - Each of the mp entries requires n multiplications and $n - 1$ additions.
  - Total cost $2mnp$ operations.
- Gaussian elimination (solving $Ax = b$): For an $n \times n$ system:
  - Roughly $\frac{2}{3}n^3$ operations.

These counts show how quickly costs grow with dimension. Doubling n makes the work 8 times larger for elimination.

Why Cost Counts Matter

1. Scalability: Small problems (2×2 or 3×3) are trivial, but modern datasets involve matrices with millions of rows. Knowing the cost is essential.
2. Feasibility: Some exact algorithms become impossible for very large matrices. Approximation methods are used instead.
3. Optimization: Engineers and scientists design specialized algorithms to reduce costs by exploiting structure (sparsity, symmetry, triangular form).

Simple Speedups with Structure

- Diagonal Matrices: Multiplying by a diagonal matrix costs only n operations (scale each component).
- Triangular Matrices: Solving triangular systems requires only $\frac{1}{2}n^2$ operations (substitution), far cheaper than general elimination.
- Sparse Matrices: If most entries are zero, we skip multiplications by zero. For large sparse systems, cost scales with the number of nonzeros, not n^2.
- Block Matrices: Breaking matrices into blocks allows algorithms to reuse optimized small-matrix routines (common in BLAS libraries).

Memory Considerations

Cost is not only arithmetic: storage also matters.

- A dense $n \times n$ matrix requires n^2 entries of memory.
- Sparse storage formats (like CSR, COO) record only nonzero entries and their positions, saving massive space.
- Memory access speed can dominate arithmetic cost in large computations.

Parallelism and Hardware

Modern computing leverages hardware for speed:

- Vectorization (SIMD): Perform many multiplications at once.
- Parallelization: Split work across many CPU cores.
- GPUs: Specialize in massive parallel matrix–vector and matrix–matrix operations (critical in deep learning).

This is why linear algebra libraries (BLAS, LAPACK, cuBLAS) are indispensable: they squeeze performance from hardware.

Why It Matters

1. Efficiency: Understanding cost lets us choose the right algorithm for the job.
2. Algorithm design: Structured matrices (diagonal, sparse, orthogonal) make computations much faster and more stable.
3. Applications: Every field that uses matrices-graphics, optimization, statistics, AI-relies on efficient computation.
4. Foundations: Later topics like LU/QR/SVD factorization are motivated by balancing cost and stability.

Try It Yourself

1. Compute the number of operations required for multiplying a 1000×500 matrix with a 500×200 matrix. Compare with multiplying a 1000×1000 dense matrix by a vector.
2. Show how solving a 3×3 triangular system is faster than Gaussian elimination. Count the exact multiplications and additions.
3. Construct a sparse 5×5 matrix with only 7 nonzero entries. Estimate the cost of multiplying it by a vector versus a dense 5×5 matrix.
4. Challenge: Suppose you need to store a 1,000,000×1,000,000 dense matrix. Estimate how much memory (in bytes) it would take if each entry is 8 bytes. Could it fit on a laptop? Why do sparse formats save the day?

By learning to count costs and exploit structure, you prepare yourself not only to understand matrices abstractly but also to use them effectively in real-world, large-scale problems. This balance between theory and computation is at the heart of modern linear algebra.

Closing

```
Patterns intertwine,
transformations gently fold,
structure in the square.
```

Chapter 3. Linear Systems and Elimination

21. From Equations to Matrices

Linear algebra often begins with systems of equations-collections of unknowns linked by linear relationships. While these systems can be solved directly using substitution or elimination, they quickly become messy when there are many variables. The key insight of linear algebra is that all systems of linear equations can be captured compactly by matrices and vectors. This section explains how we move from equations written out in words and symbols to the matrix form that powers computation.

A Simple Example

Consider this system of two equations in two unknowns:

$$\begin{cases} 2x + y = 5 \\ 3x - y = 4 \end{cases}$$

At first glance, this is just algebra: two equations, two unknowns. But notice the structure: each equation is a sum of multiples of the variables, set equal to a constant. This pattern-linear combinations of unknowns equal to a result-is exactly what matrices capture.

Writing in Coefficient Table Form

Extract the coefficients of each variable from the system:

- First equation: coefficients are 2 for x, 1 for y.
- Second equation: coefficients are 3 for x, -1 for y.

Arrange these coefficients in a rectangular array:

$$A = \begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix}.$$

This matrix A is called the coefficient matrix.

Next, write the unknowns as a vector:

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}.$$

Finally, write the right-hand side constants as another vector:

$$\mathbf{b} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}.$$

Now the entire system can be written in a single line:

$$A\mathbf{x} = \mathbf{b}.$$

Why This is Powerful

This compact form hides no information; it is equivalent to the original equations. But it gives us enormous advantages:

1. Clarity: We see the structure clearly-the system is "matrix times vector equals vector."
2. Scalability: Whether we have 2 equations or 2000, the same notation applies.
3. Tools: All the machinery of matrix operations (elimination, inverses, decompositions) now becomes available.

4. Geometry: The matrix equation $A\mathbf{x} = \mathbf{b}$ means: combine the columns of A (scaled by entries of x) to land on b.

A Larger Example

System of three equations in three unknowns:

$$\begin{cases} x + 2y - z = 2 \\ 2x - y + 3z = 1 \\ 3x + y + 2z = 4 \end{cases}$$

- Coefficient matrix:

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 2 & -1 & 3 \\ 3 & 1 & 2 \end{bmatrix}.$$

- Unknown vector:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

- Constant vector:

$$\mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}.$$

Matrix form:

$$A\mathbf{x} = \mathbf{b}.$$

This single equation captures three equations and three unknowns in one object.

Row vs. Column View

- Row view: Each row of A dotted with x gives one equation.
- Column view: The entire system means b is a linear combination of the columns of A.

For the 2×2 case earlier:

$$A\mathbf{x} = \begin{bmatrix} 2 & 1 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} 2 \\ 3 \end{bmatrix} + y \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

So solving the system means finding scalars x, y that combine the columns of A to reach $\mathbf{b}$.

Augmented Matrix Form

Sometimes we want to save space further. We can put the coefficients and constants side by side in an augmented matrix:

$$[A|\mathbf{b}] = \left[\begin{array}{cc|c} 2 & 1 & 5 \\ 3 & -1 & 4 \end{array}\right].$$

This form is especially useful for elimination methods, where we manipulate rows without writing variables at each step.

Why It Matters

This step-rewriting equations as matrix form-is the gateway into linear algebra. Once you can do it, you no longer think of systems of equations as isolated lines on paper, but as a unified object that can be studied with general tools. It opens the door to:

- Gaussian elimination,
- rank and null space,
- determinants,
- eigenvalues,
- optimization methods.

Every major idea flows from this compact representation.

Try It Yourself

1. Write the system

$$\begin{cases} 4x - y = 7 \\ -2x + 3y = 5 \end{cases}$$

   in matrix form.

2. For the system

$$\begin{cases} x + y + z = 6 \\ 2x - y + z = 3 \\ x - y - z = -2 \end{cases}$$

   build the coefficient matrix, unknown vector, and constant vector.

3. Express the augmented matrix for the above system.

4. Challenge: Interpret the system in column view. What does it mean geometrically to express $(6, 3, -2)$ as a linear combination of the columns of the coefficient matrix?

By practicing these rewrites, you will see that linear algebra is not about juggling many equations-it is about seeing structure in one compact equation. This step transforms scattered equations into the language of matrices, where the real power begins.

22. Row Operations

Once a system of linear equations has been expressed as a matrix, the next step is to simplify that matrix into a form where the solutions become clear. The main tool for this simplification is the set of elementary row operations. These operations allow us to manipulate the rows of a matrix in systematic ways that preserve the solution set of the corresponding system of equations.

The Three Types of Row Operations

There are exactly three types of legal row operations, each with a clear algebraic meaning:

1. Row Swapping ($R_i \leftrightarrow R_j$): Exchange two rows. This corresponds to reordering equations in a system. Since the order of equations doesn't change the solutions, this operation is always valid.

   Example:

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

2. Row Scaling ($R_i \to cR_i,\ c \neq 0$): Multiply all entries in a row by a nonzero constant. This is like multiplying both sides of an equation by the same number, which doesn't change its truth.

   Example:

$$\left[\begin{array}{cc|c} 2 & 1 & 5 \\ 3 & -1 & 4 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 1 & \frac{1}{2} & \frac{5}{2} \\ 3 & -1 & 4 \end{array}\right].$$

3. Row Replacement ($R_i \to R_i + cR_j$): Add a multiple of one row to another. This corresponds to replacing one equation with a linear combination of itself and another, a fundamental elimination step.

   Example:

$$\left[\begin{array}{cc|c} 2 & 1 & 5 \\ 3 & -1 & 4 \end{array}\right] \overset{R_2 \to R_2 - \frac{3}{2}R_1}{\longrightarrow} \left[\begin{array}{cc|c} 2 & 1 & 5 \\ 0 & -\frac{5}{2} & -\frac{7}{2} \end{array}\right].$$

Why These Are the Only Allowed Operations

These three operations are the backbone of elimination because they do not alter the solution set of the system. Each is equivalent to applying an invertible transformation:

- Row swaps are reversible (swap back).
- Row scalings by c can be undone by scaling by $1/c$.
- Row replacements can be undone by adding the opposite multiple.

Thus, each operation is invertible, and the transformed system is always equivalent to the original.

Row Operations as Matrices

Each elementary row operation can itself be represented by multiplying on the left with a special matrix called an elementary matrix.

For example:

- Swapping rows 1 and 2 in a 2×2 system is done by

$$E = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

- Scaling row 1 by 3 in a 2×2 system is done by

$$E = \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix}.$$

This perspective is crucial later for factorization methods like LU decomposition, where elimination is expressed as a product of elementary matrices.

Step-by-Step Example

System:

$$\begin{cases} x + 2y = 4 \\ 3x + 4y = 10 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 3 & 4 & 10 \end{array}\right].$$

1. Eliminate the $3x$ under the first pivot: $R_2 \to R_2 - 3R_1$.

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 0 & -2 & -2 \end{array}\right].$$

2. Scale the second row: $R_2 \to -\frac{1}{2}R_2$.

$$\left[\begin{array}{cc|c} 1 & 2 & 4 \\ 0 & 1 & 1 \end{array}\right].$$

3. Eliminate above the pivot: $R_1 \to R_1 - 2R_2$.

$$\left[\begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & 1 \end{array}\right].$$

Solution: $x = 2,\ y = 1$.

Geometry of Row Operations

Row operations do not alter the solution space:

- Swapping rows reorders equations but keeps the same lines or planes.
- Scaling rows rescales equations but leaves their geometric set unchanged.
- Adding rows corresponds to combining constraints, but the shared intersection (solution set) is preserved.

Thus, row operations act like "reshaping the system" while leaving the intersection intact.

Why It Matters

Row operations are the essential moves in solving linear systems by hand or computer. They:

1. Make elimination systematic.
2. Preserve solution sets while simplifying structure.
3. Lay the groundwork for echelon forms, rank, and factorization.
4. Provide the mechanical steps that computers automate in Gaussian elimination.

Try It Yourself

1. Apply row operations to reduce

$$\left[\begin{array}{cc|c} 2 & 1 & 7 \\ 1 & -1 & 1 \end{array}\right]$$

   to a form where the solution is obvious.

2. Show explicitly why swapping two equations in a system doesn't change its solutions.
3. Construct the elementary matrix for "add –2 times row 1 to row 3" in a 3×3 system.
4. Challenge: Prove that any elementary row operation corresponds to multiplication by an invertible matrix.

Mastering these operations equips you with the mechanical and conceptual foundation for the next stage: systematically reducing matrices to row-echelon form.

23. Row-Echelon and Reduced Row-Echelon Forms

After introducing row operations, the natural question is: *what are we trying to achieve by performing them?* The answer is to transform a matrix into a standardized, simplified form where the solutions to the corresponding system of equations can be read off directly. Two such standardized forms are central in linear algebra: row-echelon form (REF) and reduced row-echelon form (RREF).

Row-Echelon Form (REF)

A matrix is in row-echelon form if:

1. All nonzero rows are above any rows of all zeros.
2. In each nonzero row, the first nonzero entry (called the leading entry or pivot) is to the right of the leading entry of the row above it.
3. All entries below a pivot are zero.

Example of REF:

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

Here, the pivots are the first 1 in row 1, the 1 in row 2, and the 5 in row 3. Each pivot is to the right of the one above it, and all entries below pivots are zero.

Reduced Row-Echelon Form (RREF)

A matrix is in reduced row-echelon form if, in addition to the rules of REF:

1. Each pivot is equal to 1.
2. Each pivot is the only nonzero entry in its column (everything above and below pivots is zero).

Example of RREF:

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

This form is so simplified that solutions can be read directly: here, $x = 3$, $y = -2$, $z = 1$.

Relationship Between REF and RREF

- REF is easier to reach-it only requires eliminating entries below pivots.
- RREF requires going further-clearing entries above pivots and scaling pivots to 1.
- Every matrix can be reduced to REF (many possible versions), but RREF is unique: no matter how you proceed, if you carry out all row operations fully, you end with the same RREF.

Example: Step-by-Step to RREF

System:

$$\begin{cases} x + 2y + z = 4 \\ 2x + 5y + z = 7 \\ 3x + 6y + 2z = 10 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{ccc|c} 1 & 2 & 1 & 4 \\ 2 & 5 & 1 & 7 \\ 3 & 6 & 2 & 10 \end{array}\right].$$

1. Eliminate below first pivot (the 1 in row 1, col 1):

   - $R_2 \to R_2 - 2R_1$
   - $R_3 \to R_3 - 3R_1$

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

   This is now in REF.

2. Scale pivots and eliminate above them:

   - $R_3 \to -R_3$ to make pivot 1.
   - $R_2 \to R_2 + R_3$.
   - $R_1 \to R_1 - R_2 - R_3$.

   Final:

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right].$$

Solution: $x = 2, y = 1, z = 2$.

Geometry of REF and RREF

- REF corresponds to simplifying the system step by step, making it "triangular" so variables can be solved one after another.
- RREF corresponds to a system that is fully disentangled-each variable isolated, with its value or free-variable relationship explicitly visible.

Why It Matters

1. REF is the foundation of Gaussian elimination, the workhorse algorithm for solving systems.
2. RREF gives complete clarity: unique representation of solution sets, revealing free and pivot variables.
3. RREF underlies algorithms in computer algebra systems, symbolic solvers, and educational tools.
4. Understanding these forms builds intuition for rank, null space, and solution structure.

Try It Yourself

1. Reduce

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

   to REF, then RREF.

2. Find the RREF of

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 2 & 3 & 4 & 8 \\ 1 & 2 & 3 & 5 \end{array}\right].$$

3. Explain why two different elimination sequences can lead to different REF but the same RREF.
4. Challenge: Prove that every matrix has a unique RREF by considering the effect of row operations systematically.

Reaching row-echelon and reduced row-echelon forms transforms messy systems into structured ones, turning algebraic clutter into an organized path to solutions.

24. Pivots, Free Variables, and Leading Ones

When reducing a matrix to row-echelon or reduced row-echelon form, certain positions in the matrix take on a special importance. These are the pivots-the leading nonzero entries in each row. Around them, the entire solution structure of a linear system is organized. Understanding pivots, the variables they anchor, and the freedom that arises from non-pivot columns is essential to solving linear equations systematically.

What is a Pivot?

In row-echelon form, a pivot is the first nonzero entry in a row, moving from left to right. After scaling in reduced row-echelon form, each pivot is set to exactly 1.

Example:

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

- Pivot in row 1: the 1 in column 1.
- Pivot in row 2: the 1 in column 2.
- Column 3 has no pivot.

Columns with pivots are pivot columns. Columns without pivots correspond to free variables.

Pivot Variables vs. Free Variables

- Pivot variables: Variables that align with pivot columns. They are determined by the equations.
- Free variables: Variables that align with non-pivot columns. They are unconstrained and can take arbitrary values.

Example:

$$\begin{bmatrix} 1 & 0 & 2 & | & 3 \\ 0 & 1 & -1 & | & 4 \end{bmatrix}.$$

This corresponds to:

$$x_1 + 2x_3 = 3, \quad x_2 - x_3 = 4.$$

Here:

- x_1 and x_2 are pivot variables (from pivot columns 1 and 2).
- x_3 is a free variable.

Thus, x_1 and x_2 depend on x_3:

$$x_1 = 3 - 2x_3, \quad x_2 = 4 + x_3, \quad x_3 \text{ free.}$$

The solution set is infinite, described by the freedom in x_3.

Geometric Meaning

- Pivot variables represent coordinates that are "pinned down."
- Free variables correspond to directions along which the solution can extend infinitely.

In 2D:

- If there is one pivot variable and one free variable, solutions form a line. In 3D:
- Two pivots, one free $\rightarrow$ solutions form a line.
- One pivot, two free $\rightarrow$ solutions form a plane.

Thus, the number of free variables determines the dimension of the solution set.

Rank and Free Variables

The number of pivot columns equals the rank of the matrix.

If the coefficient matrix A is $m \times n$:

- Rank = number of pivots.
- Number of free variables = $n - \text{rank}(A)$.

This is the rank–nullity connection in action:

$$\text{number of variables} = \text{rank} + \text{nullity}.$$

Step-by-Step Example

System:

$$\begin{cases} x + 2y + z = 4 \\ 2x + 5y + z = 7 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{ccc|c} 1 & 2 & 1 & 4 \\ 2 & 5 & 1 & 7 \end{array}\right].$$

Reduce:

- $R_2 \to R_2 - 2R_1 \to$

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

Now:

- Pivot columns: 1 and 2 $\to$ variables x, y.
- Free column: 3 $\to$ variable z.

Solution:

$$x = 4 - 2y - z, \quad y = -1 + z, \quad z \text{ free.}$$

Substitute:

$$(x, y, z) = (6 - 3z,\ -1 + z,\ z).$$

Solutions form a line in 3D parameterized by z.

Why Leading Ones Matter

In RREF, each pivot is scaled to 1, making it easy to isolate pivot variables. Without leading ones, equations may still be correct but harder to interpret.

For example:

$$\left[\begin{array}{cc|c} 2 & 0 & 6 \\ 0 & -3 & 9 \end{array}\right]$$

becomes

$$\left[\begin{array}{cc|c} 1 & 0 & 3 \\ 0 & 1 & -3 \end{array}\right].$$

The solutions are immediately visible: $x = 3, y = -3$.

Why It Matters

1. Identifying pivots shows which variables are determined and which are free.
2. The number of pivots defines rank, a central concept in linear algebra.
3. Free variables determine whether the system has a unique solution, infinitely many, or none.
4. Leading ones in RREF give immediate transparency to the solution set.

Try It Yourself

1. Reduce

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 5 \\ 2 & 6 & 2 & 10 \end{array}\right]$$

   and identify pivot and free variables.

2. For the system

$$x + y + z = 2, \quad 2x + 3y + 5z = 7,$$

   write the RREF and express the solution with free variables.

3. Compute the rank and number of free variables of a 3×5 matrix with two pivot columns.

4. Challenge: Show that if the number of pivots equals the number of variables, the system has either no solution or a unique solution, but never infinitely many.

Understanding pivots and free variables provides the key to classifying solution sets: unique, infinite, or none. This classification lies at the heart of solving linear systems.

25. Solving Consistent Systems

A system of linear equations is called consistent if it has at least one solution. Consistency is the first property to check when working with a system, because before worrying about uniqueness or parametrization, we must know whether a solution exists at all. This section explains how to recognize consistent systems, how to solve them using row-reduction, and how to describe their solutions in terms of pivots and free variables.

What Consistency Means

Given a system $A\mathbf{x} = \mathbf{b}$:

- Consistent: At least one solution $\mathbf{x}$ satisfies the system.
- Inconsistent: No solution exists.

Consistency depends on the relationship between the vector $\mathbf{b}$ and the column space of A:

$$\mathbf{b} \in \mathrm{Col}(A) \quad \Longleftrightarrow \quad \text{system is consistent.}$$

If $\mathbf{b}$ cannot be written as a linear combination of the columns of A, the system has no solution.

Checking Consistency with Row Reduction

To test consistency, reduce the augmented matrix $[A|\mathbf{b}]$ to row-echelon form.

- If you find a row of the form:

$$[0 \;\; 0 \;\; \ldots \;\; 0 \mid c], \quad c \neq 0,$$

  then the system is inconsistent (contradiction: 0 = c).
- If no such contradiction appears, the system is consistent.

Example 1: Consistent System with Unique Solution

System:

$$\begin{cases} x + y = 2 \\ x - y = 0 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 1 & -1 & 0 \end{array}\right].$$

Row reduce:

- $R_2 \to R_2 - R_1$:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & -2 & -2 \end{array}\right].$$

- $R_2 \to -\frac{1}{2}R_2$:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 1 & 1 \end{array}\right].$$

- $R_1 \to R_1 - R_2$:

$$\left[\begin{array}{cc|c} 1 & 0 & 1 \\ 0 & 1 & 1 \end{array}\right].$$

Solution: $x = 1,\ y = 1$. Unique solution.

Example 2: Consistent System with Infinitely Many Solutions

System:

$$\begin{cases} x + y + z = 3 \\ 2x + 2y + 2z = 6 \end{cases}$$

Augmented matrix:

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

Row reduce:

- $R_2 \to R_2 - 2R_1$:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 \end{array}\right].$$

No contradiction, so consistent. Solution:

$$x = 3 - y - z, \quad y \text{ free}, \quad z \text{ free}.$$

The solution set is a plane in $\mathbb{R}^3$.

Example 3: Inconsistent System (for contrast)

System:

$$\begin{cases} x + y = 1 \\ x + y = 2 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 1 & 1 \\ 1 & 1 & 2 \end{array}\right].$$

Row reduce:

- $R_2 \to R_2 - R_1$:

$$\left[\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & 0 & 1 \end{array}\right].$$

Contradiction: $0 = 1$. Inconsistent, no solution.

Geometric Interpretation of Consistency

- In 2D:
  - Two lines intersect at a point → consistent, unique solution.
  - Two lines overlap → consistent, infinitely many solutions.
  - Two lines are parallel and distinct → inconsistent, no solution.
- In 3D:
  - Three planes intersect at a point → unique solution.
  - Planes intersect along a line or coincide → infinitely many solutions.
  - Planes fail to meet (like a triangular "gap") → no solution.

Pivot Structure and Solutions

- Unique solution: Every variable is a pivot variable (no free variables).
- Infinitely many solutions: At least one free variable exists, but no contradiction.
- No solution: Contradictory row appears in augmented matrix.

Why It Matters

1. Consistency is the first checkpoint in solving systems.
2. The classification into unique, infinite, or none underpins all of linear algebra.
3. Understanding consistency ties algebra (row operations) to geometry (intersections of lines, planes, hyperplanes).
4. These ideas scale: in data science and engineering, checking whether equations are consistent is equivalent to asking if a model fits observed data.

Try It Yourself

1. Reduce the augmented matrix

$$\left[\begin{array}{ccc|c} 1 & 2 & 1 & 5 \\ 2 & 4 & 2 & 10 \\ 3 & 6 & 3 & 15 \end{array}\right]$$

   and determine if the system is consistent.

2. Classify the system as having unique, infinite, or no solutions:

$$\begin{cases} x + y + z = 2 \\ x - y + z = 0 \\ 2x + 0y + 2z = 3 \end{cases}$$

3. Explain geometrically what it means when the augmented matrix has a contradictory row.

4. Challenge: Show algebraically that a system is consistent if and only if $\mathbf{b}$ lies in the span of the columns of A.

Consistent systems mark the balance point between algebraic rules and geometric reality: they are where equations and space meet in harmony.

26. Detecting Inconsistency

Not every system of linear equations has a solution. Some are inconsistent, meaning the equations contradict one another and no vector $\mathbf{x}$ can satisfy them all at once. Detecting such inconsistency early is crucial: it saves wasted effort trying to solve an impossible system and reveals important geometric and algebraic properties.

What Inconsistency Looks Like Algebraically

Consider the system:

$$\begin{cases} x + y = 1 \\ x + y = 3 \end{cases}$$

Clearly, the two equations cannot both be true. In augmented matrix form:

$$\left[\begin{array}{cc|c} 1 & 1 & 1 \\ 1 & 1 & 3 \end{array}\right].$$

Row reduction gives:

$$\left[\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & 0 & 2 \end{array}\right].$$

The bottom row says $0 = 2$, a contradiction. This is the hallmark of inconsistency: a row of zeros in the coefficient part, with a nonzero constant in the augmented part.

General Rule for Detection

A system $A\mathbf{x} = \mathbf{b}$ is inconsistent if, after row reduction, the augmented matrix contains a row of the form:

$$[0 \;\; 0 \;\; \ldots \;\; 0 \mid c], \quad c \neq 0.$$

This indicates that all variables vanish from the equation, leaving an impossible statement like $0 = c$.

Example 1: Parallel Lines in 2D

$$\begin{cases} x + y = 2 \\ 2x + 2y = 5 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 2 & 2 & 5 \end{array}\right].$$

Row reduce:

- $R_2 \to R_2 - 2R_1$:

$$\left[\begin{array}{cc|c} 1 & 1 & 2 \\ 0 & 0 & 1 \end{array}\right].$$

Contradiction: no solution. Geometrically, the two equations are parallel lines that never intersect.

Example 2: Contradictory Planes in 3D

$$\begin{cases} x + y + z = 1 \\ 2x + 2y + 2z = 2 \\ x + y + z = 3 \end{cases}$$

The first and third equations already conflict: the same plane equation is forced to equal two different constants.

Augmented matrix reduces to:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{array}\right].$$

Contradiction: no solution. The "planes" fail to intersect in common.

Geometry of Inconsistency

- In 2D: Inconsistent systems correspond to parallel lines with different intercepts.
- In 3D: They correspond to planes that are parallel but offset, or planes arranged in a way that leaves a "gap" (no shared intersection).
- In higher dimensions: Inconsistency means the target vector $\mathbf{b}$ lies outside the column space of A.

Rank Test for Consistency

Another way to detect inconsistency is using ranks.

- Let $\text{rank}(A)$ be the number of pivots in the coefficient matrix.
- Let $\text{rank}([A|\mathbf{b}])$ be the number of pivots in the augmented matrix.

Rule:

- If $\text{rank}(A) = \text{rank}([A|\mathbf{b}])$, the system is consistent.
- If $\text{rank}(A) < \text{rank}([A|\mathbf{b}])$, the system is inconsistent.

This rank condition is fundamental and works in any dimension.

Why It Matters

1. Inconsistency reveals overdetermined or contradictory data in real problems (physics, engineering, statistics).
2. The ability to detect inconsistency quickly through row reduction or rank saves computation.
3. It connects geometry (non-intersecting spaces) with algebra (contradictory rows).
4. It prepares the way for least-squares methods, where inconsistent systems are approximated instead of solved exactly.

Try It Yourself

1. Reduce the augmented matrix

$$\left[\begin{array}{cc|c} 1 & -1 & 2 \\ 2 & -2 & 5 \end{array}\right]$$

and decide if the system is consistent.

2. Show geometrically why the system

$$x + y = 0, \quad x + y = 1$$

is inconsistent.

3. Use the rank test to check consistency of

$$\begin{cases} x + y + z = 2 \\ 2x + 2y + 2z = 4 \\ 3x + 3y + 3z = 5 \end{cases}$$

4. Challenge: Explain why $\operatorname{rank}(A) < \operatorname{rank}([A|\mathbf{b}])$ implies inconsistency, using the concept of the column space.

Detecting inconsistency is not just about spotting contradictions-it connects algebra, geometry, and linear transformations, showing exactly when a system cannot possibly fit together.

27. Gaussian Elimination by Hand

Gaussian elimination is the systematic procedure for solving systems of linear equations by using row operations to simplify the augmented matrix. The goal is to transform the system into row-echelon form (REF) and then use back substitution to find the solutions. This method is the backbone of linear algebra computations and is the foundation of most computer algorithms for solving linear systems.

The Big Idea

1. Represent the system as an augmented matrix.
2. Use row operations to eliminate variables step by step, moving left to right, top to bottom.
3. Stop when the matrix is in REF.
4. Solve the triangular system by back substitution.

Step-by-Step Recipe

Suppose we have n equations with n unknowns.

1. Choose a pivot in the first column (a nonzero entry). If needed, swap rows to bring a nonzero entry to the top.
2. Eliminate below the pivot by subtracting multiples of the pivot row from lower rows so that all entries below the pivot become zero.
3. Move to the next row and next column, pick the next pivot, and repeat elimination.
4. Continue until all pivots are in stair-step form (REF).
5. Use back substitution to solve for the unknowns starting from the bottom row.

Example 1: A 2×2 System

System:

$$\begin{cases} x + 2y = 5 \\ 3x + 4y = 11 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{cc|c} 1 & 2 & 5 \\ 3 & 4 & 11 \end{array}\right].$$

1. Pivot at (1,1) = 1.

2. Eliminate below: $R_2 \to R_2 - 3R_1$.

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

3. Back substitution: From row 2: $-2y = -4 \implies y = 2$. Substitute into row 1: $x + 2(2) = 5 \implies x = 1$.

Solution: $(x, y) = (1, 2)$.

Example 2: A 3×3 System

System:

$$\begin{cases} x + y + z = 6 \\ 2x + 3y + z = 14 \\ x - y + 2z = 2 \end{cases}$$

Augmented matrix:

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 2 & 3 & 1 & 14 \\ 1 & -1 & 2 & 2 \end{array}\right].$$

Step 1: Pivot at (1,1). Eliminate below:

- $R_2 \to R_2 - 2R_1$.
- $R_3 \to R_3 - R_1$.

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

Step 2: Pivot at (2,2). Eliminate below: $R_3 \to R_3 + 2R_2$.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 0 & 1 & -1 & 2 \\ 0 & 0 & -1 & 0 \end{array}\right].$$

Step 3: Pivot at (3,3). Scale row: $R_3 \to -R_3$.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 6 \\ 0 & 1 & -1 & 2 \\ 0 & 0 & 1 & 0 \end{array}\right].$$

Back substitution:

- From row 3: $z = 0$.
- From row 2: $y - z = 2 \implies y = 2$.
- From row 1: $x + y + z = 6 \implies x = 4$.

Solution: $(x, y, z) = (4, 2, 0)$.

Why Gaussian Elimination Always Works

- Each step reduces the number of variables in the lower equations.
- Pivoting ensures stability (swap rows to avoid dividing by zero).
- The algorithm either produces a triangular system (solvable by substitution) or reveals inconsistency (contradictory row).

Geometric Interpretation

- Elimination corresponds to progressively restricting the solution set:
  - First equation → a plane in $\mathbb{R}^3$.
  - Add second equation → intersection becomes a line.
  - Add third equation → intersection becomes a point (unique solution) or vanishes (inconsistent).

Why It Matters

1. Gaussian elimination is the foundation for solving systems by hand and by computer.
2. It reveals whether a system is consistent and if solutions are unique or infinite.
3. It is the starting point for advanced methods like LU decomposition, QR factorization, and numerical solvers.
4. It shows the interplay between algebra (row operations) and geometry (intersections of subspaces).

Try It Yourself

1. Solve the system

$$\begin{cases} 2x + y = 7 \\ 4x + 3y = 15 \end{cases}$$

   using Gaussian elimination.

2. Reduce

$$\left[\begin{array}{ccc|c} 1 & 2 & -1 & 3 \\ 3 & 8 & 1 & 12 \\ 2 & 6 & 3 & 11 \end{array}\right]$$

   to REF and solve.

3. Practice with a system that has infinitely many solutions:

$$x + y + z = 4, \quad 2x + 2y + 2z = 8.$$

4. Challenge: Explain why Gaussian elimination always terminates in at most n pivot steps for an $n \times n$ system.

Gaussian elimination transforms the complexity of many equations into an orderly process, making the hidden structure of solutions visible step by step.

28. Back Substitution and Solution Sets

Once Gaussian elimination reduces a system to row-echelon form (REF), the next step is to actually solve for the unknowns. This process is called back substitution: we begin with the bottom equation (which involves the fewest variables) and work our way upward, solving step by step. Back substitution is what converts the structured triangular system into explicit solutions.

The Structure of Row-Echelon Form

A system in REF looks like this:

$$\left[\begin{array}{cccc|c} - & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \end{array}\right]$$

- Each row corresponds to an equation with fewer variables than the row above.
- The bottom equation has only one or two variables.
- This triangular form makes it possible to solve "from the bottom up."

Step-by-Step Example: Unique Solution

System after elimination:

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

This corresponds to:

$$\begin{cases} x + 2y - z = 3 \\ y + 2z = 4 \\ z = 2 \end{cases}$$

1. From the last equation: $z = 2$.
2. Substitute into the second: $y + 2(2) = 4 \implies y = 0$.
3. Substitute into the first: $x + 2(0) - 2 = 3 \implies x = 5$.

Solution: $(x, y, z) = (5, 0, 2)$.

Infinite Solutions with Free Variables

Not all systems reduce to unique solutions. If there are free variables (non-pivot columns), back substitution expresses pivot variables in terms of free ones.

Example:

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

Equations:

$$\begin{cases} x + 2y + z = 4 \\ y - z = 1 \end{cases}$$

1. From row 2: $y = 1 + z$.
2. From row 1: $x + 2(1 + z) + z = 4 \implies x = 2 - 3z$.

Solution set:

$$(x, y, z) = (2 - 3t,\ 1 + t,\ t), \quad t \in \mathbb{R}.$$

Here $z = t$ is the free variable. The solutions form a line in 3D.

General Solution Structure

For a consistent system:

1. Unique solution → every variable is a pivot variable (no free variables).
2. Infinitely many solutions → some free variables remain. The solution set is parametrized by these variables and forms a line, plane, or higher-dimensional subspace.
3. No solution → contradiction discovered earlier, so back substitution is impossible.

Geometric Meaning

- Unique solution → a single intersection point of lines/planes.
- Infinite solutions → overlapping subspaces (e.g., two planes intersecting in a line).
- Back substitution describes the exact shape of this intersection.

Example: Parametric Vector Form

For the infinite-solution example above:

$$(x, y, z) = (2, 1, 0) + t(-3, 1, 1).$$

This expresses the solution set as a base point plus a direction vector, making the geometry clear.

Why It Matters

1. Back substitution turns row-echelon form into concrete answers.
2. It distinguishes unique vs. infinite solutions.
3. It provides a systematic method usable by hand for small systems and forms the basis of computer algorithms for large ones.
4. It reveals the structure of solution sets-whether a point, line, plane, or higher-dimensional object.

Try It Yourself

1. Solve by back substitution:

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

2. Reduce and solve:

$$x + y + z = 2, \quad 2x + 2y + 2z = 4.$$

3. Express the solution set of the above system in parametric vector form.
4. Challenge: For a 4×4 system with two free variables, explain why the solution set forms a plane in $\mathbb{R}^4$.

Back substitution completes the elimination process, translating triangular structure into explicit solutions, and shows how algebra and geometry meet in the classification of solution sets.

29. Rank and Its First Meaning

The concept of rank lies at the heart of linear algebra. It connects the algebra of solving systems, the geometry of subspaces, and the structure of matrices into one unifying idea. Rank measures the amount of independent information in a matrix: how many rows or columns carry unique directions instead of being repetitions or combinations of others.

Definition of Rank

The rank of a matrix A is the number of pivots in its row-echelon form. Equivalently, it is:

- The dimension of the column space (number of independent columns).
- The dimension of the row space (number of independent rows).

All these definitions agree.

First Encounter with Rank: Pivot Counting

When solving a system with Gaussian elimination:

- Every pivot corresponds to one determined variable.
- The number of pivots = the rank.
- The number of free variables = total variables – rank.

Example:

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

Here, there are 2 pivots. So:

- Rank = 2.
- With 3 variables total, there is 1 free variable.

Rank in Terms of Independence

A set of vectors is linearly independent if none can be expressed as a combination of the others.

- The rank of a matrix tells us how many independent rows or columns it has.
- If some columns are combinations of others, they do not increase the rank.

Example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}.$$

Here, each row is a multiple of the first. Rank = 1, since only one independent row/column direction exists.

Rank and Solutions of Systems

Consider $A\mathbf{x} = \mathbf{b}$.

- If $\text{rank}(A) = \text{rank}([A|\mathbf{b}])$, the system is consistent.
- If not, inconsistent.
- If rank = number of variables, the system has a unique solution.
- If rank $<$ number of variables, there are infinitely many solutions.

Thus, rank classifies solution sets.

Rank and Geometry

Rank tells us the dimension of the subspace spanned by rows or columns.

- Rank 1: all information lies along a line.
- Rank 2: lies in a plane.
- Rank 3: fills 3D space.

Example:

- In $\mathbb{R}^3$, a matrix of rank 2 has columns spanning a plane through the origin.
- A matrix of rank 1 has all columns on a single line.

Rank and Row vs. Column View

It is a remarkable fact that the number of independent rows = number of independent columns. This is not obvious at first glance, but it is always true. So we can define rank either by rows or by columns-it makes no difference.

Why It Matters

1. Rank is the bridge between algebra and geometry: pivots dimension.
2. It classifies solutions to systems of equations.
3. It measures redundancy in data (important in statistics, machine learning, signal processing).
4. It prepares the way for advanced concepts like nullity, rank–nullity theorem, and singular value decomposition.

Try It Yourself

1. Find the rank of

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 6 & 8 \end{bmatrix}.$$

2. Solve the system

$$x + y + z = 2, \quad 2x + 2y + 2z = 4,$$

   and identify the rank of the coefficient matrix.

3. In $\mathbb{R}^3$, what is the geometric meaning of a 3×3 matrix of rank 2?

4. Challenge: Prove that the row rank always equals the column rank by considering the echelon form of the matrix.

Rank is the first truly unifying concept in linear algebra: it tells us how much independent structure a matrix contains and sets the stage for understanding spaces, dimensions, and transformations.

30. LU Factorization

Gaussian elimination not only solves systems but also reveals a deeper structure: many matrices can be factored into simpler pieces. One of the most useful is the LU factorization, where a matrix A is written as the product of a lower-triangular matrix L and an upper-triangular matrix U. This factorization captures all the elimination steps in a compact form and allows systems to be solved efficiently.

What is LU Factorization?

If A is an $n \times n$ matrix, then

$$A = LU,$$

where:

- L is lower-triangular (entries below diagonal may be nonzero, diagonal entries = 1).
- U is upper-triangular (entries above diagonal may be nonzero).

This means:

- U stores the result of elimination (the triangular system).
- L records the multipliers used during elimination.

Example: 2×2 Case

Take

$$A = \begin{bmatrix} 2 & 3 \\ 4 & 7 \end{bmatrix}.$$

Elimination: $R_2 \to R_2 - 2R_1$.

- Multiplier = 2 (used to eliminate entry 4).
- Resulting U:

$$U = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}.$$

- L:

$$L = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}.$$

Check:

$$LU = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 4 & 7 \end{bmatrix} = A.$$

Example: 3×3 Case

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{bmatrix}.$$

Step 1: Eliminate below pivot (row 1).

- Multiplier $m_{21} = 4/2 = 2$.
- Multiplier $m_{31} = -2/2 = -1$.

Step 2: Eliminate below pivot in column 2.

- After substitutions, multipliers and pivots are collected.

Result:

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 2 & 1 & 1 \\ 0 & -8 & -2 \\ 0 & 0 & 1 \end{bmatrix}.$$

Thus $A = LU$.

Solving Systems with LU

Suppose $Ax = b$. If $A = LU$:

1. Solve $Ly = b$ by forward substitution (since L is lower-triangular).
2. Solve $Ux = y$ by back substitution (since U is upper-triangular).

This two-step process is much faster than elimination from scratch each time, especially if solving multiple systems with the same A but different b.

Pivoting and Permutations

Sometimes elimination requires row swaps (to avoid division by zero or instability). Then factorization is written as:

$$PA = LU,$$

where P is a permutation matrix recording the row swaps. This is the practical form used in numerical computing.

Applications of LU Factorization

1. Efficient solving: Multiple right-hand sides $Ax = b$. Compute LU once, reuse for each b.
2. Determinants: $\det(A) = \det(L)\det(U)$. Since diagonals of L are 1, this reduces to the product of the diagonal of U.
3. Matrix inverse: By solving $Ax = e_i$ for each column e_i, we can compute A^{-1} efficiently with LU.
4. Numerical methods: LU is central in scientific computing, engineering simulations, and optimization.

Geometric Meaning

LU decomposition separates the elimination process into:

- L: shear transformations (adding multiples of rows).
- U: scaling and alignment into triangular form.

Together, they represent the same linear transformation as A, but decomposed into simpler building blocks.

Why It Matters

1. LU factorization compresses elimination into a reusable format.
2. It is a cornerstone of numerical linear algebra and used in almost every solver.
3. It links computation (efficient algorithms) with theory (factorization of transformations).
4. It introduces the broader idea that matrices can be broken into simple, interpretable parts.

Try It Yourself

1. Factor

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 8 \end{bmatrix}$$

   into LU.

2. Solve

$$\begin{bmatrix} 2 & 1 \\ 6 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 15 \end{bmatrix}$$

   using LU decomposition.

3. Compute $\det(A)$ for

$$A = \begin{bmatrix} 2 & 1 & 1 \\ 4 & -6 & 0 \\ -2 & 7 & 2 \end{bmatrix}$$

   by using its LU factorization.

4. Challenge: Prove that if A is invertible, then it has an LU factorization (possibly after row swaps).

LU factorization organizes elimination into a powerful tool: compact, efficient, and deeply tied to both the theory and practice of linear algebra.

Closing

```
Paths diverge or merge,
pivots mark the way forward,
truth distilled in rows.
```

Chapter 4. Vector spaces and subspaces

Opening

```
Endless skies expand,
spaces within spaces grow,
freedom takes its shape.
```

31. Axioms of Vector Spaces

Up to now, we have worked with vectors in $\mathbb{R}^2$, $\mathbb{R}^3$, and higher-dimensional Euclidean spaces. But the true power of linear algebra comes from abstracting away from coordinates. A vector space is not tied to arrows in physical space-it is any collection of objects that behave like vectors, provided they satisfy certain rules. These rules are called the axioms of vector spaces.

The Idea of a Vector Space

A vector space is a set V equipped with two operations:

1. Vector addition: Combine two vectors in V to get another vector in V.
2. Scalar multiplication: Multiply a vector in V by a scalar (a number from a field, usually $\mathbb{R}$ or $\mathbb{C}$).

The magic is that as long as certain rules (axioms) hold, the objects in V can be treated as vectors. They need not be arrows or coordinate lists-they could be polynomials, functions, matrices, or sequences.

The Eight Axioms

Let $u, v, w \in V$ (vectors) and $a, b \in \mathbb{R}$ (scalars). The axioms are:

1. Closure under addition: $u + v \in V$.
2. Commutativity of addition: $u + v = v + u$.
3. Associativity of addition: $(u + v) + w = u + (v + w)$.
4. Existence of additive identity: There exists a zero vector $0 \in V$ such that $v + 0 = v$.
5. Existence of additive inverses: For every v, there is $-v$ such that $v + (-v) = 0$.
6. Closure under scalar multiplication: $av \in V$.
7. Distributivity of scalar multiplication over vector addition: $a(u + v) = au + av$.
8. Distributivity of scalar multiplication over scalar addition: $(a + b)v = av + bv$.
9. Associativity of scalar multiplication: $a(bv) = (ab)v$.
10. Existence of multiplicative identity: $1 \cdot v = v$.

(These are sometimes listed as eight, with some grouped together, but the essence is the same.)

Examples of Vector Spaces

1. Euclidean spaces: $\mathbb{R}^n$ with standard addition and scalar multiplication.
2. Polynomials: The set of all polynomials with real coefficients, $\mathbb{R}[x]$.
3. Functions: The set of all continuous functions on $[0, 1]$, with addition of functions and scalar multiplication.
4. Matrices: The set of all $m \times n$ matrices with real entries.
5. Sequences: The set of all infinite real sequences $(a_1, a_2, \dots)$.

All of these satisfy the vector space axioms.

Non-Examples

1. The set of natural numbers $\mathbb{N}$ is not a vector space (no additive inverses).
2. The set of positive real numbers $\mathbb{R}^+$ is not a vector space (not closed under scalar multiplication with negative numbers).
3. The set of polynomials of degree exactly 2 is not a vector space (not closed under addition: $x^2 + x^2 = 2x^2$ is still degree 2, but $x^2 - x^2 = 0$, which is degree 0, not allowed).

These examples show why the axioms are essential: without them, the structure breaks.

The Zero Vector

Every vector space must contain a zero vector. This is not optional. It is the "do nothing" element for addition. In $\mathbb{R}^n$, this is $(0, 0, \dots, 0)$. In polynomials, it is the zero polynomial. In function spaces, it is the function $f(x) = 0$.

Additive Inverses

For every vector v, we require $-v$. This ensures that equations like $u + v = w$ can always be rearranged to $u = w - v$. Without additive inverses, solving linear equations would not work.

Scalars and Fields

Scalars come from a field: usually the real numbers $\mathbb{R}$ or the complex numbers $\mathbb{C}$. The choice of scalars matters:

- Over $\mathbb{R}$, a polynomial space is different from over $\mathbb{C}$.
- Over finite fields (like integers modulo p), vector spaces exist in discrete mathematics and coding theory.

Geometric Interpretation

- The axioms guarantee that vectors can be added and scaled in predictable ways.
- Closure ensures the space is "self-contained."
- Additive inverses ensure symmetry: every direction can be reversed.
- Distributivity ensures consistency between scaling and addition.

Together, these rules make vector spaces stable and reliable mathematical objects.

Why It Matters

1. Vector spaces unify many areas of math under a single framework.
2. They generalize $\mathbb{R}^n$ to functions, polynomials, and beyond.
3. The axioms guarantee that all the tools of linear algebra-span, basis, dimension, linear maps-apply.
4. Recognizing vector spaces in disguise is a major step in advanced math and physics.

Try It Yourself

1. Verify that the set of all 2×2 matrices is a vector space under matrix addition and scalar multiplication.
2. Show that the set of polynomials of degree at most 3 is a vector space, but the set of polynomials of degree exactly 3 is not.
3. Check whether the set of all even functions $f(-x) = f(x)$ is a vector space.
4. Challenge: Consider the set of all differentiable functions f on $[0, 1]$. Show that this set forms a vector space under the usual operations.

The axioms of vector spaces provide the foundation on which the rest of linear algebra is built. Everything that follows-subspaces, independence, basis, dimension-grows naturally from this formal framework.

32. Subspaces, Column Space, and Null Space

Once the idea of a vector space is in place, the next step is to recognize smaller vector spaces that live inside bigger ones. These are called subspaces. Subspaces are central in linear algebra because they reveal the internal structure of matrices and linear systems. Two special subspaces-the column space and the null space-play particularly important roles.

What Is a Subspace?

A subspace W of a vector space V is a subset of V that is itself a vector space under the same operations. To qualify as a subspace, W must satisfy:

1. The zero vector 0 is in W.
2. If $u, v \in W$, then $u + v \in W$ (closed under addition).
3. If $u \in W$ and c is a scalar, then $cu \in W$ (closed under scalar multiplication).

That's it-no further checking of all ten vector space axioms is needed, because those are inherited from V.

Simple Examples of Subspaces

- In $\mathbb{R}^3$:
  - A line through the origin is a 1-dimensional subspace.
  - A plane through the origin is a 2-dimensional subspace.
  - The whole space itself is a subspace.
  - The trivial subspace $\{0\}$ contains only the zero vector.
- In the space of polynomials:
  - All polynomials of degree 3 form a subspace.
  - All polynomials with zero constant term form a subspace.
- In function spaces:
  - All continuous functions on $[0, 1]$ form a subspace of all functions on $[0, 1]$.
  - All solutions to a linear differential equation form a subspace.

The Column Space of a Matrix

Given a matrix A, the column space is the set of all linear combinations of its columns. Formally,

$$C(A) = \{A\mathbf{x} : \mathbf{x} \in \mathbb{R}^n\}.$$

- The column space lives inside $\mathbb{R}^m$ if A is $m \times n$.
- It represents all possible outputs of the linear transformation defined by A.
- Its dimension is equal to the rank of A.

Example:

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

The second column is just twice the first. So the column space is all multiples of $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, which is a line in $\mathbb{R}^3$. Rank = 1.

The Null Space of a Matrix

The null space (or kernel) of a matrix A is the set of all vectors $\mathbf{x}$ such that

$$A\mathbf{x} = 0.$$

- It lives in $\mathbb{R}^n$ if A is $m \times n$.
- It represents the "invisible" directions that collapse to zero under the transformation.
- Its dimension is the nullity of A.

Example:

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

Solve $A\mathbf{x} = 0$. This yields a null space spanned by one vector, meaning it is a line through the origin in $\mathbb{R}^3$.

Column Space vs. Null Space

- Column space: describes outputs (y-values that can be reached).
- Null space: describes hidden inputs (directions that vanish).

Together, they capture the full behavior of a matrix.

Geometric Interpretation

- In $\mathbb{R}^3$, the column space could be a plane or a line inside 3D space.
- The null space is orthogonal (in a precise sense) to the row space, which we'll study later.
- Understanding both spaces gives a complete picture of how the matrix transforms vectors.

Why It Matters

1. Subspaces are the natural habitat of linear algebra: almost everything happens inside them.
2. The column space explains what systems $Ax = b$ are solvable.
3. The null space explains why some systems have multiple solutions (free variables).
4. These ideas extend to advanced topics like eigenvectors, SVD, and differential equations.

Try It Yourself

1. Show that the set $\{(x, y, 0) : x, y \in \mathbb{R}\}$ is a subspace of $\mathbb{R}^3$.
2. For

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 1 & 2 & 3 \end{bmatrix},$$

   find the column space and its dimension.
3. For the same A, compute the null space and its dimension.
4. Challenge: Prove that the null space of A is always a subspace of $\mathbb{R}^n$.

Subspaces-especially the column space and null space-are the first glimpse of the hidden geometry inside every matrix, showing us which directions survive and which vanish.

33. Span and Generating Sets

The idea of a span captures the simplest and most powerful way to build new vectors from old ones: by taking linear combinations. A span is not just a set of scattered points but a structured, complete collection of all combinations of a given set of vectors. Understanding span leads directly to the concepts of bases, dimension, and the structure of subspaces.

Definition of Span

Given vectors $v_1, v_2, \ldots, v_k \in V$, the span of these vectors is

$$\text{span}\{v_1, v_2, \ldots, v_k\} = \{a_1 v_1 + a_2 v_2 + \cdots + a_k v_k : a_i \in \mathbb{R}\}.$$

- A span is the set of all possible linear combinations of the vectors.
- It is always a subspace.

- The given vectors are called a generating set.

Simple Examples

1. In $\mathbb{R}^2$:
    - Span of $(1, 0)$ = all multiples of the x-axis (a line).
    - Span of $(1, 0)$ and $(0, 1)$ = the entire plane $\mathbb{R}^2$.
    - Span of $(1, 0)$ and $(2, 0)$ = still the x-axis, since the second vector is redundant.
2. In $\mathbb{R}^3$:
    - Span of a single vector = a line.
    - Span of two independent vectors = a plane through the origin.
    - Span of three independent vectors = the whole space $\mathbb{R}^3$.

Span as Coverage

- If you think of vectors as "directions," the span is everything you can reach by walking in those directions, with any step lengths (scalars) allowed.
- If you only have one direction, you can walk back and forth on a line.
- With two independent directions, you can sweep out a plane.
- With three independent directions in 3D, you can move anywhere.

Generating Sets

A set of vectors is a generating set (or spanning set) for a subspace if their span equals that subspace.

- Example: $\{(1, 0), (0, 1)\}$ generates $\mathbb{R}^2$.
- Example: $\{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$ generates $\mathbb{R}^3$.
- Example: The columns of a matrix generate its column space.

Different generating sets can span the same space. Some may be redundant, others minimal. Later, the concept of a basis refines this idea.

Redundancy in Spanning Sets

- If one vector is a linear combination of others, it does not enlarge the span.
- Example: In $\mathbb{R}^2$, $\{(1, 0), (0, 1), (1, 1)\}$ spans the same space as $\{(1, 0), (0, 1)\}$.
- Eliminating redundancy leads to a more efficient generating set.

Span and Linear Systems

Consider the system $Ax = b$.

- The question "Is there a solution?" is equivalent to "Is b in the span of the columns of A?"
- Thus, span provides the geometric language for solvability.

Why It Matters

1. Span is the foundation for defining subspaces generated by vectors.
2. It connects directly to solvability of linear equations.
3. It introduces the notion of redundancy, preparing for bases and independence.
4. It generalizes naturally to function spaces and abstract vector spaces.

Try It Yourself

1. Find the span of $\{(1, 2), (2, 4)\}$ in $\mathbb{R}^2$.
2. Show that the vectors $(1, 0, 1), (0, 1, 1), (1, 1, 2)$ span only a plane in $\mathbb{R}^3$.
3. Decide whether $(1, 2, 3)$ is in the span of $(1, 0, 1)$ and $(0, 1, 2)$.
4. Challenge: Prove that the set of all polynomials $\{1, x, x^2, \dots\}$ spans the space of all polynomials.

The concept of span transforms our perspective: instead of focusing on single vectors, we see the entire landscape of possibilities they generate.

34. Linear Independence and Dependence

Having introduced span and generating sets, the natural question arises: *when are the vectors in a spanning set truly necessary, and when are some redundant?* This leads to the idea of linear independence. It is the precise way to distinguish between essential vectors (those that add new directions) and dependent vectors (those that can be expressed in terms of others).

Definition of Linear Independence

A set of vectors $\{v_1, v_2, \dots, v_k\}$ is linearly independent if the only solution to

$$a_1 v_1 + a_2 v_2 + \cdots + a_k v_k = 0$$

is

$$a_1 = a_2 = \cdots = a_k = 0.$$

If there exists a nontrivial solution (some $a_i \neq 0$), then the vectors are linearly dependent.

Intuition

- Independent vectors point in genuinely different directions.
- Dependent vectors overlap: at least one can be built from the others.
- In terms of span: removing a dependent vector does not shrink the span, because it adds no new direction.

Simple Examples in $\mathbb{R}^2$

1. $(1, 0)$ and $(0, 1)$ are independent.
    - Equation $a(1, 0) + b(0, 1) = (0, 0)$ forces $a = b = 0$.
2. $(1, 0)$ and $(2, 0)$ are dependent.
    - Equation $2(1, 0) - (2, 0) = (0, 0)$ shows dependence.
3. Any set of 3 vectors in $\mathbb{R}^2$ is dependent, since the dimension of the space is 2.

Examples in $\mathbb{R}^3$

1. $(1, 0, 0), (0, 1, 0), (0, 0, 1)$ are independent.
2. $(1, 2, 3), (2, 4, 6)$ are dependent, since the second is just $2\times$ the first.
3. $(1, 0, 1), (0, 1, 1), (1, 1, 2)$ are dependent: the third is the sum of the first two.

Detecting Independence with Matrices

Put the vectors as columns in a matrix. Perform row reduction:

- If every column has a pivot $\rightarrow$ the set is independent.
- If some column is free $\rightarrow$ the set is dependent.

Example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 0 \end{bmatrix}.$$

Here the third column has no pivot $\rightarrow$ the 3rd vector is dependent on the first two.

Relationship with Dimension

- In $\mathbb{R}^n$, at most n independent vectors exist.
- If you have more than n, dependence is guaranteed.
- A basis of a vector space is simply a maximal independent set that spans the space.

Geometric Interpretation

- Independent vectors = different directions.
- Dependent vectors = one vector lies in the span of others.
- In 2D: two independent vectors span the plane.
- In 3D: three independent vectors span the space.

Why It Matters

1. Independence ensures a generating set is minimal and efficient.
2. It determines whether a system of vectors is a basis.
3. It connects directly to rank: rank = number of independent columns (or rows).
4. It is crucial in geometry, data compression, and machine learning-where redundancy must be identified and removed.

Try It Yourself

1. Test whether $(1, 2)$ and $(2, 4)$ are independent.
2. Are the vectors $(1, 0, 0), (0, 1, 0), (1, 1, 0)$ independent in $\mathbb{R}^3$?
3. Place the vectors $(1, 0, 1), (0, 1, 1), (1, 1, 2)$ into a matrix and row-reduce to check independence.
4. Challenge: Prove that any set of $n + 1$ vectors in $\mathbb{R}^n$ is linearly dependent.

Linear independence is the tool that separates essential directions from redundant ones. It is the key to defining bases, counting dimensions, and understanding the structure of all vector spaces.

35. Basis and Coordinates

The concepts of span and linear independence come together in the powerful idea of a basis. A basis gives us the minimal set of building blocks needed to generate an entire vector space, with no redundancy. Once a basis is chosen, every vector in the space can be described uniquely by a list of numbers called its coordinates.

What Is a Basis?

A basis of a vector space V is a set of vectors $\{v_1, v_2, \dots, v_k\}$ that satisfies two properties:

1. Spanning property: $\text{span}\{v_1, \dots, v_k\} = V$.
2. Independence property: The vectors are linearly independent.

In short: a basis is a spanning set with no redundancy.

Example: Standard Bases

1. In $\mathbb{R}^2$, the standard basis is $\{(1,0),(0,1)\}$.
2. In $\mathbb{R}^3$, the standard basis is $\{(1,0,0),(0,1,0),(0,0,1)\}$.
3. In $\mathbb{R}^n$, the standard basis is the collection of unit vectors, each with a 1 in one position and 0 elsewhere.

These are called standard because they are the default way of describing coordinates.

Uniqueness of Coordinates

One of the most important facts about bases is that they provide unique representations of vectors.

- Given a basis $\{v_1, \dots, v_k\}$, any vector $x \in V$ can be written uniquely as:

$$x = a_1 v_1 + a_2 v_2 + \cdots + a_k v_k.$$

- The coefficients $(a_1, a_2, \dots, a_k)$ are the coordinates of x relative to that basis.

This uniqueness distinguishes bases from arbitrary spanning sets, where redundancy allows multiple representations.

Example in $\mathbb{R}^2$

Let basis $= \{(1,0),(0,1)\}$.

- Vector $(3,5) = 3(1,0) + 5(0,1)$.
- Coordinates relative to this basis: $(3,5)$.

If we switch to a different basis, the coordinates change even though the vector itself does not.

Example with Non-Standard Basis

Basis $= \{(1,1),(1,-1)\}$ in $\mathbb{R}^2$. Find coordinates of $x = (2,0)$.

Solve $a(1,1) + b(1,-1) = (2,0)$. This gives system:

$$a + b = 2, \quad a - b = 0.$$

So $a = 1, b = 1$. Coordinates relative to this basis: $(1,1)$.

Notice: coordinates depend on basis choice.

Basis of Function Spaces

1. For polynomials of degree 2: basis $= \{1, x, x^2\}$.
    - Example: $2 + 3x + 5x^2$ has coordinates $(2,3,5)$.
2. For continuous functions on $[0,1]$, one possible basis is the infinite set $\{1, x, x^2, \dots\}$.

This shows bases are not restricted to geometric vectors.

Dimension

The number of vectors in a basis is the dimension of the vector space.

- $\mathbb{R}^2$ has dimension 2.
- $\mathbb{R}^3$ has dimension 3.
- The space of polynomials of degree 3 has dimension 4.

Dimension tells us how many independent directions exist in the space.

Change of Basis

- Switching from one basis to another is like translating between languages.
- The same vector looks different depending on which "dictionary" (basis) you use.
- Change-of-basis matrices allow systematic translation between coordinate systems.

Geometric Interpretation

- A basis is like setting up coordinate axes in a space.
- In 2D, two independent vectors define a grid.
- In 3D, three independent vectors define a full coordinate system.
- Different bases = different grids overlaying the same space.

Why It Matters

1. Bases provide the simplest possible description of a vector space.
2. They allow us to assign unique coordinates to vectors.
3. They connect the abstract structure of a space with concrete numerical representations.
4. The concept underlies almost all of linear algebra: dimension, transformations, eigenvectors, and more.

Try It Yourself

1. Show that $\{(1,2),(3,4)\}$ is a basis of $\mathbb{R}^2$.
2. Express $(4,5)$ in terms of basis $\{(1,1),(1,-1)\}$.
3. Prove that no basis of $\mathbb{R}^3$ can have more than 3 vectors.
4. Challenge: Show that the set $\{1, \cos x, \sin x\}$ is a basis for the space of all linear combinations of $1, \cos x, \sin x$.

A basis is the minimal, elegant foundation of a vector space, turning the infinite into the manageable by providing a finite set of independent building blocks.

36. Dimension

Dimension is one of the most profound and unifying ideas in linear algebra. It gives a single number that captures the "size" or "capacity" of a vector space: how many independent directions it has. Unlike length, width, or height in everyday geometry, dimension in linear algebra applies to spaces of any kind-geometric, algebraic, or even function spaces.

Definition

The dimension of a vector space V is the number of vectors in any basis of V.

- Since all bases of a vector space have the same number of elements, dimension is well-defined.
- If $\dim V = n$, then:
  - Every set of more than n vectors in V is dependent.
  - Every set of exactly n independent vectors forms a basis.

Examples in Familiar Spaces

1. $\dim(\mathbb{R}^2) = 2$.
   - Basis: $(1, 0), (0, 1)$.
   - Two directions cover the whole plane.
2. $\dim(\mathbb{R}^3) = 3$.
   - Basis: $(1, 0, 0), (0, 1, 0), (0, 0, 1)$.
   - Three independent directions span 3D space.
3. The set of all polynomials of degree 2 has dimension 3.
   - Basis: $\{1, x, x^2\}$.
4. The space of all $m \times n$ matrices has dimension mn.
   - Each entry is independent, and the standard basis consists of matrices with a single 1 and the rest 0.

Finite vs. Infinite Dimensions

- Finite-dimensional spaces: $\mathbb{R}^n$, polynomials of degree k.
- Infinite-dimensional spaces:
  - The space of all polynomials (no degree limit).
  - The space of all continuous functions.
  - These cannot be spanned by a finite set of vectors.

Dimension and Subspaces

- Any subspace of $\mathbb{R}^n$ has dimension n.
- A line through the origin in $\mathbb{R}^3$: dimension 1.
- A plane through the origin in $\mathbb{R}^3$: dimension 2.
- The whole space: dimension 3.
- The trivial subspace $\{0\}$: dimension 0.

Dimension and Systems of Equations

When solving $A\mathbf{x} = \mathbf{b}$:

- The dimension of the column space = rank = number of independent directions in the outputs.
- The dimension of the null space = number of free variables.
- By the rank–nullity theorem:

$$\dim(\text{column space}) + \dim(\text{null space}) = \text{number of variables.}$$

Geometric Meaning

- Dimension counts the minimum number of coordinates needed to describe a vector.
- In $\mathbb{R}^2$, you need 2 numbers.
- In $\mathbb{R}^3$, you need 3 numbers.
- In the polynomial space of degree 3, you need 4 coefficients.

Thus, dimension = length of coordinate list.

Checking Dimension in Practice

1. Place candidate vectors as columns of a matrix.
2. Row reduce to echelon form.
3. Count pivots. That number = dimension of the span of those vectors.

Why It Matters

1. Dimension is the most fundamental measure of a vector space.
2. It tells us how "large" or "complex" the space is.
3. It sets absolute limits: in $\mathbb{R}^n$, no more than n independent vectors exist.
4. It underlies coordinate systems, bases, and transformations.
5. It bridges geometry (lines, planes, volumes) with algebra (solutions, equations, matrices).

Try It Yourself

1. What is the dimension of the span of $(1, 2, 3)$, $(2, 4, 6)$, $(0, 0, 0)$?
2. Find the dimension of the subspace of $\mathbb{R}^3$ defined by $x + y + z = 0$.
3. Prove that the set of all 2×2 symmetric matrices has dimension 3.
4. Challenge: Show that the space of polynomials of degree k has dimension $k + 1$.

Dimension is the measuring stick of linear algebra: it tells us how many independent pieces of information are needed to describe the whole space.

37. Rank–Nullity Theorem

The rank–nullity theorem is one of the central results of linear algebra. It gives a precise balance between two fundamental aspects of a matrix: the dimension of its column space (rank) and the dimension of its null space (nullity). It shows that no matter how complicated a matrix looks, the distribution of information between its "visible" outputs and its "hidden" null directions always obeys a strict law.

Statement of the Theorem

Let A be an $m \times n$ matrix (mapping $\mathbb{R}^n \to \mathbb{R}^m$):

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

where:

- rank(A) = dimension of the column space of A.
- nullity(A) = dimension of the null space of A.
- n = number of columns of A, i.e., the number of variables.

Intuition

Think of a matrix as a machine that transforms input vectors into outputs:

- Rank measures how many independent output directions survive.
- Nullity measures how many input directions get "lost" (mapped to zero).
- The theorem says: total inputs = useful directions (rank) + wasted directions (nullity).

This ensures nothing disappears mysteriously-every input direction is accounted for.

Example 1: Full Rank

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

- Rank = 2 (two independent columns).
- Null space = $\{0\}$, so nullity = 0.
- Rank + nullity = 2 = number of variables.

Example 2: Dependent Columns

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

- Second column is a multiple of the first. Rank = 1.
- Null space contains all vectors (x, y) with $y = -2x$. Nullity = 1.
- Rank + nullity = 1 + 1 = 2 = number of variables.

Example 3: Larger System

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

- Columns: $(1, 0), (0, 1), (1, 1)$.
- Only two independent columns $\rightarrow$ Rank = 2.
- Null space: solve $x + z = 0, y + z = 0 \Rightarrow (x, y, z) = (-t, -t, t)$. Nullity = 1.
- Rank + nullity = 2 + 1 = 3 = number of variables.

Proof Sketch (Conceptual)

1. Row reduce A to echelon form.
2. Pivots correspond to independent columns $\rightarrow$ count = rank.
3. Free variables correspond to null space directions $\rightarrow$ count = nullity.
4. Each column is either a pivot column or corresponds to a free variable, so:

$$\text{rank} + \text{nullity} = \text{number of columns}.$$

Geometric Meaning

- In $\mathbb{R}^3$, if a transformation collapses all vectors onto a plane (rank = 2), then one direction disappears entirely (nullity = 1).
- In $\mathbb{R}^4$, if a matrix has rank 2, then its null space has dimension 2, meaning half the input directions vanish.

The theorem guarantees the geometry of "surviving" and "vanishing" directions always adds up consistently.

Applications

1. Solving systems $Ax = b$:
    - Rank determines consistency and structure of solutions.
    - Nullity tells how many free parameters exist in the solution.
2. Data compression: Rank identifies independent features; nullity shows redundancy.
3. Computer graphics: Rank–nullity explains how 3D coordinates collapse into 2D images: one dimension of depth is lost.
4. Machine learning: Rank signals how much real information a dataset contains; nullity indicates degrees of freedom that add nothing new.

Why It Matters

1. The rank–nullity theorem connects the abstract ideas of rank and nullity into a single, elegant formula.
2. It ensures conservation of dimension: no information magically appears or disappears.
3. It is essential in understanding solutions of systems, dimensions of subspaces, and the structure of linear transformations.
4. It prepares the ground for deeper results in algebra, topology, and differential equations.

Try It Yourself

1. Verify rank–nullity for

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

2. For a 4×5 matrix of rank 3, what is its nullity?
3. In $\mathbb{R}^3$, suppose a matrix maps all of space onto a line. What are its rank and nullity?
4. Challenge: Prove rigorously that the row space and null space are orthogonal complements, and use this to derive rank–nullity again.

The rank–nullity theorem is the law of balance in linear algebra: every input dimension is accounted for, either as a surviving direction (rank) or as one that vanishes (nullity).

38. Coordinates Relative to a Basis

Once a basis for a vector space is chosen, every vector in that space can be described uniquely in terms of the basis. These descriptions are called coordinates. Coordinates transform abstract vectors into concrete lists of numbers, making computation possible. Changing the basis changes the coordinates, but the underlying vector remains the same.

The Core Idea

Given a vector space V and a basis $B = \{v_1, v_2, \dots, v_n\}$, every vector $x \in V$ can be written uniquely as:

$$x = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n.$$

The coefficients $(a_1, a_2, \dots, a_n)$ are the coordinates of x with respect to the basis B.

This representation is unique because basis vectors are independent.

Example in $\mathbb{R}^2$

1. Standard basis: $B = \{(1,0),(0,1)\}$.
   - Vector $x = (3,5)$.
   - Coordinates relative to B: $(3,5)$.
2. Non-standard basis: $B = \{(1,1),(1,-1)\}$.
   - Write $x = (3,5)$ as $a(1,1) + b(1,-1)$.
   - Solve:

     $$a + b = 3, \quad a - b = 5.$$

     Adding: $2a = 8 \implies a = 4$. Subtracting: $2b = -2 \implies b = -1$.
   - Coordinates relative to this basis: $(4,-1)$.

The same vector looks different depending on the chosen basis.

Example in $\mathbb{R}^3$

Let $B = \{(1,0,0),(1,1,0),(1,1,1)\}$. Find coordinates of $x = (2,3,4)$.

Solve $a(1,0,0) + b(1,1,0) + c(1,1,1) = (2,3,4)$. This gives system:

$$a + b + c = 2, \quad b + c = 3, \quad c = 4.$$

From $c = 4$, we get $b + c = 3 \implies b = -1$. Then $a + b + c = 2 \implies a - 1 + 4 = 2 \implies a = -1$. Coordinates: $(-1,-1,4)$.

Matrix Formulation

If $B = \{v_1, \ldots, v_n\}$, form the basis matrix

$$P = [v_1 \ v_2 \ \ldots \ v_n].$$

Then for a vector x, its coordinate vector $[x]_B$ satisfies

$$P[x]_B = x.$$

Thus,

$$[x]_B = P^{-1}x.$$

This shows coordinate transformation is simply matrix multiplication.

Changing Coordinates

Suppose a vector has coordinates $[x]_B$ relative to basis B. If we switch to another basis C, we use a change-of-basis matrix to convert coordinates:

$$[x]_C = (P_C^{-1}P_B)[x]_B.$$

This process is fundamental in computer graphics, robotics, and data transformations.

Geometric Meaning

- A basis defines a coordinate system: axes in the space.
- Coordinates are the "addresses" of vectors relative to those axes.
- Changing basis is like rotating or stretching the grid: the address changes, but the point does not.

Why It Matters

1. Coordinates make abstract vectors computable.
2. They allow us to represent functions, polynomials, and geometric objects numerically.
3. Changing basis simplifies problems-e.g., diagonalization makes matrices easy to analyze.
4. They connect the abstract (spaces, bases) with the concrete (numbers, matrices).

Try It Yourself

1. Express $x = (4, 2)$ relative to basis $\{(1, 1), (1, -1)\}$.
2. Find coordinates of $x = (2, 1, 3)$ relative to basis $\{(1, 0, 1), (0, 1, 1), (1, 1, 0)\}$.
3. If basis B is the standard basis and basis $C = \{(1, 1), (1, -1)\}$, compute the change-of-basis matrix from B to C.
4. Challenge: Show that if P is invertible, its columns form a basis, and explain why this guarantees uniqueness of coordinates.

Coordinates relative to a basis are the bridge between geometry and algebra: they turn abstract spaces into numerical systems where computation, reasoning, and transformation become systematic and precise.

39. Change-of-Basis Matrices

Every vector space allows multiple choices of basis, and each basis provides a different way of describing the same vectors. The process of moving from one basis to another is called a change of basis. To perform this change systematically, we use a change-of-basis matrix. This matrix acts as a translator between coordinate systems: it converts the coordinates of a vector relative to one basis into coordinates relative to another.

Why Change Bases?

1. Simplicity of computation: Some problems are easier in certain bases. For example, diagonalizing a matrix allows us to raise it to powers more easily.
2. Geometry: Different bases can represent rotated or scaled coordinate systems.
3. Applications: In physics, computer graphics, robotics, and data science, changing bases is equivalent to switching perspectives or reference frames.

The Basic Setup

Let V be a vector space with two bases:

- $B = \{b_1, b_2, \dots, b_n\}$
- $C = \{c_1, c_2, \dots, c_n\}$

Suppose a vector $x \in V$ has coordinates $[x]_B$ relative to B, and $[x]_C$ relative to C.

We want a matrix $P_{B \to C}$ such that:

$$[x]_C = P_{B \to C}[x]_B.$$

This matrix $P_{B\to C}$ is the change-of-basis matrix from B to C.

Constructing the Change-of-Basis Matrix

1. Write each vector in the basis B in terms of the basis C.
2. Place these coordinate vectors as the columns of a matrix.
3. The resulting matrix converts coordinates from B to C.

In matrix form:

$$P_{B\to C} = [[b_1]_C\ [b_2]_C\ \ ...\ \ [b_n]_C].$$

Example in $\mathbb{R}^2$

Let

- $B = \{(1,0),(0,1)\}$ (standard basis).
- $C = \{(1,1),(1,-1)\}$.

To build $P_{B\to C}$:

- Express each vector of B in terms of C.

Solve:

$$(1,0) = a(1,1) + b(1,-1).$$

This gives system:

$$a + b = 1, \quad a - b = 0.$$

Solution: $a = \frac{1}{2}, b = \frac{1}{2}$. So $(1,0) = \frac{1}{2}(1,1) + \frac{1}{2}(1,-1)$.

Next:

$$(0,1) = a(1,1) + b(1,-1).$$

System:

$$a + b = 0, \quad a - b = 1.$$

Solution: $a = \frac{1}{2}, b = -\frac{1}{2}$.

Thus:

$$P_{B\to C} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

So for any vector x,

$$[x]_C = P_{B\to C}[x]_B.$$

Inverse Change of Basis

If $P_{B\to C}$ is the change-of-basis matrix from B to C, then its inverse is the change-of-basis matrix in the opposite direction:

$$P_{C\to B} = (P_{B\to C})^{-1}.$$

This makes sense: translating back and forth between languages should undo itself.

General Formula with Basis Matrices

Let

$$P_B = [b_1\ b_2\ \ ...\ \ b_n], \quad P_C = [c_1\ c_2\ \ ...\ \ c_n],$$

the matrices whose columns are basis vectors written in standard coordinates.

Then the change-of-basis matrix from B to C is:

$$P_{B\to C} = P_C^{-1} P_B.$$

This formula is extremely useful because it reduces the problem to matrix multiplication.

Geometric Interpretation

- Changing basis is like rotating or stretching the grid lines of a coordinate system.
- The vector itself (the point in space) does not move. What changes is its description in terms of the new grid.
- The change-of-basis matrix is the tool that translates between these descriptions.

Applications

1. Diagonalization: Expressing a matrix in a basis of its eigenvectors makes it diagonal, simplifying analysis.
2. Computer graphics: Changing camera viewpoints requires change-of-basis matrices.
3. Robotics: Coordinate transformations connect robot arms, joints, and workspace frames.
4. Data science: PCA finds a new basis (principal components) where data is easier to analyze.

Why It Matters

1. Provides a universal method to translate coordinates between bases.
2. Makes abstract transformations concrete and computable.
3. Forms the backbone of diagonalization, Jordan form, and the spectral theorem.
4. Connects algebraic manipulations with geometry and real-world reference frames.

Try It Yourself

1. Compute the change-of-basis matrix from the standard basis to $\{(2,1),(1,1)\}$ in $\mathbb{R}^2$.
2. Find the change-of-basis matrix from basis $\{(1,0,0),(0,1,0),(0,0,1)\}$ to $\{(1,1,0),(0,1,1),(1,0,1)\}$ in $\mathbb{R}^3$.
3. Show that applying $P_{B\to C}$ then $P_{C\to B}$ returns the original coordinates.
4. Challenge: Derive the formula $P_{B\to C} = P_C^{-1} P_B$ starting from the definition of coordinates.

Change-of-basis matrices give us the precise mechanism for switching perspectives. They ensure that although bases change, vectors remain invariant, and computations remain consistent.

40. Affine Subspaces

So far, vector spaces and subspaces have always passed through the origin. But in many real-world situations, we deal with shifted versions of these spaces: planes not passing through the origin, lines offset from the zero vector, or solution sets to linear equations with nonzero constants. These structures are called affine subspaces. They extend the idea of subspaces by allowing "translation away from the origin."

Definition

An affine subspace of a vector space V is a set of the form

$$x_0 + W = \{x_0 + w : w \in W\},$$

where:

- $x_0 \in V$ is a fixed vector (the "base point" or "anchor"),
- $W \subseteq V$ is a linear subspace.

Thus, an affine subspace is simply a subspace shifted by a vector.

Examples in $\mathbb{R}^2$

1. A line through the origin: $\text{span}\{(1, 2)\}$. This is a subspace.
2. A line not through the origin: $(3, 1) + \text{span}\{(1, 2)\}$. This is an affine subspace.
3. The entire plane: $\mathbb{R}^2$, which is both a subspace and an affine subspace.

Examples in $\mathbb{R}^3$

1. Plane through the origin: $\text{span}\{(1, 0, 0), (0, 1, 0)\}$.
2. Plane not through the origin: $(2, 3, 4) + \text{span}\{(1, 0, 0), (0, 1, 0)\}$.
3. Line parallel to the z-axis but passing through $(1, 1, 5)$: $(1, 1, 5) + \text{span}\{(0, 0, 1)\}$.

Relation to Linear Systems

Affine subspaces naturally arise as solution sets of linear equations.

1. Homogeneous system: $Ax = 0$.
    - Solution set is a subspace (the null space).
2. Non-homogeneous system: $Ax = b$ with $b \neq 0$.
    - Solution set is affine.
    - If x_p is one particular solution, then the general solution is:

      $$x = x_p + N(A),$$

      where $N(A)$ is the null space.

Thus, the geometry of solving equations leads naturally to affine subspaces.

Affine Dimension

The dimension of an affine subspace is defined as the dimension of its direction subspace W.

- A point: affine subspace of dimension 0.
- A line: dimension 1.
- A plane: dimension 2.
- Higher analogs continue in $\mathbb{R}^n$.

Difference Between Subspaces and Affine Subspaces

- Subspaces always contain the origin.
- Affine subspaces may or may not pass through the origin.
- Every subspace is an affine subspace (with base point $x_0 = 0$).

Geometric Intuition

Think of affine subspaces as "flat sheets" floating in space:

- A line through the origin is a rope tied at the center.
- A line parallel to it but offset is the same rope moved to the side.
- Affine subspaces preserve shape and direction, but not position.

Applications

1. Linear equations: General solutions are affine subspaces.
2. Optimization: Feasible regions in linear programming are affine subspaces (intersected with inequalities).
3. Computer graphics: Affine transformations map affine subspaces to affine subspaces, preserving straightness and parallelism.
4. Machine learning: Affine decision boundaries (like hyperplanes) separate data into classes.

Why It Matters

1. Affine subspaces generalize subspaces, making linear algebra more flexible.
2. They allow us to describe solution sets that don't include the origin.
3. They provide the geometric foundation for affine geometry, computer graphics, and optimization.
4. They serve as the bridge from pure linear algebra to applied modeling.

Try It Yourself

1. Show that the set of solutions to

$$x + y + z = 1$$

   is an affine subspace of $\mathbb{R}^3$. Identify its dimension.

2. Find the general solution to

$$x + 2y = 3$$

   and describe it as an affine subspace.

3. Prove that the intersection of two affine subspaces is either empty or another affine subspace.

4. Challenge: Show that every affine subspace can be written uniquely as $x_0 + W$ with W a subspace.

Affine subspaces are the natural setting for most real-world linear problems: they combine the strict structure of subspaces with the freedom of translation, capturing both direction and position.

Closing

```
Each basis a song,
dimension counts melodies,
the space breathes its form.
```

Chapter 5. Linear Transformation and Structure

Opening

```
Maps preserve the line,
reflections ripple outward,
motion kept in frame.
```

41. Linear Transformations

A linear transformation is the heart of linear algebra. It is the rule that connects two vector spaces in a way that respects their linear structure: addition and scalar multiplication. Instead of thinking of vectors as static objects, linear transformations let us study how vectors move, stretch, rotate, project, or reflect. They give linear algebra its dynamic power and are the bridge between abstract theory and concrete applications.

Definition

A function $T : V \to W$ between vector spaces is called a linear transformation if for all $u, v \in V$ and scalars $a, b \in \mathbb{R}$ (or another field),

$$T(au + bv) = aT(u) + bT(v).$$

This single condition encodes two rules:

1. Additivity: $T(u + v) = T(u) + T(v)$.
2. Homogeneity: $T(av) = aT(v)$.

If both are satisfied, the transformation is linear.

Examples of Linear Transformations

1. Scaling: $T(x) = 3x$ in $\mathbb{R}$. Every number is stretched threefold.
2. Rotation in the plane: $T(x, y) = (x\cos\theta - y\sin\theta,\ x\sin\theta + y\cos\theta)$.
3. Projection: Projecting (x, y, z) onto the xy-plane: $T(x, y, z) = (x, y, 0)$.
4. Differentiation: On the space of polynomials, $T(p(x)) = p'(x)$.
5. Integration: On continuous functions, $T(f)(x) = \int_0^x f(t)\,dt$.

All these are linear because they preserve addition and scaling.

Non-Examples

- $T(x) = x^2$ is not linear, because $(x+y)^2 \neq x^2 + y^2$.
- $T(x, y) = (x + 1, y)$ is not linear, because it fails homogeneity: scaling doesn't preserve the "+1."

Nonlinear rules break the structure of vector spaces.

Matrix Representation

Every linear transformation from $\mathbb{R}^n$ to $\mathbb{R}^m$ can be represented by a matrix.

If $T : \mathbb{R}^n \to \mathbb{R}^m$, then there exists an $m \times n$ matrix A such that:

$$T(x) = Ax.$$

The columns of A are simply $T(e_1), T(e_2), \ldots, T(e_n)$, where e_i are the standard basis vectors.

Example: Let $T(x, y) = (2x + y, x - y)$.

- $T(e_1) = T(1, 0) = (2, 1)$.
- $T(e_2) = T(0, 1) = (1, -1)$. So

$$A = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}.$$

Then $T(x, y) = A \begin{bmatrix} x \\ y \end{bmatrix}$.

Properties of Linear Transformations

1. The image of the zero vector is always zero: $T(0) = 0$.
2. The image of a line through the origin is again a line (or collapsed to a point).
3. Composition of linear transformations is linear.
4. Every linear transformation preserves the structure of subspaces.

Kernel and Image (Preview)

For $T : V \to W$:

- The kernel (or null space) is all vectors mapped to zero: $\ker T = \{v \in V : T(v) = 0\}$.
- The image (or range) is all outputs that can be achieved: $\text{im}(T) = \{T(v) : v \in V\}$. The rank–nullity theorem applies here:

$$\dim(\ker T) + \dim(\text{im}(T)) = \dim(V).$$

Geometric Interpretation

Linear transformations reshape space:

- Scaling stretches space uniformly in one direction.
- Rotation spins space while preserving lengths.
- Projection flattens space onto lower dimensions.
- Reflection flips space across a line or plane.

The key feature: straight lines remain straight, and the origin stays fixed.

Applications

1. Computer graphics: Scaling, rotating, projecting 3D objects onto 2D screens.
2. Robotics: Transformations between joint coordinates and workspace positions.
3. Data science: Linear mappings represent dimensionality reduction and feature extraction.
4. Differential equations: Solutions often involve linear operators acting on function spaces.
5. Machine learning: Weight matrices in neural networks are stacked linear transformations, interspersed with nonlinearities.

Why It Matters

1. Linear transformations generalize matrices to any vector space.
2. They unify geometry, algebra, and applications under one concept.
3. They provide the natural framework for studying eigenvalues, eigenvectors, and decompositions.
4. They model countless real-world processes: physical, computational, and abstract.

Try It Yourself

1. Prove that $T(x, y, z) = (x + 2y, z, x - y + z)$ is linear.
2. Find the matrix representation of the transformation that reflects vectors in $\mathbb{R}^2$ across the line $y = x$.
3. Show why $T(x, y) = (x^2, y)$ is not linear.
4. Challenge: For the differentiation operator $D : P_3 \to P_2$ on polynomials of degree 3, find its matrix relative to the basis $\{1, x, x^2, x^3\}$ in the domain and $\{1, x, x^2\}$ in the codomain.

Linear transformations are the language of linear algebra. They capture the essence of symmetry, motion, and structure in spaces of any kind, making them indispensable for both theory and practice.

42. Matrix Representation of a Linear Map

Every linear transformation can be expressed concretely as a matrix. This is one of the most powerful bridges in mathematics: it translates abstract functional rules into arrays of numbers that can be calculated, manipulated, and visualized.

From Abstract Rule to Concrete Numbers

Suppose $T : V \to W$ is a linear transformation between two finite-dimensional vector spaces. To represent T as a matrix, we first select bases:

- $B = \{v_1, v_2, \dots, v_n\}$ for the domain V.
- $C = \{w_1, w_2, \dots, w_m\}$ for the codomain W.

For each basis vector v_j, compute $T(v_j)$. Each image $T(v_j)$ is a vector in W, so it can be written as a combination of the basis C:

$$T(v_j) = a_{1j}w_1 + a_{2j}w_2 + \cdots + a_{mj}w_m.$$

The coefficients $(a_{1j}, a_{2j}, \dots, a_{mj})$ become the j-th column of the matrix representing T.

Thus, the matrix of T relative to bases B and C is

$$[T]_{B\to C} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}.$$

This guarantees that for any vector x in coordinates relative to B,

$$[T(x)]_C = [T]_{B\to C}[x]_B.$$

Standard Basis Case

When both B and C are the standard bases, the process simplifies:

- Take $T(e_1), T(e_2), \ldots, T(e_n)$.
- Place them as columns in a matrix.

That matrix directly represents T.

Example: Let $T(x, y) = (2x + y, x - y)$.

- $T(e_1) = (2, 1)$.
- $T(e_2) = (1, -1)$.

So the standard matrix is

$$A = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}.$$

For any vector $\begin{bmatrix} x \\ y \end{bmatrix}$,

$$T(x, y) = A \begin{bmatrix} x \\ y \end{bmatrix}.$$

Multiple Perspectives

- Columns-as-images: Each column shows where a basis vector goes.
- Row view: Each row encodes how to compute one coordinate of the output.
- Operator view: The matrix acts like a machine: input vector $\to$ multiply $\to$ output vector.

Geometric Insight

Matrices reshape space. In $\mathbb{R}^2$:

- The first column shows where the x-axis goes.
- The second column shows where the y-axis goes. The entire grid is determined by these two images.

In $\mathbb{R}^3$, the three columns are the images of the unit coordinate directions, defining how the whole space twists, rotates, or compresses.

Applications

1. Computer graphics: Rotations, scaling, and projections are represented by small matrices.
2. Robotics: Coordinate changes between joints and workspaces rely on transformation matrices.
3. Data science: Linear maps such as PCA are implemented with matrices that project data into lower dimensions.
4. Physics: Linear operators like rotations, boosts, and stress tensors are matrix representations.

Why It Matters

1. Matrices are computational tools: we can add, multiply, invert them.
2. They let us use algorithms like Gaussian elimination, LU/QR/SVD to study transformations.
3. They link abstract vector space theory to hands-on numerical calculation.
4. They reveal the structure of transformations at a glance, just by inspecting columns and rows.

Try It Yourself

1. Find the matrix for the transformation $T(x, y, z) = (x + 2y, y + z, x + z)$ in the standard basis.
2. Compute the matrix of $T : \mathbb{R}^2 \to \mathbb{R}^2$, where $T(x, y) = (x - y, x + y)$.
3. Using the basis $B = \{(1, 1), (1, -1)\}$ for $\mathbb{R}^2$, find the matrix of $T(x, y) = (2x, y)$ relative to B.
4. Challenge: Show that matrix multiplication corresponds to composition of transformations, i.e. $[S \circ T] = [S][T]$.

Matrix representations are the practical form of linear transformations, turning elegant definitions into something we can compute, visualize, and apply across science and engineering.

43. Kernel and Image

Every linear transformation hides two essential structures: the set of vectors that collapse to zero, and the set of all possible outputs. These are called the kernel and the image. They are the DNA of a linear map, revealing its internal structure, its strengths, and its limitations.

The Kernel

The kernel (or null space) of a linear transformation $T : V \to W$ is defined as:

$$\ker(T) = \{v \in V : T(v) = 0\}.$$

- It is the set of all vectors that the transformation sends to the zero vector.
- It measures how much information is "lost" under the transformation.
- The kernel is always a subspace of the domain V.

Examples:

1. For $T : \mathbb{R}^2 \to \mathbb{R}^2$, $T(x, y) = (x, 0)$.
    - Kernel: all vectors of the form $(0, y)$. This is the y-axis.
2. For $T : \mathbb{R}^3 \to \mathbb{R}^2$, $T(x, y, z) = (x, y)$.
    - Kernel: all vectors of the form $(0, 0, z)$. This is the z-axis.

The kernel tells us which directions in the domain vanish under T.

The Image

The image (or range) of a linear transformation is defined as:

$$\text{im}(T) = \{T(v) : v \in V\}.$$

- It is the set of all vectors that can actually be reached by applying T.
- It describes the "output space" of the transformation.
- The image is always a subspace of the codomain W.

Examples:

1. For $T(x, y) = (x, 0)$:
    - Image: all vectors of the form $(a, 0)$. This is the x-axis.

2. For $T(x, y, z) = (x + y, y + z)$:

    - Image: all of $\mathbb{R}^2$. Any vector (u, v) can be achieved by solving equations for (x, y, z).

Kernel and Image Together

These two subspaces reflect two aspects of T:

- The kernel measures the collapse in dimension.
- The image measures the preserved and transmitted directions.

A central result is the Rank–Nullity Theorem:

$$\dim(\ker T) + \dim(\text{im } T) = \dim(V).$$

- $\dim(\ker T)$ is the nullity.
- $\dim(\text{im } T)$ is the rank.

This theorem guarantees a perfect balance: the domain splits into lost directions (kernel) and active directions (image).

Matrix View

For a matrix A, the linear map is $T(x) = Ax$.

- The kernel is the solution set of $Ax = 0$.
- The image is the column space of A.

Example:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 1 \end{bmatrix}.$$

- Image: span of the columns

$$\text{im}(A) = \text{span}\{(1, 0), (2, 1), (3, 1)\}.$$

- Kernel: solve

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

This leads to solutions like $x = -y - 2z$. So the kernel is 1-dimensional, the image is 2-dimensional, and the domain (3D) splits as $1 + 2 = 3$.

Geometric Intuition

- The kernel is the set of invisible directions, like shadows disappearing in projection.
- The image is the set of all shadows that can appear.
- Together they describe projection, flattening, stretching, or collapsing.

Example: Projecting $\mathbb{R}^3$ onto the xy-plane:

- Kernel: the z-axis (all points collapsed to zero height).
- Image: the entire xy-plane (all possible shadows).

Applications

1. Solving equations: Kernel describes all solutions to $Ax = 0$. Image describes what right-hand sides b make $Ax = b$ solvable.
2. Data science: Nullity corresponds to redundant features; rank corresponds to useful independent features.
3. Physics: In mechanics, symmetries often form the kernel of a transformation, while observable quantities form the image.
4. Control theory: The kernel and image determine controllability and observability of systems.

Why It Matters

1. Kernel and image classify transformations into invertible or not.
2. They give a precise language to describe dimension changes.
3. They are the foundation of rank, nullity, and invertibility.
4. They generalize far beyond matrices: to polynomials, functions, operators, and differential equations.

Try It Yourself

1. Compute the kernel and image of $T(x, y, z) = (x + y, y + z)$.
2. For the projection $T(x, y, z) = (x, y, 0)$, identify kernel and image.
3. Show that if the kernel is trivial ($\{0\}$), then the transformation is injective.
4. Challenge: Prove the rank–nullity theorem for a 3×3 matrix by working through examples.

The kernel and image are the twin lenses through which linear transformations are understood. One tells us what disappears, the other what remains. Together, they give the clearest picture of a transformation's essence.

44. Invertibility and Isomorphisms

Linear transformations come in many forms: some collapse space into lower dimensions, others stretch it, and a special group preserves all information perfectly. These special transformations are invertible, meaning they can be reversed exactly. When two vector spaces are related by such a transformation, we say they are isomorphic-structurally identical, even if they look different on the surface.

Invertibility of Linear Transformations

A linear transformation $T : V \to W$ is invertible if there exists another linear transformation $S : W \to V$ such that:

$$S \circ T = I_V \quad \text{and} \quad T \circ S = I_W,$$

where I_V and I_W are identity maps on V and W.

- S is called the inverse of T.
- If such an inverse exists, T is a bijection: both one-to-one (injective) and onto (surjective).
- In finite-dimensional spaces, this is equivalent to saying that T is represented by an invertible matrix.

Invertible Matrices

An $n \times n$ matrix A is invertible if there exists another $n \times n$ matrix A^{-1} such that:

$$AA^{-1} = A^{-1}A = I.$$

Characterizations of Invertibility:

1. A is invertible $\det(A) \neq 0$.
2. Columns of A are linearly independent.
3. Columns of A span $\mathbb{R}^n$.
4. Rank of A is n.
5. The system $Ax = b$ has exactly one solution for every b.

All these properties tie together: invertibility means no information is lost when transforming vectors.

Isomorphisms of Vector Spaces

Two vector spaces V and W are isomorphic if there exists a bijective linear transformation $T : V \to W$.

- This means V and W are "the same" in structure, though they may look different.
- For finite-dimensional spaces:

$$V \cong W \quad \text{if and only if} \quad \dim(V) = \dim(W).$$

- Example: $\mathbb{R}^2$ and the set of all polynomials of degree 1 are isomorphic, because both have dimension 2.

Examples of Invertibility

1. Rotation in the plane: Every rotation matrix has an inverse (rotation by the opposite angle).

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad R(\theta)^{-1} = R(-\theta).$$

2. Scaling by nonzero factor: $T(x) = ax$ with $a \neq 0$. Inverse is $T^{-1}(x) = \frac{1}{a}x$.
3. Projection onto a line: Not invertible, because depth is lost. The kernel is nontrivial.
4. Differentiation on polynomials of degree n: Not invertible, since constant terms vanish in the kernel.
5. Differentiation on exponential functions: Invertible: the inverse is integration (up to constants).

Geometric Interpretation

- Invertible transformations preserve dimension: no flattening or collapsing occurs.
- They may rotate, shear, stretch, or reflect, but every input vector can be uniquely recovered.
- The determinant tells the "volume scaling" of the transformation: invertibility requires this volume not to collapse to zero.

Applications

1. Computer graphics: Invertible matrices allow smooth transformations where no information is lost. Non-invertible maps (like projections) create 2D renderings from 3D worlds.
2. Cryptography: Encryption systems rely on invertible linear maps for encoding/decoding.
3. Robotics: Transformations between joint and workspace coordinates must often be invertible for precise control.
4. Data science: PCA often reduces dimension (non-invertible), but whitening transformations are invertible within the chosen subspace.
5. Physics: Coordinate changes (e.g., Galilean or Lorentz transformations) are invertible, ensuring that physical laws remain consistent.

Why It Matters

1. Invertible maps preserve the entire structure of a vector space.
2. They classify vector spaces: if two have the same dimension, they are fundamentally the same via isomorphism.
3. They allow reversible modeling, essential in physics, cryptography, and computation.
4. They highlight the delicate balance between lossless transformations (invertible) and lossy ones (non-invertible).

Try It Yourself

1. Prove that the matrix $\begin{bmatrix} 2 & 1 \\ 3 & 2 \end{bmatrix}$ is invertible by computing its determinant and its inverse.
2. Show that projection onto the x-axis in $\mathbb{R}^2$ is not invertible. Identify its kernel.
3. Construct an explicit isomorphism between $\mathbb{R}^3$ and the space of polynomials of degree 2.
4. Challenge: Prove that if T is an isomorphism, then it maps bases to bases.

Invertibility and isomorphism are the gateways from "linear rules" to the grand idea of equivalence. They allow us to say, with mathematical precision, when two spaces are truly the same in structure-different clothes, same skeleton.

45. Composition, Powers, and Iteration

Linear transformations are not isolated operations-they can be combined, repeated, and layered to build more complex effects. This leads us to the ideas of composition, powers of transformations, and iteration. These concepts form the backbone of linear dynamics, algorithms, and many real-world systems where repeated actions accumulate into surprising results.

Composition of Linear Transformations

If $T : U \to V$ and $S : V \to W$ are linear transformations, then their composition is another transformation

$$S \circ T : U \to W, \quad (S \circ T)(u) = S(T(u)).$$

- Composition is associative: $(R \circ S) \circ T = R \circ (S \circ T)$.
- Composition is linear: the result of composing two linear maps is still linear.
- In terms of matrices, if $T(x) = Ax$ and $S(x) = Bx$, then

$$(S \circ T)(x) = B(Ax) = (BA)x.$$

  Notice that the order matters: composition corresponds to matrix multiplication.

Example:

1. $T(x, y) = (x + 2y, y)$.
2. $S(x, y) = (2x, x - y)$. Then $(S \circ T)(x, y) = S(x + 2y, y) = (2(x + 2y), (x + 2y) - y) = (2x + 4y, x + y)$. Matrix multiplication confirms the same result.

Powers of Transformations

If $T : V \to V$, we can apply it repeatedly:

$$T^2 = T \circ T, \quad T^3 = T \circ T \circ T, \quad ...$$

- These are called powers of T.
- If $T(x) = Ax$, then $T^k(x) = A^k x$.
- Powers of transformations capture repeated processes, like compounding interest, population growth, or iterative algorithms.

Example: Let $T(x, y) = (2x, 3y)$. Then

$$T^n(x, y) = (2^n x, 3^n y).$$

Each iteration amplifies the scaling along different directions.

Iteration and Dynamical Systems

Iteration means applying the same transformation repeatedly to study long-term behavior:

$$x_{k+1} = T(x_k), \quad x_0 \text{ given.}$$

- This creates a discrete dynamical system.
- Depending on T, vectors may grow, shrink, oscillate, or stabilize.

Example 1 (Markov Chains): If T is a stochastic matrix, iteration describes probability evolution over time. Eventually, the system may converge to a steady-state distribution.

Example 2 (Population Models): If T describes how sub-populations interact, iteration simulates generations. Eigenvalues dictate whether populations explode, stabilize, or vanish.

Example 3 (Computer Graphics): Repeated affine transformations create fractals like the Sierpinski triangle.

Stability and Eigenvalues

The behavior of $T^n(x)$ depends heavily on eigenvalues of the transformation.

- If $|\lambda| < 1$, repeated application shrinks vectors in that direction to zero.
- If $|\lambda| > 1$, repeated application causes exponential growth.
- If $|\lambda| = 1$, vectors rotate or oscillate without changing length.

This link between powers and eigenvalues underpins many algorithms in numerical analysis and physics.

Geometric Interpretation

- Composition = chaining geometric actions (rotate then reflect, scale then shear).
- Powers = applying the same action repeatedly (rotating 90° four times = identity).
- Iteration = exploring the "orbit" of a vector under repeated transformations.

Applications

1. Search engines: PageRank is computed by iterating a linear transformation until it stabilizes.
2. Economics: Input–output models iterate to predict long-term equilibrium of industries.
3. Physics: Time evolution of quantum states is modeled by repeated application of unitary operators.
4. Numerical methods: Iterative solvers (like power iteration) approximate eigenvectors.
5. Computer graphics: Iterated function systems generate self-similar fractals.

Why It Matters

1. Composition unifies matrix multiplication and transformation chaining.
2. Powers reveal exponential growth, decay, and oscillation.
3. Iteration is the core of modeling dynamic processes in mathematics, science, and engineering.
4. The link to eigenvalues makes these ideas the foundation of stability analysis.

Try It Yourself

1. Let $T(x, y) = (x + y, y)$. Compute $T^2(x, y)$ and $T^3(x, y)$. What happens as $n \to \infty$?
2. Consider rotation by $90°$ in $\mathbb{R}^2$. Show that $T^4 = I$.
3. For matrix $A = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$, iterate A^n. What happens to arbitrary vectors?
4. Challenge: Prove that if A is diagonalizable as $A = PDP^{-1}$, then $A^n = PD^nP^{-1}$. Use this to analyze long-term behavior.

Composition, powers, and iteration take linear algebra beyond static equations into the world of processes over time. They explain how small, repeated steps shape long-term outcomes-whether stabilizing systems, amplifying signals, or creating infinite complexity.

46. Similarity and Conjugation

In linear algebra, different matrices can represent the same underlying transformation when written in different coordinate systems. This relationship is captured by the idea of similarity. Two matrices are similar if one is obtained from the other by a conjugation with an invertible change-of-basis matrix. This concept is central to understanding canonical forms, eigenvalue decompositions, and the deep structure of linear operators.

Definition of Similarity

Two $n \times n$ matrices A and B are called similar if there exists an invertible matrix P such that:

$$B = P^{-1}AP.$$

- Here, P represents a change of basis.
- A and B describe the same linear transformation, but expressed relative to different bases.

Conjugation as Change of Basis

Suppose $T : V \to V$ is a linear transformation and A is its matrix in basis B. If we switch to a new basis C, the matrix becomes B. The conversion is:

$$B = P^{-1}AP,$$

where P is the change-of-basis matrix from basis B to basis C.

This shows that similarity is not just algebraic coincidence-it's geometric: the operator is the same, but our perspective (basis) has changed.

Properties Preserved Under Similarity

If A and B are similar, they share many key properties:

1. Determinant: $\det(A) = \det(B)$.
2. Trace: $\mathrm{tr}(A) = \mathrm{tr}(B)$.
3. Rank: $\mathrm{rank}(A) = \mathrm{rank}(B)$.
4. Eigenvalues: Same set of eigenvalues (with multiplicity).
5. Characteristic polynomial: Identical.
6. Minimal polynomial: Identical.

These invariants define the "skeleton" of a linear operator, unaffected by coordinate changes.

Examples

1. Rotation in the plane: The matrix for rotation by 90° is

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

   In another basis, the rotation might be represented by a more complicated-looking matrix, but all such matrices are similar to A.

2. Diagonalization: A matrix A is diagonalizable if it is similar to a diagonal matrix D. That is,

$$A = PDP^{-1}.$$

   Here, similarity reduces A to its simplest form.

3. Shear transformation: A shear matrix $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ is not diagonalizable, but it may be similar to a Jordan block.

Geometric Interpretation

- Similarity says: two matrices may look different, but they are "the same" transformation seen from different coordinate systems.
- Conjugation is the mathematical act of relabeling coordinates.
- Think of shifting your camera angle: the scene hasn't changed, only the perspective has.

Applications

1. Diagonalization: Reducing a matrix to diagonal form (when possible) uses similarity. This simplifies powers, exponentials, and iterative analysis.
2. Jordan canonical form: Every square matrix is similar to a Jordan form, giving a complete structural classification.
3. Quantum mechanics: Operators on state spaces often change representation, but similarity guarantees invariance of spectra.
4. Control theory: Canonical forms simplify analysis of system stability and controllability.
5. Numerical methods: Eigenvalue algorithms rely on repeated similarity transformations (e.g., QR algorithm).

Why It Matters

1. Similarity reveals the true identity of a linear operator, independent of coordinates.
2. It allows simplification: many problems become easier in the right basis.
3. It preserves invariants, giving us tools to classify and compare operators.
4. It connects abstract algebra with concrete computations in geometry, physics, and engineering.

Try It Yourself

1. Show that $\begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$ is similar to $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$. Why or why not?
2. Compute $P^{-1}AP$ for $A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $P = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. Interpret the result.
3. Prove that if two matrices are similar, they must have the same trace.
4. Challenge: Show that if A and B are similar, then A^k and B^k are also similar for all integers $k \geq 0$.

Similarity and conjugation elevate linear algebra from mere calculation to structural understanding. They tell us when two seemingly different matrices are just different "faces" of the same underlying transformation.

47. Projections and Reflections

Among the many transformations in linear algebra, two stand out for their geometric clarity and practical importance: projections and reflections. These operations reshape vectors in simple but powerful ways, and they form the building blocks of algorithms in statistics, optimization, graphics, and physics.

Projection: Flattening onto a Subspace

A projection is a linear transformation that takes a vector and drops it onto a subspace, like casting a shadow.

Formally, if W is a subspace of V, the projection of a vector v onto W is the unique vector $w \in W$ that is closest to v.

In $\mathbb{R}^2$: projecting onto the x-axis takes (x, y) and produces $(x, 0)$.

Orthogonal Projection Formula

Suppose u is a nonzero vector. The projection of v onto the line spanned by u is:

$$\text{proj}_u(v) = \frac{v \cdot u}{u \cdot u} u.$$

This formula works in any dimension. It uses the dot product to measure how much of v points in the direction of u.

Example: Project $(2, 3)$ onto $u = (1, 1)$:

$$\text{proj}_u(2, 3) = \frac{(2, 3) \cdot (1, 1)}{(1, 1) \cdot (1, 1)}(1, 1) = \frac{5}{2}(1, 1) = (2.5, 2.5).$$

The vector $(2, 3)$ splits into $(2.5, 2.5)$ along the line plus $(-0.5, 0.5)$ orthogonal to it.

Projection Matrices

For unit vector u:

$$P = uu^T$$

is the projection matrix onto the span of u.

For a general subspace with orthonormal basis columns in matrix Q:

$$P = QQ^T$$

projects any vector onto that subspace.

Properties:

1. $P^2 = P$ (idempotent).
2. $P^T = P$ (symmetric, for orthogonal projections).

Reflection: Flipping Across a Subspace

A reflection takes a vector and flips it across a line or plane. Geometrically, it's like a mirror.

Reflection across a line spanned by unit vector u:

$$R(v) = 2\text{proj}_u(v) - v.$$

Matrix form:

$$R = 2uu^T - I.$$

Example: Reflect $(2, 3)$ across the line $y = x$. With $u = (1/\sqrt{2}, 1/\sqrt{2})$:

$$R = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

So reflection swaps coordinates: $(2, 3) \mapsto (3, 2)$.

Geometric Insight

- Projection shortens vectors by removing components orthogonal to the subspace.
- Reflection preserves length but flips orientation relative to the subspace.
- Projection is about approximation ("closest point"), reflection is about symmetry.

Applications

1. Statistics & Machine Learning: Least-squares regression is projection of data onto the span of predictor variables.
2. Computer Graphics: Projection transforms 3D scenes into 2D screen images. Reflections simulate mirrors and shiny surfaces.
3. Optimization: Projections enforce constraints by bringing guesses back into feasible regions.
4. Physics: Reflections describe wave behavior, optics, and particle interactions.
5. Numerical Methods: Projection operators are key to iterative algorithms (like Krylov subspace methods).

Why It Matters

1. Projection captures the essence of approximation: keeping what fits, discarding what doesn't.
2. Reflection embodies symmetry and invariance, key to geometry and physics.
3. Both are linear, with elegant matrix representations.
4. They combine easily with other transformations, making them versatile in computation.

Try It Yourself

1. Find the projection matrix onto the line spanned by $(3, 4)$. Verify it is idempotent.
2. Compute the reflection of $(1, 2)$ across the x-axis.
3. Show that reflection matrices are orthogonal ($R^T R = I$).
4. Challenge: For subspace W with orthonormal basis Q, derive the reflection matrix $R = 2QQ^T - I$.

Projections and reflections are two of the purest examples of how linear transformations embody geometric ideas. One approximates, the other symmetrizes-but both expose the deep structure of space through the lens of linear algebra.

48. Rotations and Shear

Linear transformations can twist, turn, and distort space in strikingly different ways. Two of the most fundamental examples are rotations-which preserve lengths and angles while turning vectors-and shears-which slide one part of space relative to another, distorting shape while often preserving area. These two transformations form the geometric heart of linear algebra, and they are indispensable in graphics, physics, and engineering.

Rotations in the Plane

A rotation in $\mathbb{R}^2$ by an angle θ is defined as:

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

For any vector (x, y):

$$R_\theta \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

Properties:

1. Preserves lengths: $\|R_\theta v\| = \|v\|$.
2. Preserves angles: the dot product is unchanged.
3. Determinant $= +1$, so it preserves orientation and area.
4. Inverse: $R_\theta^{-1} = R_{-\theta}$.

Example: A 90° rotation:

$$R_{90^\circ} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \quad (1,0) \mapsto (0,1).$$

Rotations in Three Dimensions

Rotations in $\mathbb{R}^3$ occur around an axis. For example, rotation by angle θ around the z-axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

- Leaves the z-axis fixed.
- Rotates the xy-plane like a 2D rotation.

General rotations in 3D are described by orthogonal matrices with determinant $+1$, forming the group $SO(3)$.

Shear Transformations

A shear slides one coordinate direction while keeping another fixed, distorting shapes.

In $\mathbb{R}^2$:

$$S = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 1 & 0 \\ k & 1 \end{bmatrix}.$$

- The first form "slides" x-coordinates depending on y.
- The second form slides y-coordinates depending on x.

Example:

$$S = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad (x,y) \mapsto (x+y, y).$$

- Squares become parallelograms.
- Areas are preserved if $\det(S) = 1$.

In $\mathbb{R}^3$: shears distort volumes while preserving parallelism of faces.

Geometric Comparison

- Rotation: Preserves size and shape exactly, only changes orientation. Circles remain circles.
- Shear: Distorts shape but often preserves area (in 2D) or volume (in 3D). Circles become ellipses or slanted figures.

Together, rotations and shears can generate a vast variety of linear distortions.

Applications

1. Computer Graphics: Rotations orient objects; shears simulate perspective.
2. Engineering: Shear stresses deform materials; rotations model rigid-body motion.
3. Robotics: Rotations define arm orientation; shears approximate local deformations.
4. Physics: Rotations are symmetries of space; shears appear in fluid flows and elasticity.
5. Data Science: Shears represent changes of variables that preserve volume but distort distributions.

Why It Matters

1. Rotations model pure symmetry-no distortion, just reorientation.
2. Shears show how geometry can be distorted while preserving volume or area.
3. Both are building blocks: any invertible matrix in $\mathbb{R}^2$ can be factored into rotations, shears, and scalings.
4. They bridge algebra and geometry, giving visual meaning to abstract matrices.

Try It Yourself

1. Rotate $(1, 0)$ by $60°$ and compute the result explicitly.
2. Apply the shear $S = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ to the square with vertices $(0, 0), (1, 0), (0, 1), (1, 1)$. What shape results?
3. Show that rotation matrices are orthogonal ($R^T R = I$).
4. Challenge: Prove that any area-preserving 2×2 matrix with determinant 1 can be decomposed into a product of rotations and shears.

Rotations and shears highlight two complementary sides of linear algebra: symmetry versus distortion. Together, they show how transformations can either preserve the essence of space or bend it into new shapes while keeping its structure intact.

49. Rank and Operator Viewpoint

The rank of a linear transformation or matrix is one of the most important measures of its power. It captures how many independent directions a transformation preserves, how much information it carries from input to output, and how "full" its action on space is. Thinking of rank not just as a number, but as a description of an operator, gives us a clearer picture of what transformations really do.

Definition of Rank

For a matrix A representing a linear transformation $T : V \to W$:

$$\operatorname{rank}(A) = \dim(\operatorname{im}(A)) = \dim(\operatorname{im}(T)).$$

That is, the rank is the dimension of the image (or column space). It counts the maximum number of linearly independent columns.

Basic Properties

1. $\operatorname{rank}(A) \leq \min(m, n)$ for an $m \times n$ matrix.
2. $\operatorname{rank}(A) = \operatorname{rank}(A^T)$.
3. Rank is equal to the number of pivot columns in row-reduced form.
4. Rank links directly with nullity via the rank–nullity theorem:

$$\operatorname{rank}(A) + \operatorname{nullity}(A) = n.$$

Operator Perspective

Instead of focusing on rows and columns, imagine rank as a measure of how much of the domain is transmitted faithfully to the codomain.

- If rank = full (n), the transformation is injective: nothing collapses.
- If rank = dimension of codomain (m), the transformation is surjective: every target vector can be reached.
- If rank is smaller, the transformation compresses space: parts of the domain are "invisible" and collapse into the kernel.

Example 1 (Projection): Projection from $\mathbb{R}^3$ onto the xy-plane has rank 2. It annihilates the z-direction but preserves two independent directions.

Example 2 (Rotation): Rotation in $\mathbb{R}^2$ has rank 2. No directions are lost.

Example 3 (Zero map): The transformation sending everything to zero has rank 0.

Geometric Meaning

- Rank = number of independent directions preserved.
- A rank-1 transformation maps all of space onto a single line.
- Rank-2 in $\mathbb{R}^3$ maps space onto a plane.
- Rank-full maps space onto its entire dimension without collapse.

Visually: rank describes the "dimensional thickness" of the image.

Rank and Matrix Factorizations

Rank reveals hidden structure:

1. LU factorization: Rank determines the number of nonzero pivots.
2. QR factorization: Rank controls the number of orthogonal directions.
3. SVD (Singular Value Decomposition): The number of nonzero singular values equals the rank.

SVD in particular gives a geometric operator view: each nonzero singular value corresponds to a preserved dimension, while zeros indicate collapsed directions.

Rank in Applications

1. Data compression: Low-rank approximations reduce storage (e.g., image compression with SVD).
2. Statistics: Rank of the design matrix determines identifiability of regression coefficients.
3. Machine learning: Rank of weight matrices controls expressive power of models.
4. Control theory: Rank conditions ensure controllability and observability of systems.
5. Network analysis: Rank of adjacency or Laplacian matrices reflects connectivity of graphs.

Rank Deficiency

If a transformation has less than full rank, it is rank-deficient. This means:

- Some directions are lost (kernel nontrivial).
- Some outputs are unreachable (image smaller than codomain).
- Equations $Ax = b$ may be inconsistent or underdetermined.

Detecting and handling rank deficiency is crucial in numerical linear algebra, where ill-conditioning can hide in nearly dependent columns.

Why It Matters

1. Rank measures the true dimensional effect of a transformation.
2. It distinguishes between full-strength operators and those that collapse information.
3. It connects row space, column space, image, and kernel under one number.
4. It underpins algorithms for regression, decomposition, and dimensionality reduction.

Try It Yourself

1. Find the rank of $\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix}$. Why is it less than 2?
2. Describe geometrically the image of a rank-1 transformation in $\mathbb{R}^3$.
3. For a 5×5 diagonal matrix with diagonal entries $(2, 0, 3, 0, 5)$, compute rank and nullity.
4. Challenge: Show that for any matrix A, the rank equals the number of nonzero singular values of A.

Rank tells us not just how many independent vectors survive a transformation, but also how much structure the operator truly preserves. It is the bridge between abstract linear maps and their practical power.

50. Block Matrices and Block Maps

As problems grow in size, matrices become large and difficult to manage element by element. A powerful strategy is to organize matrices into blocks-submatrices grouped together like tiles in a mosaic. This allows us to treat large transformations as compositions of smaller, more understandable ones. Block matrices preserve structure, simplify computations, and reveal deep insights into how transformations act on subspaces.

What Are Block Matrices?

A block matrix partitions a matrix into rectangular submatrices. Each block is itself a matrix, and the entire matrix can be manipulated using block rules.

Example: a 4×4 matrix divided into four 2×2 blocks:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where each A_{ij} is 2×2.

Instead of thinking in terms of 16 entries, we work with 4 blocks.

Block Maps as Linear Transformations

Suppose $V = V_1 \oplus V_2$ is decomposed into two subspaces. A linear map $T : V \to V$ can be described in terms of how it acts on each component. Relative to this decomposition, the matrix of T has block form:

$$[T] = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix}.$$

- T_{11}: how V_1 maps into itself.
- T_{12}: how V_2 contributes to V_1.
- T_{21}: how V_1 contributes to V_2.
- T_{22}: how V_2 maps into itself.

This decomposition highlights how subspaces interact under the transformation.

Block Matrix Operations

Block matrices obey the same rules as normal matrices, but operations are done block by block.

Addition:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} + \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} A+E & B+F \\ C+G & D+H \end{bmatrix}.$$

Multiplication:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

The formulas look like ordinary multiplication, but each term is itself a product of submatrices.

Special Block Structures

1. Block Diagonal Matrices:

$$\begin{bmatrix} A & 0 \\ 0 & D \end{bmatrix}.$$

   Independent actions on subspaces-no mixing between them.

2. Block Upper Triangular:

$$\begin{bmatrix} A & B \\ 0 & D \end{bmatrix}.$$

   Subspace V_1 influences V_2, but not vice versa.

3. Block Symmetric: If overall matrix is symmetric, so are certain block relationships: $A^T = A, D^T = D, B^T = C$.

These structures appear naturally in decomposition and iterative algorithms.

Block Matrix Inverses

Some block matrices can be inverted using special formulas. For

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

if A is invertible, the inverse can be expressed using the Schur complement:

$$M^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}.$$

This formula is central in statistics, optimization, and numerical analysis.

Geometric Interpretation

- A block diagonal matrix acts like two independent transformations operating side by side.
- A block triangular matrix shows a "hierarchy": one subspace influences the other but not the reverse.
- This decomposition mirrors how systems can be separated into smaller interacting parts.

Applications

1. Numerical Linear Algebra: Block operations optimize computation on large sparse matrices.
2. Control Theory: State-space models are naturally expressed in block form.
3. Statistics: Partitioned covariance matrices rely on block inversion formulas.
4. Machine Learning: Neural networks layer transformations, often structured into blocks for efficiency.
5. Parallel Computing: Block decomposition distributes large matrix problems across processors.

Why It Matters

1. Block matrices turn big problems into manageable smaller ones.
2. They reflect natural decompositions of systems into interacting parts.
3. They make explicit the geometry of subspace interactions.
4. They provide efficient algorithms, especially for large-scale scientific computing.

Try It Yourself

1. Multiply two 4×4 matrices written as 2×2 block matrices and confirm the block multiplication rule.
2. Write the projection matrix onto a 2D subspace in $\mathbb{R}^4$ using block form.
3. Compute the Schur complement of

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

4. Challenge: Show that the determinant of a block triangular matrix equals the product of the determinants of its diagonal blocks.

Block matrices and block maps show how complexity can be organized. Instead of drowning in thousands of entries, we see structure, interaction, and hierarchy-revealing how large systems can be built from simple linear pieces.

Closing

```
Shadows twist and turn,
kernels hide and images flow,
form remains within.
```

Chapter 6. Determinants and volume

Opening

```
Areas unfold,
parallels stretch into waves,
scale whispers in signs.
```

51. Areas, Volumes, and Signed Scale Factors

Determinants often feel like an abstract formula until we see their geometric meaning: they measure area in 2D, volume in 3D, and, in higher dimensions, the general "size" of a transformed shape. Even more, determinants encode whether orientation is preserved or flipped, giving them a "signed" interpretation. This perspective transforms determinants from algebraic curiosities into geometric tools.

Transformations and Scaling of Space

Consider a linear transformation $T : \mathbb{R}^n \to \mathbb{R}^n$ represented by a square matrix A. When A acts on vectors, it reshapes space: it stretches, compresses, rotates, reflects, or shears regions.

- If you apply A to a unit square in $\mathbb{R}^2$, the image is a parallelogram.
- If you apply A to a unit cube in $\mathbb{R}^3$, the image is a parallelepiped.
- In general, the determinant of A tells us how the measure (area, volume, hyper-volume) of the shape has changed.

Determinant as Signed Scale Factor

- $|\det(A)|$ = the scale factor for areas (2D), volumes (3D), or n-dimensional content.
- If $\det(A) = 0$, the transformation collapses space into a lower dimension, flattening all volume away.
- If $\det(A) > 0$, the orientation of space is preserved.
- If $\det(A) < 0$, the orientation is flipped (like a reflection in a mirror).

Thus, determinants are not just numbers-they carry both magnitude and sign, telling us about size and handedness.

2D Case: Area of Parallelogram

Take two column vectors $u, v \in \mathbb{R}^2$. Place them as columns in a matrix:

$$A = \begin{bmatrix} u & v \end{bmatrix}.$$

The absolute value of the determinant gives the area of the parallelogram spanned by u and v:

$$\text{Area} = |\det(A)|.$$

Example:

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}.$$

Then $\det(A) = (2)(3) - (1)(1) = 5$. The unit square maps to a parallelogram of area 5.

3D Case: Volume of Parallelepiped

For three vectors $u, v, w \in \mathbb{R}^3$, form a matrix

$$A = \begin{bmatrix} u & v & w \end{bmatrix}.$$

Then the absolute determinant gives the volume of the parallelepiped:

$$\text{Volume} = |\det(A)|.$$

Geometrically, this is the scalar triple product:

$$\det(A) = u \cdot (v \times w).$$

Example:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}, \quad \det(A) = 6.$$

So the unit cube is stretched into a box with volume 6.

Orientation and Signed Measure

Determinants do more than measure size-they also detect orientation:

- In 2D, flipping x and y axes changes the sign of the determinant.
- In 3D, swapping two vectors changes the "handedness" (right-hand rule becomes left-hand rule).

This explains why determinants can be negative: they mark transformations that reverse orientation.

Higher Dimensions

In $\mathbb{R}^n$, determinants extend the same idea. A unit hypercube (side length 1) is transformed into an n-dimensional parallelotope, whose volume is given by $|\det(A)|$.

Though we cannot visualize beyond 3D, the concept generalizes smoothly: determinants encode how much an n-dimensional object is stretched or collapsed.

Applications

1. Geometry: Computing areas, volumes, and orientation directly from vectors.
2. Computer Graphics: Determinants detect whether a transformation preserves or flips orientation, useful in rendering.
3. Physics: Determinants describe Jacobians for coordinate changes in integrals, adjusting volume elements.
4. Engineering: Determinants quantify deformation and stress in materials (strain tensors).
5. Data Science: Determinants of covariance matrices encode "volume" of uncertainty ellipsoids.

Why It Matters

1. Determinants connect algebra (formulas) to geometry (shapes).
2. They explain why some transformations lose information: $\det = 0$.
3. They preserve orientation, key for consistent physical laws and geometry.
4. They prepare us for advanced tools like Jacobians, eigenvalues, and volume-preserving maps.

Try It Yourself

1. Compute the area of the parallelogram spanned by $(1, 2)$ and $(3, 1)$.
2. Find the volume of the parallelepiped defined by vectors $(1, 0, 0), (0, 1, 0), (1, 1, 1)$.
3. Show that swapping two columns of a matrix flips the sign of the determinant but keeps absolute value unchanged.
4. Challenge: Explain why $\det(A)$ gives the scaling factor for integrals under change of variables.

Determinants begin as algebraic formulas, but their real meaning lies in geometry: they measure how linear transformations scale, compress, or flip space itself.

52. Determinant via Linear Rules

The determinant is not just a mysterious formula-it is a function built from a few simple rules that uniquely determine its behavior. These rules, often called determinant axioms, allow us to see the determinant as the only measure of "signed volume" compatible with linear algebra. Understanding these rules gives clarity: instead of memorizing expansion formulas, we see why determinants behave as they do.

The Setup

Take a square matrix $A \in \mathbb{R}^{n \times n}$. Think of A as a list of n column vectors:

$$A = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}.$$

The determinant is a function $\det : \mathbb{R}^{n \times n} \to \mathbb{R}$ that assigns a single number to A. Geometrically, it gives the signed volume of the parallelotope spanned by $(a_1, \ldots, a_n)$. Algebraically, it follows three key rules.

Rule 1: Linearity in Each Column

If you scale one column by a scalar c, the determinant scales by c.

$$\det(a_1, \dots, ca_j, \dots, a_n) = c \cdot \det(a_1, \dots, a_j, \dots, a_n).$$

If you replace a column with a sum, the determinant splits:

$$\det(a_1, \dots, (b + c), \dots, a_n) = \det(a_1, \dots, b, \dots, a_n) + \det(a_1, \dots, c, \dots, a_n).$$

This linearity means determinants behave predictably with respect to scaling and addition.

Rule 2: Alternating Property

If two columns are the same, the determinant is zero:

$$\det(\dots, a_i, \dots, a_i, \dots) = 0.$$

This makes sense geometrically: if two spanning vectors are identical, they collapse the volume to zero.

Equivalently: if you swap two columns, the determinant flips sign:

$$\det(\dots, a_i, \dots, a_j, \dots) = -\det(\dots, a_j, \dots, a_i, \dots).$$

Rule 3: Normalization

The determinant of the identity matrix is 1:

$$\det(I_n) = 1.$$

This anchors the function: the unit cube has volume 1, with positive orientation.

Consequence: Uniqueness

These three rules (linearity, alternating, normalization) uniquely define the determinant. Any function satisfying them must be the determinant. This makes it less of an arbitrary formula and more of a natural consequence of linear structure.

Small Cases: Explicit Formulas

- 2×2 matrices:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc.$$

  This formula arises directly from the rules: linearity in columns and alternating sign when swapping them.

- 3×3 matrices: Expansion formula:

$$\det \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aei + bfg + cdh - ceg - bdi - afh.$$

This looks complicated, but it comes from systematically applying the rules to break down the volume.

Geometric Interpretation of the Rules

1. Linearity: Stretching one side of a parallelogram or parallelepiped scales the area or volume.
2. Alternating: If two sides collapse into the same direction, the area/volume vanishes. Swapping sides flips orientation.
3. Normalization: The unit cube has size 1 by definition.

Together, these mirror geometric intuition exactly.

Higher-Dimensional Generalization

In $\mathbb{R}^n$, determinants measure oriented hyper-volume. For example, in 4D, determinants give the "4-volume" of a parallelotope. Though impossible to picture, the same rules apply.

Applications

1. Defining area and volume: Determinants provide a universal formula for computing geometric sizes from coordinates.
2. Jacobian determinants: Used in calculus when changing variables in multiple integrals.
3. Orientation detection: Whether transformations preserve handedness in geometry or physics.
4. Computer graphics: Ensuring consistent orientation of polygons and meshes.

Why It Matters

Determinants are not arbitrary. They arise naturally once we demand a function that is linear in columns, alternating, and normalized. This explains why so many different formulas and properties agree: they are all shadows of the same underlying definition.

Try It Yourself

1. Show that scaling one column by 3 multiplies the determinant by 3.
2. Compute the determinant of $\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ and explain why it is zero.
3. Swap two columns in $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and confirm the determinant changes sign.
4. Challenge: Use only the three rules to derive the 2×2 determinant formula.

The determinant is the unique bridge between algebra and geometry, born from a handful of simple but powerful rules.

53. Determinant and Row Operations

One of the most practical ways to compute determinants is by using row operations, the same tools used in Gaussian elimination. Determinants interact with these operations in very structured ways. By understanding the rules, we can compute determinants systematically without resorting to long expansion formulas.

Row Operations Recap

There are three elementary row operations:

1. Row Swap ($\mathrm{R}_i \leftrightarrow \mathrm{R}_j$) – exchange two rows.
2. Row Scaling ($\mathrm{c} \cdot \mathrm{R}_i$) – multiply a row by a scalar c.
3. Row Replacement ($\mathrm{R}_i + \mathrm{c} \cdot \mathrm{R}_j$) – replace one row with itself plus a multiple of another row.

Since the determinant is defined in terms of linearity and alternation of rows (or columns), each operation has a clear effect.

Rule 1: Row Swap Changes Sign

If you swap two rows, the determinant changes sign:

$$\det(A \text{ with } R_i \leftrightarrow R_j) = -\det(A).$$

Reason: Swapping two spanning vectors flips orientation. In 2D, swapping basis vectors flips a parallelogram across the diagonal, reversing handedness.

Rule 2: Row Scaling Multiplies Determinant

If you multiply a row by a scalar c:

$$\det(A \text{ with } cR_i) = c \cdot \det(A).$$

Reason: Scaling one side of a parallelogram multiplies its area; scaling one dimension of a cube multiplies its volume.

Rule 3: Row Replacement Leaves Determinant Unchanged

If you replace one row with itself plus a multiple of another row:

$$\det(A \text{ with } R_i \to R_i + cR_j) = \det(A).$$

Reason: Adding a multiple of one spanning vector to another doesn't change the spanned volume. The parallelogram or parallelepiped is sheared, but its area or volume remains the same.

Why These Rules Work Together

These three rules align perfectly with the determinant axioms:

- Alternating $\to$ row swaps flip sign.
- Linearity $\to$ scaling multiplies by scalar.
- Normalization $\to$ row replacement preserves measure.

Thus, row operations provide a complete framework for computing determinants.

Computing Determinants with Elimination

To compute $\det(A)$:

1. Perform Gaussian elimination to reduce A to an upper triangular matrix U.
2. Track how row swaps and scalings affect the determinant.
3. Use the fact that the determinant of a triangular matrix is the product of its diagonal entries.

Example:

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 1 & 7 \\ -2 & 5 & 1 \end{bmatrix}.$$

- Step 1: $R_2 \to R_2 - 2R_1$, $R_3 \to R_3 + R_1$. No determinant change.
- Step 2: Upper triangular form emerges:

$$U = \begin{bmatrix} 2 & 1 & 3 \\ 0 & -1 & 1 \\ 0 & 0 & -5 \end{bmatrix}.$$

- Step 3: Determinant is product of diagonals: $\det(A) = 2 \cdot (-1) \cdot (-5) = 10$.

Efficient, clear, and no messy cofactor expansions.

Geometric View

- Row swap: Flips orientation of the volume.
- Row scaling: Stretches or compresses one dimension of the volume.
- Row replacement: Slides faces of the volume without changing its size.

This geometric reasoning reinforces why the rules are natural.

Applications

1. Efficient computation: Algorithms for large determinants (LU decomposition) are based on row operations.
2. Numerical analysis: Determinant rules help detect stability and singularity.
3. Geometry: Orientation tests for polygons rely on row swap rules.
4. Theoretical results: Many determinant identities are derived directly from row operation behavior.

Why It Matters

- Determinants link algebra to geometry, but computation requires efficient methods.
- Row operations give a hands-on toolkit: they're the backbone of practical determinant computation.
- Understanding these rules explains why algorithms like LU factorization work so well.

Try It Yourself

1. Compute the determinant of $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ using elimination.
2. Verify that replacing $R_2 \to R_2 + 3R_1$ does not change the determinant.
3. Check how many sign flips occur if you reorder rows into strictly increasing order.
4. Challenge: Prove that elimination combined with these rules always leads to the triangular product formula.

Determinants are not meant to be expanded by brute force; row operations transform the problem into a clear sequence of steps, connecting algebraic efficiency with geometric intuition.

54. Triangular Matrices and Product of Diagonals

Among all types of matrices, triangular matrices stand out for their simplicity. These are matrices where every entry either above or below the main diagonal is zero. What makes them especially important is that their determinants can be computed almost instantly: the determinant of a triangular matrix is simply the product of its diagonal entries. This property is not only computationally convenient, it also reveals deep connections between determinants, row operations, and structure in linear algebra.

Triangular Matrices Defined

A square matrix is called upper triangular if all entries below the main diagonal are zero, and lower triangular if all entries above the diagonal are zero.

- Upper triangular example:

$$U = \begin{bmatrix} 2 & 5 & -1 \\ 0 & 3 & 4 \\ 0 & 0 & 7 \end{bmatrix}.$$

- Lower triangular example:

$$L = \begin{bmatrix} 4 & 0 & 0 \\ -2 & 5 & 0 \\ 1 & 3 & 6 \end{bmatrix}.$$

Both share the key feature: "everything off one side of the diagonal vanishes."

Determinant Rule

For any triangular matrix,

$$\det(T) = \prod_{i=1}^{n} t_{ii},$$

where t_{ii} are the diagonal entries.

So for the upper triangular U above,

$$\det(U) = 2 \times 3 \times 7 = 42.$$

Why This Works

The determinant is multilinear and alternating. When you expand it (e.g., via cofactor expansion), only one product of entries survives in the expansion: the one that picks exactly the diagonal terms.

- If you try to pick an off-diagonal entry in a row, you eventually get stuck with a zero entry because of the triangular shape.
- The only surviving term is the product of the diagonals, with sign $+1$.

This elegant reasoning explains why the rule holds universally.

Connection to Row Operations

Recall: elimination reduces any square matrix to an upper triangular form. Once triangular, the determinant is simply the product of the diagonals, adjusted for row swaps and scalings.

Thus, triangular matrices are not just simple-they are the end goal of elimination algorithms for determinant computation.

Geometric Meaning

In geometric terms:

- A triangular matrix represents a transformation where each coordinate direction depends only on itself and earlier coordinates.
- The determinant equals the product of scaling along each axis.
- Example: In 3D, scaling x by 2, y by 3, and z by 7 gives a volume scaling of $2 \cdot 3 \cdot 7 = 42$.

Even if shear is present in the upper entries, the determinant ignores it-it only cares about the pure diagonal scaling.

Applications

1. Efficient computation: LU decomposition reduces determinants to diagonal product form.
2. Theoretical proofs: Many determinant identities reduce to triangular cases.
3. Numerical stability: Triangular matrices are well-behaved in computation, crucial for algorithms in numerical linear algebra.
4. Eigenvalues: For triangular matrices, eigenvalues are exactly the diagonal entries; thus determinant = product of eigenvalues.
5. Computer graphics: Triangular forms simplify geometric transformations.

Why It Matters

1. Provides the fastest way to compute determinants in special cases.
2. Serves as the computational foundation for general determinant algorithms.
3. Connects determinants directly to eigenvalues and scaling factors.
4. Illustrates how elimination transforms complexity into simplicity.

Try It Yourself

1. Compute the determinant of

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}.$$

   (Check: it should equal $1 \cdot 4 \cdot 6$).

2. Verify that a lower triangular matrix with diagonal entries $(2, -1, 5)$ has determinant -10.

3. Explain why an upper triangular matrix with a zero on the diagonal must have determinant 0.
4. Challenge: Prove that every square matrix can be reduced to triangular form with determinant tracked by elimination steps.

The triangular case reveals the heart of determinants: a product of diagonal scalings, stripped of all extra noise. It is the simplest lens through which determinants become transparent.

55. The Multiplicative Property of Determinants: $\det(AB) = \det(A)\det(B)$

One of the most remarkable and useful facts about determinants is that they multiply across matrix products. For two square matrices of the same size,

$$\det(AB) = \det(A) \cdot \det(B).$$

This property is fundamental: it connects algebra (matrix multiplication) with geometry (scaling volumes) and is essential for proofs, computations, and applications across mathematics, physics, and engineering.

The Statement in Words

- If you first apply a linear transformation B, and then apply A, the total scaling of space is the product of their individual scalings.
- Determinants track exactly this: the signed volume change under linear transformations.

Geometric Intuition

Think of $\det(A)$ as the signed scale factor by which A changes volume.

1. Apply B: a unit cube becomes some parallelepiped with volume $|\det(B)|$.
2. Apply A: the new parallelepiped scales again by $|\det(A)|$.
3. Total effect: volume scales by $|\det(A)| \times |\det(B)|$.

The orientation flips are also consistent: if both flip (negative determinants), the total orientation is preserved (positive product).

Algebraic Reasoning

The proof can be approached in multiple ways:

1. Row Operations and Elimination:

   - A and B can be factored into elementary matrices (row swaps, scalings, replacements).
   - Determinants behave predictably for each operation.
   - Since both sides agree for elementary operations and determinant is multiplicative, the identity holds in general.

2. Abstract Characterization:

   - Determinants are the unique multilinear alternating functions normalized at the identity.
   - Composition of linear maps preserves this property, so multiplicativity follows.

Small Cases

- 2×2 matrices:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}.$$

  Compute AB, then $\det(AB)$. After expansion, you find: $\det(AB) = (ad-bc)(eh-fg) = \det(A)\det(B)$.

- 3×3 matrices: A direct computation is messy, but the property still holds and is consistent with elimination proofs.

Key Consequences

1. Determinant of a Power:

$$\det(A^k) = (\det(A))^k.$$

   Geometric meaning: applying the same transformation k times multiplies volume scale repeatedly.

2. Inverse Matrix: If A is invertible,

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

3. Eigenvalues: Since $\det(A)$ is the product of eigenvalues, this property matches the fact that eigenvalues multiply under matrix multiplication (when considered via characteristic polynomials).

Geometric Meaning in Higher Dimensions

- If B scales space by 3 and flips it (det = –3), and A scales by 2 without flipping (det = 2), then AB scales by –6, consistent with the rule.
- Determinants encapsulate both magnitude (volume scaling) and sign (orientation). Multiplicativity ensures these combine correctly.

Applications

1. Change of Variables in Calculus: The Jacobian determinant follows this multiplicative rule, ensuring transformations compose consistently.
2. Group Theory: det defines a group homomorphism from the general linear group GL_n to the nonzero reals under multiplication.
3. Numerical Analysis: Determinant multiplicativity underlies LU decomposition methods.
4. Physics: In mechanics and relativity, volume elements transform consistently under successive transformations.
5. Cryptography and Coding Theory: Determinants in modular arithmetic rely on this multiplicative property to preserve structure.

Why It Matters

- Guarantees consistency: determinants match our intuition about scaling.
- Simplifies computation: determinants of factorizations can be obtained by multiplying smaller pieces.
- Provides theoretical structure: det is a homomorphism, embedding linear algebra into the algebra of scalars.

Try It Yourself

1. Verify $\det(AB) = \det(A)\det(B)$ for

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 4 \\ 0 & -2 \end{bmatrix}.$$

2. Prove that $\det(A^{-1}) = 1/\det(A)$ using the multiplicative rule.
3. Show that if $\det(A) = 0$, then $\det(AB) = 0$ for any B. Explain why this makes sense geometrically.
4. Challenge: Using row operations, show explicitly how multiplicativity emerges from properties of elementary matrices.

The rule $\det(AB) = \det(A)\det(B)$ transforms determinants from a mysterious calculation into a natural and consistent measure of how linear transformations combine.

56. Invertibility and Zero Determinant

The determinant is more than a geometric scale factor-it is the ultimate test of whether a matrix is invertible. A square matrix $A \in \mathbb{R}^{n\times n}$ has an inverse if and only if its determinant is nonzero. When the determinant vanishes, the matrix collapses space into a lower dimension, losing information that no transformation can undo.

The Criterion

$$A \text{ invertible} \iff \det(A) \neq 0.$$

- If $\det(A) \neq 0$, the transformation stretches or shrinks space but never flattens it. Every output corresponds to exactly one input, so A^{-1} exists.
- If $\det(A) = 0$, some directions are squashed into lower dimensions. Information is destroyed, so no inverse exists.

Geometric Meaning

1. In 2D:
    - A nonzero determinant means the unit square is sent to a parallelogram with nonzero area.
    - A zero determinant means the square collapses into a line segment or a point.

2. In 3D:

   - Nonzero determinant → unit cube becomes a 3D parallelepiped with volume.
   - Zero determinant → cube flattens into a sheet or a line; 3D volume is lost.

3. In Higher Dimensions:

   - Nonzero determinant preserves n-dimensional volume.
   - Zero determinant collapses dimension, destroying invertibility.

Algebraic Meaning

- The determinant is the product of eigenvalues:

$$\det(A) = \lambda_1 \lambda_2 \cdots \lambda_n.$$

  If any eigenvalue is zero, then $\det(A) = 0$ and the matrix is singular (not invertible).

- Equivalently, a zero determinant means the matrix has linearly dependent columns or rows. This dependence implies redundancy: not all directions are independent, so the mapping cannot be one-to-one.

Connection with Linear Systems

- If $\det(A) \neq 0$:
  - The system $Ax = b$ has a unique solution for every b.
  - The inverse matrix A^{-1} exists and satisfies $x = A^{-1}b$.

- If $\det(A) = 0$:
  - Either no solutions (inconsistent system) or infinitely many solutions (dependent equations).
  - The mapping $x \mapsto Ax$ cannot be reversed.

Example: Invertible vs. Singular

1.

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \det(A) = 5 \neq 0.$$

Invertible.

2.

$$B = \begin{bmatrix} 2 & 4 \\ 1 & 2 \end{bmatrix}, \quad \det(B) = 0.$$

Not invertible, since the second column is just twice the first.

Applications

1. Solving Systems: Inverse-based methods rely on nonzero determinants.
2. Numerical Methods: Detecting near-singularity warns of unstable solutions.
3. Geometry: A singular matrix corresponds to degenerate shapes (flattened, collapsed).
4. Physics: In mechanics and relativity, invertibility ensures that transformations can be reversed.
5. Computer Graphics: Non-invertible transformations crush dimensions, breaking rendering pipelines.

Why It Matters

- Determinants provide a single scalar test for invertibility.
- This connects geometry (volume collapse), algebra (linear dependence), and analysis (solvability of systems).
- The zero/nonzero divide is one of the sharpest and most important in all of linear algebra.

Try It Yourself

1. Determine whether

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

   is invertible. Explain both geometrically and algebraically.

2. For

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

   compute the determinant and describe the geometric transformation.

3. Challenge: Show that if $\det(A) = 0$, the rows (or columns) of A are linearly dependent.

The determinant acts as the ultimate yes-or-no test: nonzero means full-dimensional, reversible transformation; zero means collapse and irreversibility.

57. Cofactor Expansion

While elimination gives a practical way to compute determinants, the cofactor expansion (also called Laplace expansion) offers a recursive definition that works for all square matrices. It expresses the determinant of an $n \times n$ matrix in terms of determinants of smaller $(n-1) \times (n-1)$ matrices. This method reveals the internal structure of determinants and serves as a bridge between theory and computation.

Minors and Cofactors

- The minor M_{ij} of an entry a_{ij} is the determinant of the submatrix obtained by deleting the i-th row and j-th column from A.
- The cofactor C_{ij} adds a sign factor:

$$C_{ij} = (-1)^{i+j} M_{ij}.$$

Thus each entry contributes to the determinant through its cofactor, with alternating signs arranged in a checkerboard pattern:

$$\begin{bmatrix} + & - & + & - & \cdots \\ - & + & - & + & \cdots \\ + & - & + & - & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

The Expansion Formula

For any row i:

$$\det(A) = \sum_{j=1}^{n} a_{ij} C_{ij}.$$

Or for any column j:

$$\det(A) = \sum_{i=1}^{n} a_{ij} C_{ij}.$$

That is, the determinant can be computed by expanding along any row or column.

Example: 3×3 Case

Let

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

Expanding along the first row:

$$\det(A) = a \cdot \det \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \cdot \det \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \cdot \det \begin{bmatrix} d & e \\ g & h \end{bmatrix}.$$

Simplify each 2×2 determinant:

$$= a(ei - fh) - b(di - fg) + c(dh - eg).$$

This matches the familiar expansion formula for 3×3 determinants.

Why It Works

Cofactor expansion follows directly from the multilinearity and alternating rules of determinants:

- Only one element per row and per column contributes to each term.
- Signs alternate because swapping rows/columns reverses orientation.
- Recursive expansion reduces the problem size until reaching 2×2 determinants, where the formula is simple.

Computational Complexity

- For $n = 2$, expansion is immediate.
- For $n = 3$, expansion is manageable.
- For large n, expansion is very inefficient: computing $\det(A)$ via cofactors requires $O(n!)$ operations.

That's why in practice, elimination or LU decomposition is preferred. Cofactor expansion is best for theory, proofs, and small matrices.

Geometric Interpretation

Each cofactor corresponds to excluding one direction (row/column), measuring the volume of the remaining sub-parallelotope. The alternating sign keeps track of orientation. Thus the determinant is a weighted combination of contributions from all entries along a chosen row or column.

Applications

1. Theoretical proofs: Cofactor expansion underlies many determinant identities.
2. Adjugate matrix: Cofactors form the adjugate used in the explicit formula for matrix inverses.
3. Eigenvalues: Characteristic polynomials use cofactor expansion.
4. Geometry: Cofactors describe signed volumes of faces of higher-dimensional shapes.

Why It Matters

- Cofactor expansion connects determinants across dimensions.
- It provides a universal definition independent of row operations.
- It explains why determinants behave consistently with volume, orientation, and algebraic rules.

Try It Yourself

1. Expand the determinant of

$$\begin{bmatrix} 2 & 1 & 3 \\ 0 & -1 & 4 \\ 1 & 2 & 0 \end{bmatrix}$$

   along the first row.

2. Compute the same determinant by expanding along the second column. Verify the result matches.

3. Show that expanding along two different rows gives the same determinant.

4. Challenge: Prove by induction that cofactor expansion works for all $n \times n$ matrices.

Cofactor expansion is not the fastest method, but it reveals the recursive structure of determinants and explains why they hold their rich algebraic and geometric meaning.

58. Permutations and the Sign of the Determinant

Behind every determinant formula lies a hidden structure: permutations. Determinants can be expressed as a weighted sum over all possible ways of selecting one entry from each row and each column of a matrix. The weight for each selection is determined by the sign of the permutation used. This viewpoint reveals why determinants encode orientation and why their formulas alternate between positive and negative terms.

The Permutation Definition

Let S_n denote the set of all permutations of n elements. Each permutation $\sigma \in S_n$ rearranges the numbers $\{1, 2, \ldots, n\}$.

The determinant of an $n \times n$ matrix $A = [a_{ij}]$ is defined as:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}.$$

- Each product $\prod_{i=1}^{n} a_{i,\sigma(i)}$ picks one entry from each row and each column, according to σ.
- The factor $\operatorname{sgn}(\sigma)$ is $+1$ if σ is an even permutation (achieved by an even number of swaps), and -1 if it is odd.

Why Permutations Appear

A determinant requires:

1. Linearity in each row.
2. Alternating property (row swaps flip the sign).
3. Normalization ($\det(I) = 1$).

When you expand by multilinearity, all possible combinations of choosing one entry per row and column arise. The alternating rule enforces that terms with repeated columns vanish, leaving only permutations. The sign of each permutation enforces the orientation flip.

Example: 2×2 Case

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

There are two permutations in S_2:

- Identity $(1, 2)$: sign $+1$, contributes $a \cdot d$.
- Swap $(2, 1)$: sign -1, contributes $-bc$.

So,

$$\det(A) = ad - bc.$$

Example: 3×3 Case

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

There are $3! = 6$ permutations:

1. $(1, 2, 3)$: even, $+aei$.
2. $(1, 3, 2)$: odd, $-afh$.
3. $(2, 1, 3)$: odd, $-bdi$.
4. $(2, 3, 1)$: even, $+bfg$.
5. $(3, 1, 2)$: even, $+cdh$.
6. $(3, 2, 1)$: odd, $-ceg$.

So,

$$\det(A) = aei + bfg + cdh - ceg - bdi - afh.$$

This is exactly the cofactor expansion result, but now explained as a permutation sum.

Geometric Meaning of Signs

- Even permutations correspond to consistent orientation of basis vectors.
- Odd permutations correspond to flipped orientation.
- The determinant alternates signs because flipping axes reverses handedness.

Counting Growth

- For $n = 4$, there are $4! = 24$ terms.
- For $n = 5$, $5! = 120$ terms.
- In general, $n!$ terms make this formula impractical for large matrices.
- Still, it gives the deepest definition of determinants, from which all other rules follow.

Applications

1. Abstract algebra: Determinant definition via permutations works over any field.
2. Combinatorics: Determinants encode signed sums over permutations, connecting to permanents.
3. Theoretical proofs: Many determinant properties, like multiplicativity, emerge cleanly from the permutation definition.
4. Leibniz formula: Explicit but impractical formula for computation.
5. Advanced math: Determinants generalize to alternating multilinear forms in linear algebra and differential geometry.

Why It Matters

- Provides the most fundamental definition of determinants.
- Explains alternating signs in formulas naturally.
- Bridges algebra, geometry, and combinatorics.
- Shows how orientation emerges from row/column arrangements.

Try It Yourself

1. Write out all 6 terms in the 3×3 determinant expansion and verify the sign of each permutation.
2. Compute the determinant of $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ using the permutation definition.
3. Show that if two columns are equal, all permutation terms cancel, giving $\det(A) = 0$.
4. Challenge: Prove that swapping two rows changes the sign of every permutation term, flipping the total determinant.

Determinants may look like algebraic puzzles, but the permutation formula reveals their true nature: a grand sum over all possible ways of matching rows to columns, with signs recording whether orientation is preserved or reversed.

59. Cramer's Rule

Cramer's Rule is a classical method for solving systems of linear equations using determinants. While rarely used in large-scale computation due to inefficiency, it offers deep theoretical insights into the relationship between determinants, invertibility, and linear systems. It shows how the determinant of a matrix encodes not only volume scaling but also the exact solution to equations.

The Setup

Consider a system of n linear equations with n unknowns:

$$Ax = b,$$

where A is an invertible $n \times n$ matrix, x is the vector of unknowns, and b is the right-hand side vector.

Cramer's Rule states:

$$x_i = \frac{\det(A_i)}{\det(A)},$$

where A_i is the matrix A with its i-th column replaced by b.

Example: 2×2 Case

Solve:

$$\begin{cases} 2x + y = 5 \\ x + 3y = 7 \end{cases}$$

Matrix form:

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 7 \end{bmatrix}.$$

Determinant of A:

$$\det(A) = 2 \cdot 3 - 1 \cdot 1 = 5.$$

- For x_1: replace first column with b:

$$A_1 = \begin{bmatrix} 5 & 1 \\ 7 & 3 \end{bmatrix}, \quad \det(A_1) = 15 - 7 = 8.$$

So $x_1 = 8/5$.

- For x_2: replace second column with b:

$$A_2 = \begin{bmatrix} 2 & 5 \\ 1 & 7 \end{bmatrix}, \quad \det(A_2) = 14 - 5 = 9.$$

So $x_2 = 9/5$.

Solution: $(x, y) = (8/5, 9/5)$.

Why It Works

Since A is invertible,

$$x = A^{-1}b.$$

But recall the formula for the inverse:

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

where $\text{adj}(A)$ is the adjugate (transpose of the cofactor matrix). When we multiply $\text{adj}(A)b$, each component naturally becomes a determinant with one column replaced by b. This is exactly Cramer's Rule.

Geometric Interpretation

- The denominator $\det(A)$ represents the volume of the parallelotope spanned by the columns of A.
- The numerator $\det(A_i)$ represents the volume when the i-th column is replaced by b.
- The ratio tells how much of the volume contribution is aligned with the i-th direction, giving the solution coordinate.

Efficiency and Limitations

- Good for small n: Elegant for 2×2 or 3×3 systems.
- Inefficient for large n: Requires computing $n + 1$ determinants, each with factorial complexity if done by cofactor expansion.
- Numerical instability: Determinants can be sensitive to rounding errors.
- In practice, Gaussian elimination or LU decomposition is far superior.

Applications

1. Theoretical proofs: Establishes uniqueness of solutions for small systems.
2. Geometry: Connects solutions to ratios of volumes of parallelotopes.
3. Symbolic algebra: Useful for deriving closed-form expressions.
4. Control theory: Sometimes applied in proofs of controllability/observability.

Why It Matters

- Provides a clear formula linking determinants and solutions of linear systems.
- Demonstrates the power of determinants as more than just volume measures.
- Acts as a conceptual bridge between algebraic solutions and geometric interpretations.

Try It Yourself

1. Solve $\begin{cases} x + 2y = 3 \\ 4x + 5y = 6 \end{cases}$ using Cramer's Rule.
2. For the 3×3 system with matrix $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 5 & 6 & 0 \end{bmatrix}$, compute x_1 using Cramer's Rule.
3. Verify that when $\det(A) = 0$, Cramer's Rule breaks down, matching the fact that the system is either inconsistent or has infinitely many solutions.
4. Challenge: Derive Cramer's Rule from the adjugate matrix formula.

Cramer's Rule is not a computational workhorse, but it elegantly ties together determinants, invertibility, and the solution of linear systems-showing how geometry, algebra, and computation meet in one neat formula.

60. Computing Determinants in Practice

Determinants carry deep meaning, but when it comes to actual computation, the method you choose makes all the difference. For small matrices, formulas like cofactor expansion or Cramer's Rule are manageable. For larger systems, however, these direct approaches quickly become inefficient. Practical computation relies on systematic algorithms that exploit structure-especially elimination and matrix factorizations.

Small Matrices (n 3)

- 2×2 case:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc.$$

- 3×3 case: Either expand by cofactors or use the "rule of Sarrus":

$$\det \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = aei + bfg + cdh - ceg - bdi - afh.$$

These formulas are compact, but do not generalize well beyond 3×3.

Large Matrices: Elimination and LU Decomposition

For $n > 3$, practical methods revolve around Gaussian elimination.

1. Row Reduction:
   - Reduce A to an upper triangular matrix U using row operations.
   - Keep track of operations:
     - Row swaps $\rightarrow$ flip sign of determinant.
     - Row scaling $\rightarrow$ multiply determinant by the scaling factor.
     - Row replacements $\rightarrow$ no effect.
   - Once triangular, compute determinant as the product of diagonal entries.
2. LU Factorization:
   - Express $A = LU$, where L is lower triangular and U is upper triangular.
   - Then $\det(A) = \det(L)\det(U)$.
   - Since L has 1s on its diagonal, $\det(L) = 1$, so the determinant is just the product of diagonals of U.

This approach reduces the complexity to $O(n^3)$, far more efficient than the factorial growth of cofactor expansion.

Numerical Considerations

- Floating-Point Stability: Determinants can be very large or very small, leading to overflow or underflow in computers.
- Pivoting: In practice, partial pivoting ensures stability during elimination.
- Condition Number: If a matrix is nearly singular ($\det(A)$ close to 0), computed determinants may be highly inaccurate.

For these reasons, in numerical linear algebra, determinants are rarely computed directly; instead, properties of LU or QR factorizations are used.

Determinant via Eigenvalues

Since the determinant equals the product of eigenvalues,

$$\det(A) = \lambda_1 \lambda_2 \cdots \lambda_n,$$

it can be computed by finding eigenvalues (numerically via QR iteration or other methods). This is useful when eigenvalues are already needed, but computing them just for the determinant is often more expensive than elimination.

Special Matrices

- Diagonal or triangular matrices: Determinant is product of diagonals-fastest case.
- Block diagonal matrices: Determinant is the product of determinants of blocks.
- Sparse matrices: Exploit structure-only nonzero patterns matter.
- Orthogonal matrices: Determinant is always $+1$ or -1.

Applications

1. System solving: Determinants test invertibility, but actual solving uses elimination.
2. Computer graphics: Determinants detect orientation flips (useful for rendering).
3. Optimization: Determinants of Hessians signal curvature and stability.
4. Statistics: Determinants of covariance matrices measure uncertainty volumes.
5. Physics: Determinants appear in Jacobians for change of variables in integrals.

Why It Matters

- Determinants provide a global property of matrices, but computation must be efficient.
- Direct expansion is elegant but impractical.
- Elimination-based methods balance theory, speed, and reliability, forming the backbone of modern computational linear algebra.

Try It Yourself

1. Compute the determinant of $\begin{bmatrix} 2 & 1 & 3 \\ 4 & 1 & 7 \\ -2 & 5 & 1 \end{bmatrix}$ using elimination, confirming the diagonal product method.
2. For a diagonal matrix with entries $(2, 3, -1, 5)$, verify that the determinant is simply their product.

3. Use LU decomposition to compute the determinant of a 3×3 matrix of your choice.
4. Challenge: Show that determinant computation by LU requires only $O(n^3)$ operations, while cofactor expansion requires $O(n!)$.

Determinants are central, but in practice they are best approached with systematic algorithms, where triangular forms and factorizations reveal the answer quickly and reliably.

Closing

```
Flatness or fullness,
determinants quietly weigh
depth in every move.
```

Chapter 7. Eigenvalues, eigenvectors, and dynamics

Opening

```
Stillness in motion,
directions that never fade,
time reveals its core.
```

61. Eigenvalues and Eigenvectors

Among all the concepts in linear algebra, few are as central and powerful as eigenvalues and eigenvectors. They reveal the hidden "axes of action" of a linear transformation-directions in space where the transformation behaves in the simplest possible way. Instead of mixing and rotating everything, an eigenvector is left unchanged in direction, scaled only by its corresponding eigenvalue.

The Core Idea

Let A be an $n \times n$ matrix. A nonzero vector $v \in \mathbb{R}^n$ is called an eigenvector of A if

$$Av = \lambda v,$$

for some scalar $\lambda \in \mathbb{R}$ (or $\mathbb{C}$). The scalar λ is the eigenvalue corresponding to v.

- Eigenvector: A special direction that is preserved by the transformation.
- Eigenvalue: The factor by which the eigenvector is stretched or compressed.

If $\lambda > 1$, the eigenvector is stretched. If $0 < \lambda < 1$, it is compressed. If $\lambda < 0$, it is flipped in direction and scaled. If $\lambda = 0$, the vector is flattened to zero.

Why They Matter

Eigenvalues and eigenvectors describe the intrinsic structure of a transformation:

- They give preferred directions in which the action of the matrix is simplest.
- They summarize long-term behavior of repeated applications (e.g., powers of A).
- They connect algebra, geometry, and applications in physics, data science, and engineering.

Example: A Simple 2D Case

Let

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

- Applying A to $(1, 0)$:

$$A \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

  So $(1, 0)$ is an eigenvector with eigenvalue 2.

- Applying A to $(0, 1)$:

$$A \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \end{bmatrix} = 3 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

  So $(0, 1)$ is an eigenvector with eigenvalue 3.

Here the eigenvectors align with the coordinate axes, and the eigenvalues are the diagonal entries.

General Case: The Eigenvalue Equation

To find eigenvalues, we solve

$$Av = \lambda v \quad \Leftrightarrow \quad (A - \lambda I)v = 0.$$

For nontrivial v, the matrix $(A - \lambda I)$ must be singular:

$$\det(A - \lambda I) = 0.$$

This determinant expands to the characteristic polynomial, whose roots are the eigenvalues. Eigenvectors come from solving the corresponding null spaces.

Geometric Interpretation

- Eigenvectors are invariant directions. When you apply A, the vector may stretch or flip, but it does not rotate off its line.
- Eigenvalues are scaling factors. They describe how much stretching, shrinking, or flipping happens along that invariant direction.

For example:

- In 2D, an eigenvector might be a line through the origin where the transformation acts as a stretch.
- In 3D, planes of shear often have eigenvectors along axes of invariance.

Dynamics and Repeated Applications

One reason eigenvalues are so important is that they describe repeated transformations:

$$A^k v = \lambda^k v.$$

If you apply A repeatedly to an eigenvector, the result is predictable: just multiply by λ^k. This explains stability in dynamical systems, growth in population models, and convergence in Markov chains.

- If $|\lambda| < 1$, repeated applications shrink the vector to zero.
- If $|\lambda| > 1$, the vector grows without bound.
- If $\lambda = 1$, the vector stays the same length (though direction may flip if $\lambda = -1$).

Applications

1. Physics: Vibrations of molecules, quantum energy levels, and resonance all rely on eigenvalues/eigenvectors.
2. Data Science: Principal Component Analysis (PCA) finds eigenvectors of covariance matrices to detect key directions of variance.
3. Markov Chains: Steady-state probabilities correspond to eigenvectors with eigenvalue 1.
4. Differential Equations: Eigenvalues simplify systems of linear ODEs.
5. Computer Graphics: Transformations like rotations and scalings can be analyzed with eigen-decompositions.

Why It Matters

- Eigenvalues and eigenvectors reduce complex transformations to their simplest components.
- They unify algebra (roots of characteristic polynomials), geometry (invariant directions), and applications (stability, resonance, variance).
- They are the foundation for diagonalization, SVD, and spectral analysis, which dominate modern applied mathematics.

Try It Yourself

1. Compute the eigenvalues and eigenvectors of $\begin{bmatrix} 4 & 2 \\ 1 & 3 \end{bmatrix}$.
2. For $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$, find its eigenvalues. (Hint: they are complex.)
3. Take a random 2×2 matrix and check if its eigenvectors align with coordinate axes.
4. Challenge: Prove that eigenvectors corresponding to distinct eigenvalues are linearly independent.

Eigenvalues and eigenvectors are the "fingerprints" of a matrix: they capture the essential behavior of a transformation, guiding us to understand stability, dynamics, and structure across countless disciplines.

62. The Characteristic Polynomial

To uncover the eigenvalues of a matrix, we use a central tool: the characteristic polynomial. This polynomial encodes the relationship between a matrix and its eigenvalues. The roots of the polynomial are precisely the eigenvalues, making it the algebraic gateway to spectral analysis.

Definition

For a square matrix $A \in \mathbb{R}^{n \times n}$, the characteristic polynomial is defined as

$$p_A(\lambda) = \det(A - \lambda I).$$

- I is the identity matrix of the same size as A.
- The polynomial $p_A(\lambda)$ has degree n.
- The eigenvalues of A are exactly the roots of $p_A(\lambda)$.

Why This Works

The eigenvalue equation is

$$Av = \lambda v \quad \iff \quad (A - \lambda I)v = 0.$$

For nontrivial v, the matrix $A - \lambda I$ must be singular:

$$\det(A - \lambda I) = 0.$$

Thus, eigenvalues are precisely the scalars λ for which the determinant vanishes.

Example: 2×2 Case

Let

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

Compute:

$$p_A(\lambda) = \det \begin{bmatrix} 4 - \lambda & 2 \\ 1 & 3 - \lambda \end{bmatrix}.$$

Expanding:

$$p_A(\lambda) = (4 - \lambda)(3 - \lambda) - 2.$$

$$= \lambda^2 - 7\lambda + 10.$$

The roots are $\lambda = 5$ and $\lambda = 2$. These are the eigenvalues of A.

Example: 3×3 Case

For

$$B = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 4 \\ 0 & 4 & 9 \end{bmatrix},$$

$$p_B(\lambda) = \det \begin{bmatrix} 2-\lambda & 0 & 0 \\ 0 & 3-\lambda & 4 \\ 0 & 4 & 9-\lambda \end{bmatrix}.$$

Expand:

$$p_B(\lambda) = (2-\lambda)\big[(3-\lambda)(9-\lambda) - 16\big].$$

$$= (2-\lambda)(\lambda^2 - 12\lambda + 11).$$

Roots: $\lambda = 2, 1, 11$.

Properties of the Characteristic Polynomial

1. Degree: Always degree n.
2. Leading term: $(-1)^n\lambda^n$.
3. Constant term: $\det(A)$.
4. Coefficient of λ^{n-1}: $-\mathrm{tr}(A)$, where $\mathrm{tr}(A)$ is the trace (sum of diagonal entries).

So:

$$p_A(\lambda) = (-1)^n\lambda^n + (\mathrm{tr}(A))(-1)^{n-1}\lambda^{n-1} + \cdots + \det(A).$$

This ties together trace, determinant, and eigenvalues in one polynomial.

Geometric Meaning

- The roots of the characteristic polynomial tell us scaling factors along invariant directions.
- In 2D: the polynomial encodes area scaling ($\det(A)$) and total stretching ($\text{tr}(A)$).
- In higher dimensions: it condenses the complexity of A into a single equation whose solutions reveal the spectrum.

Applications

1. Eigenvalue computation: Foundation for diagonalization and spectral theory.
2. Control theory: Stability of systems depends on eigenvalues (roots of the characteristic polynomial).
3. Differential equations: Characteristic polynomials describe natural frequencies and modes of oscillation.
4. Graph theory: The characteristic polynomial of an adjacency matrix encodes structural properties of the graph.
5. Quantum mechanics: Energy levels of quantum systems come from solving characteristic polynomials of operators.

Why It Matters

- Provides a systematic, algebraic way to find eigenvalues.
- Connects trace and determinant to deeper spectral properties.
- Bridges linear algebra, polynomial theory, and geometry.
- Forms the foundation for modern computational methods like QR iteration.

Try It Yourself

1. Compute the characteristic polynomial of $\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$. Find its eigenvalues.
2. Verify that the product of eigenvalues equals the determinant.
3. Verify that the sum of eigenvalues equals the trace.
4. Challenge: Prove that $p_{AB}(\lambda) = p_{BA}(\lambda)$ for any A, B of the same size.

The characteristic polynomial distills a matrix into a single algebraic object whose roots reveal the essential dynamics of the transformation.

63. Algebraic vs. Geometric Multiplicity

When studying eigenvalues, it's not enough to just find the roots of the characteristic polynomial. Each eigenvalue can appear multiple times, and this "multiplicity" can be understood in two distinct but related ways: algebraic multiplicity (how many times it appears as a root) and geometric multiplicity (the dimension of its eigenspace). These two multiplicities capture both the algebraic and geometric richness of eigenvalues.

Algebraic Multiplicity

The algebraic multiplicity (AM) of an eigenvalue λ is the number of times it appears as a root of the characteristic polynomial $p_A(\lambda)$.

- If $(\lambda - \lambda_0)^k$ divides $p_A(\lambda)$, then the algebraic multiplicity of λ_0 is k.
- The sum of all algebraic multiplicities equals the size of the matrix (n).

Example: If

$$p_A(\lambda) = (\lambda - 2)^3(\lambda + 1)^2,$$

then eigenvalue $\lambda = 2$ has AM $= 3$, and $\lambda = -1$ has AM $= 2$.

Geometric Multiplicity

The geometric multiplicity (GM) of an eigenvalue λ is the dimension of the eigenspace corresponding to λ:

$$\text{GM}(\lambda) = \dim(\ker(A - \lambda I)).$$

- This counts how many linearly independent eigenvectors correspond to λ.
- Always satisfies:

$$1 \leq \text{GM}(\lambda) \leq \text{AM}(\lambda).$$

Example: If

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix},$$

then $p_A(\lambda) = (\lambda - 2)^2$.

- AM of $\lambda = 2$ is 2.
- Solve $(A - 2I)v = 0$:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} v = 0 \quad \Rightarrow \quad v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

  Only 1 independent eigenvector.
- GM of $\lambda = 2$ is 1.

Relationship Between the Two

- Always: $\text{GM}(\lambda) \leq \text{AM}(\lambda)$.
- If they are equal for all eigenvalues, the matrix is diagonalizable.
- If GM $<$ AM for some eigenvalue, the matrix is defective, meaning it cannot be diagonalized, though it may still have a Jordan canonical form.

Geometric Meaning

- AM measures how strongly the eigenvalue is "encoded" in the polynomial.
- GM measures how much geometric freedom the eigenvalue's eigenspace provides.
- If AM $>$ GM, the eigenvalue "wants" more independent directions than the space allows.

Think of AM as the *theoretical demand* for eigenvectors, and GM as the *actual supply*.

Example: Diagonalizable vs. Defective

1. Diagonalizable case:

$$B = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}.$$

   - $p_B(\lambda) = (\lambda - 2)^2$.
   - AM $= 2$ for eigenvalue 2.
   - GM $= 2$, since the eigenspace is all of $\mathbb{R}^2$.
   - Enough eigenvectors to diagonalize.

2. Defective case: The earlier example

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

had AM = 2, GM = 1.

- Not enough eigenvectors.
- Cannot be diagonalized.

Applications

1. Diagonalization: Only possible when GM = AM for all eigenvalues.
2. Jordan form: Defective matrices require Jordan blocks, governed by the gap between AM and GM.
3. Differential equations: The solution form depends on multiplicity; repeated eigenvalues with fewer eigenvectors require generalized solutions.
4. Stability analysis: Multiplicities reveal degeneracies in dynamical systems.
5. Quantum mechanics: Degeneracy of eigenvalues (AM vs. GM) encodes physical symmetry.

Why It Matters

- Multiplicities separate algebraic roots from geometric structure.
- They decide whether diagonalization is possible.
- They reveal hidden constraints in systems with repeated eigenvalues.
- They form the basis for advanced concepts like Jordan canonical form and generalized eigenvectors.

Try It Yourself

1. Find AM and GM for $\begin{bmatrix} 3 & 1 \\ 0 & 3 \end{bmatrix}$.
2. Find AM and GM for $\begin{bmatrix} 3 & 0 \\ 0 & 3 \end{bmatrix}$. Compare with the first case.
3. Show that AM always equals the multiplicity of a root of the characteristic polynomial.
4. Challenge: Prove that for any eigenvalue, GM 1.

Algebraic and geometric multiplicity together tell the full story: the algebra tells us how many times an eigenvalue appears, while the geometry tells us how much room it really occupies in the vector space.

64. Diagonalization

Diagonalization is one of the most powerful ideas in linear algebra. It takes a complicated matrix and, when possible, rewrites it in a simple form where its action is completely transparent. A diagonal matrix is easy to understand: it just stretches or compresses each coordinate axis by a fixed factor. If we can transform a matrix into a diagonal one, many calculations-like computing powers or exponentials-become almost trivial.

The Core Concept

A square matrix $A \in \mathbb{R}^{n\times n}$ is diagonalizable if there exists an invertible matrix P and a diagonal matrix D such that

$$A = PDP^{-1}.$$

- The diagonal entries of D are the eigenvalues of A.
- The columns of P are the corresponding eigenvectors.

In words: A can be "rewritten" in a coordinate system made of its eigenvectors, where its action reduces to simple scaling along independent directions.

Why Diagonalization Matters

1. Simplifies Computations:

   - Computing powers:

   $$A^k = PD^kP^{-1}, \quad D^k \text{ is trivial to compute.}$$

   - Matrix exponential:

   $$e^A = Pe^DP^{-1}.$$

   Critical in solving differential equations.

2. Clarifies Dynamics:

   - Long-term behavior of iterative processes depends directly on eigenvalues.
   - Stable vs. unstable systems can be read off from D.

3. Reveals Structure:

- Tells us whether the system can be understood through independent modes.
- Connects algebraic structure with geometry.

Conditions for Diagonalization

A matrix A is diagonalizable if and only if it has enough linearly independent eigenvectors to form a basis for $\mathbb{R}^n$.

- Equivalently: For each eigenvalue, geometric multiplicity = algebraic multiplicity.
- Distinct eigenvalues guarantee diagonalizability, since their eigenvectors are linearly independent.

Example: Diagonalizable Case

Let

$$A = \begin{bmatrix} 4 & 0 \\ 1 & 3 \end{bmatrix}.$$

- Characteristic polynomial:

  $$p_A(\lambda) = (4 - \lambda)(3 - \lambda).$$

  Eigenvalues: $\lambda_1 = 4, \lambda_2 = 3$.

- Eigenvectors:
  - For $\lambda = 4$: $(1, 1)^T$.
  - For $\lambda = 3$: $(0, 1)^T$.
- Build $P = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, $D = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}$.
- Then $A = PDP^{-1}$.

Now, computing A^{10} is easy: just compute D^{10} and conjugate.

Example: Defective (Non-Diagonalizable) Case

$$B = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

- Characteristic polynomial: $(\lambda - 2)^2$.
- AM of eigenvalue 2 is 2, but GM = 1 (only one eigenvector).
- Not diagonalizable. Needs Jordan form instead.

Geometric Meaning

Diagonalization means we can rotate into a basis of eigenvectors where the transformation acts simply: scale each axis by its eigenvalue.

- Think of a room where the floor stretches more in one direction than another. In the right coordinate system (aligned with eigenvectors), the stretch is purely along axes.
- Without diagonalization, stretching mixes directions and is harder to describe.

Applications

1. Differential Equations: Solving systems of linear ODEs relies on diagonalization or Jordan form.
2. Markov Chains: Transition matrices are analyzed through diagonalization to study steady states.
3. Quantum Mechanics: Operators are diagonalized to reveal measurable states.
4. PCA (Principal Component Analysis): A covariance matrix is diagonalized to extract independent variance directions.
5. Computer Graphics: Diagonalization simplifies rotation-scaling transformations.

Why It Matters

Diagonalization transforms complexity into simplicity. It exposes the fundamental action of a matrix: scaling along preferred axes. Without it, understanding or computing repeated transformations would be intractable.

Try It Yourself

1. Diagonalize

$$C = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}.$$

   Compute C^5 using PD^5P^{-1}.

2. Show why

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

   cannot be diagonalized.

3. Challenge: Prove that any symmetric real matrix is diagonalizable with an orthogonal basis.

Diagonalization is like finding the natural "language" of a matrix: once we listen in its native basis, everything becomes clear, elegant, and simple.

65. Powers of a Matrix

Once we know about diagonalization, one of its most powerful consequences is the ability to compute powers of a matrix efficiently. Normally, multiplying a matrix by itself repeatedly is expensive and messy. But if a matrix can be diagonalized, its powers become almost trivial to calculate. This is crucial in understanding long-term behavior of dynamical systems, Markov chains, and iterative algorithms.

The General Principle

If a matrix A is diagonalizable, then

$$A = PDP^{-1},$$

where D is diagonal and P is invertible.

Then for any positive integer k:

$$A^k = (PDP^{-1})^k = PD^kP^{-1}.$$

Because $P^{-1}P = I$, the middle terms cancel out in the product.

- Computing D^k is simple: just raise each diagonal entry to the k-th power.
- Thus, eigenvalues control the growth or decay of powers of the matrix.

Example: A Simple Diagonal Case

Let

$$D = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

Then

$$D^k = \begin{bmatrix} 2^k & 0 \\ 0 & 3^k \end{bmatrix}.$$

Each eigenvalue is raised independently to the k-th power.

Example: Using Diagonalization

Consider

$$A = \begin{bmatrix} 4 & 0 \\ 1 & 3 \end{bmatrix}.$$

From before, we know it diagonalizes as

$$A = PDP^{-1}, \quad D = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}.$$

So,

$$A^k = P \begin{bmatrix} 4^k & 0 \\ 0 & 3^k \end{bmatrix} P^{-1}.$$

Instead of multiplying A by itself k times, we just exponentiate the eigenvalues.

Long-Term Behavior

Eigenvalues reveal exactly what happens as $k \to \infty$.

- If all eigenvalues satisfy $|\lambda| < 1$, then $A^k \to 0$.
- If some eigenvalues have $|\lambda| > 1$, then A^k diverges along those eigenvector directions.
- If $|\lambda| = 1$, the behavior depends on the specific structure: it may oscillate, stabilize, or remain bounded.

This explains stability in recursive systems and iterative algorithms.

Special Case: Markov Chains

In probability, the transition matrix of a Markov chain has eigenvalues less than or equal to 1.

- The largest eigenvalue is always $\lambda = 1$.
- As powers of the transition matrix grow, the chain converges to the eigenvector associated with $\lambda = 1$, representing the stationary distribution.

Thus, A^k describes the long-run behavior of the chain.

Non-Diagonalizable Matrices

If a matrix is not diagonalizable, things become more complicated. Such matrices require the Jordan canonical form, where blocks can lead to terms like $k\lambda^{k-1}$.

Example:

$$B = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

Then

$$B^k = \begin{bmatrix} 2^k & k2^{k-1} \\ 0 & 2^k \end{bmatrix}.$$

The presence of the off-diagonal entry introduces linear growth in k, in addition to exponential scaling.

Geometric Meaning

- Powers of A correspond to repeated application of the linear transformation.
- Eigenvalues dictate whether directions expand, shrink, or remain steady.
- The eigenvectors mark the axes along which the repeated action is simplest to describe.

Think of stretching a rubber sheet: after each stretch, the sheet aligns more and more strongly with the dominant eigenvector.

Applications

1. Dynamical Systems: Population models, economic growth, and iterative algorithms all rely on powers of a matrix.
2. Markov Chains: Powers reveal equilibrium behavior and mixing rates.
3. Differential Equations: Discrete-time models use matrix powers to describe state evolution.
4. Computer Graphics: Repeated transformations can be analyzed via eigenvalues.
5. Machine Learning: Convergence of iterative solvers (like gradient descent with linear updates) depends on spectral radius.

Why It Matters

Matrix powers are the foundation of stability analysis, asymptotic behavior, and convergence. Diagonalization turns this from a brute-force multiplication into a deep, structured understanding.

Try It Yourself

1. Compute A^5 for $\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$.
2. For $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, compute A^k. What happens as $k \to \infty$?
3. Explore what happens to A^k when the largest eigenvalue has absolute value < 1, $= 1$, and > 1.
4. Challenge: Show that if a diagonalizable matrix has eigenvalues $|\lambda_i| < 1$, then $\lim_{k\to\infty} A^k = 0$.

Powers of a matrix reveal the story of repetition: how a transformation evolves when applied again and again. They connect linear algebra to time, growth, and stability in every system that unfolds step by step.

66. Real vs. Complex Spectra

Not all eigenvalues are real numbers. Even when working with real matrices, eigenvalues can emerge as complex numbers. Understanding when eigenvalues are real, when they are complex, and what this means geometrically is critical for grasping the full behavior of linear transformations.

Eigenvalues Over the Complex Numbers

Every square matrix $A \in \mathbb{R}^{n\times n}$ has at least one eigenvalue in the complex numbers. This is guaranteed by the Fundamental Theorem of Algebra, which says every polynomial (like the characteristic polynomial) has roots in $\mathbb{C}$.

- If $p_A(\lambda)$ has only real roots, all eigenvalues are real.
- If $p_A(\lambda)$ has quadratic factors with no real roots, then eigenvalues appear as complex conjugate pairs.

Why Complex Numbers Appear

Consider a 2D rotation matrix:

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

The characteristic polynomial is

$$p(\lambda) = \lambda^2 - 2\cos\theta\lambda + 1.$$

The eigenvalues are

$$\lambda = \cos\theta \pm i\sin\theta = e^{\pm i\theta}.$$

- Unless $\theta = 0, \pi$, these eigenvalues are not real.
- Geometrically, this makes sense: pure rotation has no invariant real direction. Instead, the eigenvalues are complex numbers of unit magnitude, encoding the rotation angle.

Real vs. Complex Scenarios

1. Symmetric Real Matrices:
   - All eigenvalues are real.
   - Eigenvectors form an orthogonal basis.
   - Example: $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ has eigenvalues $3, 1$.
2. General Real Matrices:
   - Eigenvalues may be complex.
   - If complex, they always come in conjugate pairs: if $\lambda = a + bi$, then $\overline{\lambda} = a - bi$ is also an eigenvalue.
3. Skew-Symmetric Matrices:
   - Purely imaginary eigenvalues.
   - Example: $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$ has eigenvalues $\pm i$.

Geometric Meaning of Complex Eigenvalues

- If eigenvalues are real, the transformation scales along real directions.
- If eigenvalues are complex, the transformation involves a combination of rotation and scaling.

For $\lambda = re^{i\theta}$:

- $r = |\lambda|$ controls expansion or contraction.
- θ controls rotation.

So a complex eigenvalue represents a spiral: stretching or shrinking while rotating.

Example: Spiral Dynamics

Matrix

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

rotates vectors by 90°.

- Eigenvalues: $\pm i$.

- Magnitude = 1, angle = $\pi/2$.
- Interpretation: every step is a rotation of 90°, with no scaling.

If we change to

$$B = \begin{bmatrix} 0.8 & -0.6 \\ 0.6 & 0.8 \end{bmatrix},$$

the eigenvalues are complex with modulus < 1.

- Interpretation: rotation combined with shrinking $\rightarrow$ spiraling toward the origin.

Applications

1. Differential Equations: Complex eigenvalues produce oscillatory solutions with sine and cosine terms.
2. Physics: Vibrations and wave phenomena rely on complex eigenvalues to model periodic behavior.
3. Control Systems: Stability requires checking magnitudes of eigenvalues in the complex plane.
4. Computer Graphics: Rotations and spiral motions are naturally described by complex spectra.
5. Signal Processing: Fourier transforms rely on complex eigenstructures of convolution operators.

Why It Matters

- Real eigenvalues describe pure stretching or compression.
- Complex eigenvalues describe combined rotation and scaling.
- Together, they provide a complete picture of matrix behavior in both real and complex spaces.
- Without considering complex eigenvalues, we miss entire classes of transformations, like rotation and oscillation.

Try It Yourself

1. Find eigenvalues of $\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$. Interpret geometrically.
2. For rotation by 45°, find eigenvalues of $\begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix}$. Show that they are $e^{\pm i\pi/4}$.

3. Check eigenvalues of $\begin{bmatrix} 2 & -5 \\ 1 & -2 \end{bmatrix}$. Are they real or complex?
4. Challenge: Prove that real polynomials of odd degree always have at least one real root. Connect this to eigenvalues of odd-dimensional real matrices.

Complex spectra extend our understanding of linear algebra into the full richness of oscillations, rotations, and spirals, where numbers alone are not enough-geometry and complex analysis merge to reveal the truth.

67. Defective Matrices and Jordan Form (a Glimpse)

Not every matrix can be simplified all the way into a diagonal form. Some matrices, while having repeated eigenvalues, do not have enough independent eigenvectors to span the entire space. These are called defective matrices. Understanding them requires introducing the Jordan canonical form, a generalization of diagonalization that handles these tricky cases.

Defective Matrices

A square matrix $A \in \mathbb{R}^{n \times n}$ is called defective if:

- It has an eigenvalue λ with algebraic multiplicity (AM) strictly larger than its geometric multiplicity (GM).
- Equivalently, A does not have enough linearly independent eigenvectors to form a full basis of $\mathbb{R}^n$.

Example:

$$A = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

- Characteristic polynomial: $(\lambda - 2)^2$, so AM $= 2$.
- Solving $(A - 2I)v = 0$:

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} v = 0 \quad \Rightarrow \quad v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

  Only one independent eigenvector $\rightarrow$ GM $= 1$.
- Since GM $<$ AM, this matrix is defective.

Defective matrices cannot be diagonalized.

Why Defective Matrices Exist

Diagonalization requires one independent eigenvector per eigenvalue copy. But sometimes the matrix "collapses" those directions together, producing fewer eigenvectors than expected.

- Think of it like having multiple musical notes written in the score (AM), but fewer instruments available to play them (GM).
- The matrix "wants" more independent directions, but the geometry of its null spaces prevents that.

Jordan Canonical Form (Intuition)

While defective matrices cannot be diagonalized, they can still be put into a nearly diagonal form called the Jordan canonical form (JCF):

$$J = P^{-1}AP,$$

where J consists of Jordan blocks:

$$J_k(\lambda) = \begin{bmatrix} \lambda & 1 & 0 & \cdots & 0 \\ 0 & \lambda & 1 & \cdots & 0 \\ 0 & 0 & \lambda & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 1 \\ 0 & 0 & 0 & \cdots & \lambda \end{bmatrix}.$$

Each block corresponds to one eigenvalue λ, with 1s on the superdiagonal indicating the lack of independent eigenvectors.

- If every block is 1×1, the matrix is diagonalizable.
- If larger blocks appear, the matrix is defective.

Example: Jordan Block of Size 2

The earlier defective example

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

has Jordan form

$$J = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

Notice it is already in Jordan form: one block of size 2 for eigenvalue 2.

Powers of Jordan Blocks

A key property is how powers behave. For

$$J = \begin{bmatrix} \lambda & 1 \\ 0 & \lambda \end{bmatrix},$$

$$J^k = \begin{bmatrix} \lambda^k & k\lambda^{k-1} \\ 0 & \lambda^k \end{bmatrix}.$$

- Unlike diagonal matrices, extra polynomial terms in k appear.
- This explains why defective matrices produce behavior like growth proportional to $k\lambda^{k-1}$.

Geometric Meaning

- Eigenvectors describe invariant lines.
- When there aren't enough eigenvectors, Jordan form encodes chains of generalized eigenvectors.
- Each chain captures how the matrix transforms vectors slightly off the invariant line, nudging them along directions linked together by the Jordan block.

So while a diagonalizable matrix decomposes space into neat independent directions, a defective matrix entangles some directions together, forcing them into chains.

Applications

1. Differential Equations: Jordan blocks determine the appearance of extra polynomial factors (like $te^{\lambda t}$) in solutions.
2. Markov Chains: Non-diagonalizable transition matrices produce slower convergence to steady states.
3. Numerical Analysis: Algorithms may fail or slow down if the system matrix is defective.
4. Control Theory: Stability depends not just on eigenvalues, but on whether the matrix is diagonalizable.
5. Quantum Mechanics: Degenerate eigenvalues require Jordan analysis to fully describe states.

Why It Matters

- Diagonalization is not always possible, and defective matrices are the exceptions.
- Jordan form is the universal fallback: every square matrix has one, and it generalizes diagonalization.
- It introduces generalized eigenvectors, which extend the reach of spectral theory.

Try It Yourself

1. Verify that $\begin{bmatrix} 3 & 1 \\ 0 & 3 \end{bmatrix}$ is defective. Find its Jordan form.
2. Show that for a Jordan block of size 3,

$$J^k = \lambda^k I + k\lambda^{k-1} N + \frac{k(k-1)}{2}\lambda^{k-2} N^2,$$

   where N is the nilpotent part (matrix with 1s above diagonal).
3. Compare the behavior of A^k for a diagonalizable vs. a defective matrix with the same eigenvalues.
4. Challenge: Prove that every square matrix has a Jordan form over the complex numbers.

Defective matrices and Jordan form show us that even when eigenvectors are "insufficient," we can still impose structure, capturing how linear transformations behave in their most fundamental building blocks.

68. Stability and Spectral Radius

When a matrix is applied repeatedly-through iteration, recursion, or dynamical systems-its long-term behavior is governed not by individual entries, but by its eigenvalues. The key measure here is the spectral radius, which tells us whether repeated applications lead to convergence, oscillation, or divergence.

The Spectral Radius

The spectral radius of a matrix A is defined as

$$\rho(A) = \max\{|\lambda| : \lambda \text{ is an eigenvalue of } A\}.$$

- It is the largest absolute value among all eigenvalues.

- If $|\lambda| > 1$, the eigenvalue leads to exponential growth along its eigenvector.
- If $|\lambda| < 1$, it leads to exponential decay.
- If $|\lambda| = 1$, behavior depends on whether the eigenvalue is simple or defective.

Stability in Iterative Systems

Consider a recursive process:

$$x_{k+1} = Ax_k.$$

- If $\rho(A) < 1$, then $A^k \to 0$ as $k \to \infty$. All trajectories converge to the origin.
- If $\rho(A) > 1$, then A^k grows without bound along the dominant eigenvector.
- If $\rho(A) = 1$, trajectories neither vanish nor diverge but may oscillate or stagnate.

Example: Convergence with Small Spectral Radius

$$A = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.8 \end{bmatrix}.$$

- Eigenvalues: $0.5, 0.8$.
- $\rho(A) = 0.8 < 1$.
- Powers A^k shrink vectors to zero $\to$ stable system.

Example: Divergence with Large Spectral Radius

$$B = \begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix}.$$

- Eigenvalues: $2, 0.5$.
- $\rho(B) = 2 > 1$.
- Powers B^k explode along the eigenvector $(1, 0)$.

Example: Oscillation with Complex Eigenvalues

$$C = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}.$$

- Eigenvalues: $\pm i$, both with modulus 1.
- $\rho(C) = 1$.
- System is neutrally stable: vectors rotate forever without shrinking or growing.

Beyond Simple Stability: Defective Cases

If a matrix has eigenvalues with $|\lambda| = 1$ and is defective, extra polynomial terms in k appear in A^k, leading to slow divergence even though $\rho(A) = 1$.

Example:

$$D = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

- Eigenvalue: $\lambda = 1$ (AM=2, GM=1).
- $\rho(D) = 1$.
- Powers grow linearly with k:

$$D^k = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}.$$

- System is unstable, despite spectral radius equal to 1.

Geometric Meaning

The spectral radius measures the dominant mode of a transformation:

- Imagine stretching and rotating a rubber sheet. After many repetitions, the sheet aligns with the direction corresponding to the largest eigenvalue.
- If the stretching is less than 1, everything shrinks.
- If greater than 1, everything expands.
- If exactly 1, the system is balanced on the edge of stability.

Applications

1. Numerical Methods: Convergence of iterative solvers (e.g., Jacobi, Gauss–Seidel) depends on spectral radius < 1.
2. Markov Chains: Long-term distributions exist if the largest eigenvalue $= 1$ and others $<$ 1 in magnitude.
3. Control Theory: System stability is judged by eigenvalues inside the unit circle ($|\lambda| < 1$).
4. Economics: Input-output models remain bounded only if spectral radius < 1.
5. Epidemiology: Basic reproduction number R_0 is essentially the spectral radius of a next-generation matrix.

Why It Matters

- The spectral radius condenses the entire spectrum of a matrix into a single stability criterion.
- It predicts the fate of iterative processes, from financial growth to disease spread.
- It draws a sharp boundary between decay, balance, and explosion in linear systems.

Try It Yourself

1. Compute the spectral radius of $\begin{bmatrix} 0.6 & 0.3 \\ 0.1 & 0.8 \end{bmatrix}$. Does the system converge?
2. Show that for any matrix norm $\|\cdot\|$,

$$\rho(A) \leq \|A\|.$$

   (Hint: use Gelfand's formula.)

3. For $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, explain why it diverges even though $\rho = 1$.
4. Challenge: Prove Gelfand's formula:

$$\rho(A) = \lim_{k \to \infty} \|A^k\|^{1/k}.$$

The spectral radius is the compass of linear dynamics: it points to stability, oscillation, or divergence, guiding us across disciplines wherever repeated transformations shape the future.

69. Markov Chains and Steady States

Markov chains are one of the most direct and beautiful applications of eigenvalues in probability and statistics. They describe systems that evolve step by step, where the next state depends only on the current one, not on the past. The mathematics of steady states-the long-term behavior of such chains-rests firmly on eigenvalues and eigenvectors of the transition matrix.

Transition Matrices

A Markov chain is defined by a transition matrix $P \in \mathbb{R}^{n \times n}$ with the following properties:

1. All entries are nonnegative: $p_{ij} \geq 0$.
2. Each row sums to 1: $\sum_j p_{ij} = 1$.

If the chain is in state i at time k, then p_{ij} is the probability of moving to state j at time $k+1$.

Evolution of States

If the probability distribution at time k is a row vector $\pi^{(k)}$, then

$$\pi^{(k+1)} = \pi^{(k)} P.$$

After k steps:

$$\pi^{(k)} = \pi^{(0)} P^k.$$

So understanding the long-term behavior requires analyzing P^k.

Eigenvalue Structure of Transition Matrices

- Every transition matrix P has eigenvalue $\lambda = 1$.
- All other eigenvalues satisfy $|\lambda| \leq 1$.
- If the chain is irreducible (all states communicate) and aperiodic (no cyclic locking), then:
  - $\lambda = 1$ is a simple eigenvalue (AM=GM=1).
  - All other eigenvalues have magnitude strictly less than 1.

This ensures convergence to a unique steady state.

Steady States as Eigenvectors

A steady state distribution π satisfies:

$$\pi = \pi P.$$

This is equivalent to:

$$\pi^T \text{ is a right eigenvector of } P^T \text{ with eigenvalue 1.}$$

- The steady state vector lies in the eigenspace of eigenvalue 1.
- Since probabilities must sum to 1, normalization gives a unique steady state.

Example: A 2-State Markov Chain

$$P = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}.$$

- Eigenvalues: solve $\det(P - \lambda I) = 0$.

$$\lambda_1 = 1, \quad \lambda_2 = 0.3.$$

- The steady state is found from $\pi = \pi P$:

$$\pi = \left(\frac{4}{7}, \frac{3}{7}\right).$$

- As $k \to \infty$, any initial distribution $\pi^{(0)}$ converges to this steady state.

Example: Random Walk on a Graph

Take a simple graph: 3 nodes in a line, where each node passes to neighbors equally.

Transition matrix:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \end{bmatrix}.$$

- Eigenvalues: $\{1, 0, -1\}$.
- The steady state corresponds to eigenvalue 1.
- After many steps, the distribution converges to $(0.25, 0.5, 0.25)$.

Geometric Meaning

- Eigenvalue 1: the fixed "direction" of probabilities that does not change under transitions.
- Eigenvalues < 1 in magnitude: transient modes that vanish as $k \to \infty$.
- The dominant eigenvector (steady state) is like the "center of gravity" of the system.

So powers of P filter out all but the eigenvector of eigenvalue 1.

Applications

1. Google PageRank: Steady state eigenvectors rank webpages.
2. Economics: Input-output models evolve like Markov chains.
3. Epidemiology: Spread of diseases can be modeled as Markov processes.
4. Machine Learning: Hidden Markov models (HMMs) underpin speech recognition and bioinformatics.
5. Queuing Theory: Customer arrivals and service evolve according to Markov dynamics.

Why It Matters

- The concept of steady states shows how randomness can lead to predictability.
- Eigenvalues explain why convergence happens, and at what rate.
- The link between linear algebra and probability provides one of the clearest real-world uses of eigenvectors.

Try It Yourself

1. For

$$P = \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix},$$

   compute its eigenvalues and steady state.

2. Show that for any transition matrix, the largest eigenvalue is always 1.
3. Prove that if a chain is irreducible and aperiodic, the steady state is unique.
4. Challenge: Construct a 3-state transition matrix with a cycle (periodic) and show why it doesn't converge to a steady distribution until perturbed.

Markov chains and steady states are the meeting point of probability and linear algebra: randomness, when multiplied many times, is tamed by the calm persistence of eigenvalue 1.

70. Linear Differential Systems

Many natural and engineered processes evolve continuously over time. When these processes can be expressed as linear relationships, they lead to systems of linear differential equations. The analysis of such systems relies almost entirely on eigenvalues and eigenvectors, which determine the behavior of solutions: whether they oscillate, decay, grow, or stabilize.

The General Setup

Consider a system of first-order linear differential equations:

$$\frac{d}{dt}x(t) = Ax(t),$$

where:

- $x(t) \in \mathbb{R}^n$ is the state vector at time t.
- $A \in \mathbb{R}^{n \times n}$ is a constant coefficient matrix.

The task is to solve for $x(t)$, given an initial state $x(0)$.

The Matrix Exponential

The formal solution is:

$$x(t) = e^{At}x(0),$$

where e^{At} is the matrix exponential defined as:

$$e^{At} = I + At + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \cdots.$$

But how do we compute e^{At} in practice? The answer comes from diagonalization and Jordan form.

Case 1: Diagonalizable Matrices

If A is diagonalizable:

$$A = PDP^{-1}, \quad D = \text{diag}(\lambda_1, \dots, \lambda_n).$$

Then:

$$e^{At} = Pe^{Dt}P^{-1}, \quad e^{Dt} = \text{diag}(e^{\lambda_1 t}, \dots, e^{\lambda_n t}).$$

Thus the solution is:

$$x(t) = P \begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} P^{-1}x(0).$$

Each eigenvalue λ_i dictates the time behavior along its eigenvector direction.

Case 2: Non-Diagonalizable Matrices

If A is defective, use its Jordan form $J = P^{-1}AP$:

$$e^{At} = Pe^{Jt}P^{-1}.$$

For a Jordan block of size 2:

$$J = \begin{bmatrix} \lambda & 1 \\ 0 & \lambda \end{bmatrix}, \quad e^{Jt} = e^{\lambda t} \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}.$$

Polynomial terms in t appear, multiplying the exponential part. This explains why repeated eigenvalues with insufficient eigenvectors yield solutions with extra polynomial factors.

Real vs. Complex Eigenvalues

- Real eigenvalues: solutions grow or decay exponentially along eigenvector directions.
  - If $\lambda < 0$: exponential decay $\rightarrow$ stability.
  - If $\lambda > 0$: exponential growth $\rightarrow$ instability.
- Complex eigenvalues: $\lambda = a \pm bi$. Solutions involve oscillations:

$$e^{(a+bi)t} = e^{at}(\cos(bt) + i\sin(bt)).$$

  - If $a < 0$: decaying oscillations.
  - If $a > 0$: growing oscillations.
  - If $a = 0$: pure oscillations, neutrally stable.

Example 1: Real Eigenvalues

$$A = \begin{bmatrix} -2 & 0 \\ 0 & -3 \end{bmatrix}.$$

Eigenvalues: $-2, -3$. Solution:

$$x(t) = \begin{bmatrix} c_1 e^{-2t} \\ c_2 e^{-3t} \end{bmatrix}.$$

Both terms decay $\rightarrow$ stable equilibrium at the origin.

Example 2: Complex Eigenvalues

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

Eigenvalues: $\pm i$. Solution:

$$x(t) = c_1 \begin{bmatrix} \cos t \\ \sin t \end{bmatrix} + c_2 \begin{bmatrix} -\sin t \\ \cos t \end{bmatrix}.$$

Pure oscillation $\rightarrow$ circular motion around the origin.

Example 3: Mixed Stability

$$A = \begin{bmatrix} 1 & 0 \\ 0 & -2 \end{bmatrix}.$$

Eigenvalues: $1, -2$. Solution:

$$x(t) = \begin{bmatrix} c_1 e^{t} \\ c_2 e^{-2t} \end{bmatrix}.$$

One direction grows, one decays $\rightarrow$ unstable overall, since divergence in one direction dominates.

Geometric Meaning

- The eigenvectors form the "axes of flow" of the system.
- The eigenvalues determine whether the flow along those axes spirals, grows, or shrinks.
- The phase portrait of the system-trajectories in the plane-is shaped by this interplay.

For example:

- Negative eigenvalues $\rightarrow$ trajectories funnel into the origin.
- Positive eigenvalues $\rightarrow$ trajectories repel outward.
- Complex eigenvalues $\rightarrow$ spirals or circles.

Applications

1. Control theory: Stability analysis of systems requires eigenvalue placement in the left-half plane.
2. Physics: Vibrations, quantum oscillations, and decay processes all follow eigenvalue rules.
3. Biology: Population models evolve according to linear differential equations.
4. Economics: Linear models of markets converge or diverge depending on eigenvalues.
5. Neuroscience: Neural firing dynamics can be modeled as linear ODE systems.

Why It Matters

- Linear differential systems bridge linear algebra with real-world dynamics.
- Eigenvalues determine not just numbers, but behaviors over time: growth, decay, oscillation, or equilibrium.
- They provide the foundation for analyzing nonlinear systems, which are often studied by linearizing around equilibrium points.

Try It Yourself

1. Solve $\frac{dx}{dt} = \begin{bmatrix} -1 & 2 \\ -2 & -1 \end{bmatrix} x$. Interpret the solution.
2. For $A = \begin{bmatrix} 0 & -2 \\ 2 & 0 \end{bmatrix}$, compute eigenvalues and describe the motion.
3. Verify that $e^{At} = Pe^{Dt}P^{-1}$ works when A is diagonalizable.
4. Challenge: Show that if all eigenvalues of A have negative real parts, then $\lim_{t\to\infty} x(t) = 0$ for any initial condition.

Linear differential systems show how eigenvalues control the flow of time itself in models. They explain why some processes die out, others oscillate, and others grow without bound-providing the mathematical skeleton behind countless real-world phenomena.

Closing

```
Spectra guide the flow,
growth and decay intertwining,
future sings through roots.
```

Chapter 8. Orthogonality, least squars, and QR

Opening

```
Perpendiculars,
meeting without crossing paths,
balance in silence.
```

71. Inner Products Beyond Dot Product

The dot product is the first inner product most students encounter. In $\mathbb{R}^n$, it is defined as

$$\langle x, y \rangle = x \cdot y = \sum_{i=1}^{n} x_i y_i,$$

and it provides a way to measure length, angle, and orthogonality. But the dot product is just one special case of a much broader concept. Inner products generalize the dot product, extending its geometric intuition to more abstract vector spaces.

Definition of an Inner Product

An inner product on a real vector space V is a function

$$\langle \cdot, \cdot \rangle : V \times V \to \mathbb{R}$$

that satisfies the following axioms for all $x, y, z \in V$ and scalar $\alpha \in \mathbb{R}$:

1. Positivity: $\langle x, x \rangle \geq 0$, and $\langle x, x \rangle = 0 \iff x = 0$.
2. Symmetry: $\langle x, y \rangle = \langle y, x \rangle$.
3. Linearity in the first argument: $\langle \alpha x + y, z \rangle = \alpha \langle x, z \rangle + \langle y, z \rangle$.

In complex vector spaces, the symmetry condition changes to conjugate symmetry: $\langle x, y \rangle = \overline{\langle y, x \rangle}$.

Norms and Angles from Inner Products

Once an inner product is defined, we immediately get:

- Norm (length): $\|x\| = \sqrt{\langle x, x \rangle}$.
- Distance: $d(x, y) = \|x - y\|$.
- Angle between vectors: $\cos \theta = \frac{\langle x, y \rangle}{\|x\| \|y\|}$.

Thus, inner products generalize the familiar geometry of $\mathbb{R}^n$ to broader contexts.

Examples Beyond the Dot Product

1. Weighted Inner Product (in $\mathbb{R}^n$):

$$\langle x, y \rangle_W = x^T W y,$$

where W is a symmetric positive definite matrix.

- Here, lengths and angles depend on the weights encoded in W.
- Useful when some dimensions are more important than others (e.g., weighted least squares).

2. Function Spaces (continuous inner product): On $V = C[a, b]$, the space of continuous functions on $[a, b]$:

$$\langle f, g \rangle = \int_a^b f(t)g(t)\, dt.$$

   - Length: $\|f\| = \sqrt{\int_a^b f(t)^2 dt}$.
   - Orthogonality: f and g are orthogonal if their integral product is zero.
   - This inner product underpins Fourier series.

3. Complex Inner Product (in $\mathbb{C}^n$):

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i \overline{y_i}.$$

   - Conjugation ensures positivity.
   - Critical for quantum mechanics, where states are vectors in complex Hilbert spaces.

4. Polynomial Spaces: For polynomials on $[-1, 1]$:

$$\langle p, q \rangle = \int_{-1}^{1} p(x)q(x)\, dx.$$

   - Leads to orthogonal polynomials (Legendre, Chebyshev), fundamental in approximation theory.

Geometric Interpretation

- Inner products reshape geometry. Instead of measuring lengths and angles with the Euclidean metric, we measure them with the metric induced by the chosen inner product.
- Different inner products create different geometries on the same vector space.

Example: A weighted inner product distorts circles into ellipses, changing which vectors count as "orthogonal."

Applications

1. Signal Processing: Correlation between signals is an inner product. Orthogonality means two signals carry independent information.
2. Fourier Analysis: Fourier coefficients come from inner products with sine and cosine functions.
3. Machine Learning: Kernel methods generalize inner products to infinite-dimensional spaces.
4. Quantum Mechanics: Probabilities are squared magnitudes of complex inner products.
5. Optimization: Weighted least squares problems use weighted inner products.

Why It Matters

- Inner products generalize geometry to new contexts: weighted spaces, functions, polynomials, quantum states.
- They provide the foundation for defining orthogonality, projections, and orthonormal bases in spaces far beyond $\mathbb{R}^n$.
- They unify ideas across pure mathematics, physics, engineering, and computer science.

Try It Yourself

1. Show that the weighted inner product $\langle x, y\rangle_W = x^T W y$ satisfies the inner product axioms if W is positive definite.
2. Compute $\langle f, g\rangle = \int_0^\pi \sin(t)\cos(t)\,dt$. Are $f = \sin$ and $g = \cos$ orthogonal?
3. In $\mathbb{C}^2$, verify that $\langle (1, i), (i, 1)\rangle = 0$. What does this mean geometrically?
4. Challenge: Prove that every inner product induces a norm, and that different inner products can lead to different geometries on the same space.

The dot product is just the beginning. Inner products provide the language to extend geometry into weighted spaces, continuous functions, and infinite dimensions-transforming how we measure similarity, distance, and structure across mathematics and science.

72. Orthogonality and Orthonormal Bases

Orthogonality is one of the most powerful ideas in linear algebra. It generalizes the familiar concept of perpendicularity in Euclidean space to abstract vector spaces equipped with an inner product. When orthogonality is combined with normalization (making vectors have unit length), we obtain orthonormal bases, which simplify computations, clarify geometry, and underpin many algorithms.

Orthogonality

Two vectors $x, y \in V$ are orthogonal if

$$\langle x, y \rangle = 0.$$

- In $\mathbb{R}^2$ or $\mathbb{R}^3$, this means the vectors are perpendicular.
- In function spaces, it means the integral of their product is zero.
- In signal processing, it means the signals are independent and non-overlapping.

Orthogonality captures the idea of "no overlap" or "independence" under the geometry of the inner product.

Properties of Orthogonal Vectors

1. If $x \perp y$, then $\|x + y\|^2 = \|x\|^2 + \|y\|^2$ (Pythagoras' theorem generalized).
2. Orthogonality is symmetric: if $x \perp y$, then $y \perp x$.
3. Any set of mutually orthogonal nonzero vectors is automatically linearly independent.

This last property is critical: orthogonality guarantees independence.

Orthonormal Sets

An orthonormal set is a collection of vectors $\{u_1, \dots, u_k\}$ such that

$$\langle u_i, u_j \rangle = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

- Each vector has unit length.
- Distinct vectors are mutually orthogonal.

This structure makes computations with coordinates as simple as possible.

Orthonormal Bases

A basis $\{u_1, \dots, u_n\}$ for a vector space is orthonormal if it is orthonormal as a set.

- Any vector $x \in V$ can be written as

$$x = \sum_{i=1}^{n} \langle x, u_i \rangle u_i.$$

- The coefficients are just inner products, no need to solve systems of equations.

This is why orthonormal bases are the most convenient: they make representation and projection effortless.

Examples

1. Standard Basis in $\mathbb{R}^n$: $\{e_1, e_2, \dots, e_n\}$, where e_i has 1 in the i-th coordinate and 0 elsewhere.
    - Orthonormal under the standard dot product.
2. Fourier Basis: Functions $\{\sin(nx), \cos(nx)\}$ on $[0, 2\pi]$ are orthogonal under the inner product $\langle f, g \rangle = \int_0^{2\pi} f(x)g(x)dx$.
    - This basis decomposes signals into pure frequencies.
3. Polynomial Basis: Legendre polynomials $P_n(x)$ are orthogonal on $[-1, 1]$ with respect to $\langle f, g \rangle = \int_{-1}^{1} f(x)g(x)\,dx$.

Geometric Meaning

Orthogonality splits space into independent "directions."

- Orthonormal bases are like perfectly aligned coordinate axes.
- Any vector decomposes uniquely as a sum of independent contributions along these axes.
- Distances and angles are preserved, making the geometry transparent.

Applications

1. Signal Processing: Decompose signals into orthogonal frequency components.
2. Machine Learning: Principal components form an orthonormal basis capturing variance directions.
3. Numerical Methods: Orthonormal bases improve numerical stability.
4. Quantum Mechanics: States are orthogonal if they represent mutually exclusive outcomes.
5. Computer Graphics: Rotations are represented by orthogonal matrices with orthonormal columns.

Why It Matters

- Orthogonality provides independence; orthonormality provides normalization.
- Together they make computations, decompositions, and projections clean and efficient.
- They underlie Fourier analysis, principal component analysis, and countless modern algorithms.

Try It Yourself

1. Show that $\{(1,0,0),(0,1,0),(0,0,1)\}$ is an orthonormal basis of $\mathbb{R}^3$.
2. Check whether $\{(1,1,0),(1,-1,0),(0,0,1)\}$ is orthonormal under the dot product. If not, normalize it.
3. Compute the coefficients of $x = (3,4)$ in the basis $\{(1,0),(0,1)\}$ and in the rotated orthonormal basis $\{(1/\sqrt{2},1/\sqrt{2}),(-1/\sqrt{2},1/\sqrt{2})\}$.
4. Challenge: Prove that in any finite-dimensional inner product space, an orthonormal basis always exists (hint: Gram–Schmidt).

Orthogonality and orthonormal bases are the backbone of linear algebra: they transform messy problems into elegant decompositions, giving us the cleanest possible language for describing vectors, signals, and data.

73. Gram–Schmidt Process

The Gram–Schmidt process is a systematic method for turning any linearly independent set of vectors into an orthonormal basis. This process is one of the most elegant bridges between algebra and geometry: it takes arbitrary vectors and makes them mutually perpendicular, while preserving the span.

The Problem It Solves

Given a set of linearly independent vectors $\{v_1, v_2, \dots, v_n\}$ in an inner product space:

- They span some subspace W.
- But they are not necessarily orthogonal or normalized.

Goal: Construct an orthonormal basis $\{u_1, u_2, \dots, u_n\}$ for W.

The Gram–Schmidt Algorithm

1. Start with the first vector:

$$u_1 = \frac{v_1}{\|v_1\|}.$$

2. For the second vector, subtract the projection onto u_1:

$$w_2 = v_2 - \langle v_2, u_1 \rangle u_1, \quad u_2 = \frac{w_2}{\|w_2\|}.$$

3. For the third vector, subtract projections onto both u_1 and u_2:

$$w_3 = v_3 - \langle v_3, u_1 \rangle u_1 - \langle v_3, u_2 \rangle u_2, \quad u_3 = \frac{w_3}{\|w_3\|}.$$

4. Continue inductively:

$$w_k = v_k - \sum_{j=1}^{k-1} \langle v_k, u_j \rangle u_j, \quad u_k = \frac{w_k}{\|w_k\|}.$$

At each step, w_k is made orthogonal to all previous u_j, and then normalized to form u_k.

Example in $\mathbb{R}^2$

Start with $v_1 = (1, 1)$, $v_2 = (1, 0)$.

1. Normalize first vector:

$$u_1 = \frac{(1,1)}{\sqrt{2}} = \left(\tfrac{1}{\sqrt{2}}, \tfrac{1}{\sqrt{2}}\right).$$

2. Subtract projection of v_2 on u_1:

$$w_2 = (1,0) - \left(\tfrac{1}{\sqrt{2}} \cdot 1 + \tfrac{1}{\sqrt{2}} \cdot 0\right)\left(\tfrac{1}{\sqrt{2}}, \tfrac{1}{\sqrt{2}}\right).$$

$$= (1,0) - \tfrac{1}{\sqrt{2}}\left(\tfrac{1}{\sqrt{2}}, \tfrac{1}{\sqrt{2}}\right).$$

$$= (1,0) - (0.5, 0.5) = (0.5, -0.5).$$

3. Normalize:

$$u_2 = \frac{(0.5,-0.5)}{\sqrt{0.5^2 + (-0.5)^2}} = \frac{(0.5,-0.5)}{\sqrt{0.5}} = \left(\tfrac{1}{\sqrt{2}}, -\tfrac{1}{\sqrt{2}}\right).$$

Final orthonormal basis:

$$u_1 = \left(\tfrac{1}{\sqrt{2}}, \tfrac{1}{\sqrt{2}}\right), \quad u_2 = \left(\tfrac{1}{\sqrt{2}}, -\tfrac{1}{\sqrt{2}}\right).$$

Geometric Intuition

- Each step removes "overlap" with previously chosen directions.
- Think of it as building new perpendicular coordinate axes inside the span of the original vectors.
- The result is like rotating and scaling the original set into a perfectly orthogonal system.

Numerical Stability

- Classical Gram–Schmidt can suffer from round-off errors in computer calculations.
- A numerically stable alternative is Modified Gram–Schmidt (MGS), which reorders the projection steps to reduce loss of orthogonality.
- In practice, QR factorization algorithms often implement MGS or Householder reflections.

Applications

1. QR Factorization: Gram–Schmidt provides the foundation: $A = QR$, where Q is orthogonal and R is upper triangular.
2. Data Compression: Orthonormal bases from Gram–Schmidt lead to efficient representations.
3. Signal Processing: Ensures independent frequency or wave components.
4. Machine Learning: Used in orthogonalization of features and dimensionality reduction.
5. Physics: Orthogonal states in quantum mechanics can be constructed from arbitrary states using Gram–Schmidt.

Why It Matters

- Gram–Schmidt guarantees that any independent set can be reshaped into an orthonormal basis.
- It underlies computational methods like QR decomposition, least squares, and numerical PDE solvers.
- It makes projections, coordinates, and orthogonality explicit and manageable.

Try It Yourself

1. Apply Gram–Schmidt to $(1, 0, 1)$, $(1, 1, 0)$, $(0, 1, 1)$ in $\mathbb{R}^3$. Verify orthonormality.
2. Show that the span of the orthonormal basis equals the span of the original vectors.
3. Use Gram–Schmidt to find an orthonormal basis for polynomials $\{1, x, x^2\}$ on $[-1, 1]$ with inner product $\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\,dx$.
4. Challenge: Prove that Gram–Schmidt always works for linearly independent sets, but fails if the set is dependent.

The Gram–Schmidt process is the algorithmic heart of orthogonality: it takes the messy and redundant and reshapes it into clean, perpendicular building blocks for the spaces we study.

74. Projections onto Subspaces

Projections are a natural extension of orthogonality: they describe how to "drop" a vector onto a subspace in the most natural way, minimizing the distance. Understanding projections is crucial for solving least squares problems, decomposing vectors, and interpreting data in terms of simpler, lower-dimensional structures.

Projection onto a Vector

Start with the simplest case: projecting a vector x onto a nonzero vector u.

1. The projection is the component of x that lies in the direction of u.
2. Formula:

$$\text{proj}_u(x) = \frac{\langle x, u \rangle}{\langle u, u \rangle} u.$$

   If u is normalized ($\|u\| = 1$), this simplifies to

$$\text{proj}_u(x) = \langle x, u \rangle u.$$

Geometrically, this is the foot of the perpendicular from x onto the line spanned by u.

Projection onto an Orthonormal Basis

Suppose we have an orthonormal basis $\{u_1, u_2, \dots, u_k\}$ for a subspace W. Then the projection of x onto W is:

$$\text{proj}_W(x) = \sum_{i=1}^{k} \langle x, u_i \rangle u_i.$$

This formula is powerful:

- Each coefficient $\langle x, u_i \rangle$ captures how much of x aligns with u_i.
- The sum reconstructs the best approximation of x inside W.

Projection Matrix

When working in coordinates, projections can be represented by matrices.

- If U is the $n \times k$ matrix with orthonormal columns $\{u_1, \dots, u_k\}$, then

$$P = UU^T$$

  is the projection matrix onto W.

Properties of P:

1. Idempotence: $P^2 = P$.
2. Symmetry: $P^T = P$.
3. Best approximation: For any x, $\|x - Px\|$ is minimized.

Projection and Orthogonal Complements

If W is a subspace of V, then every vector $x \in V$ can be decomposed uniquely as

$$x = \text{proj}_W(x) + \text{proj}_{W^\perp}(x),$$

where $W^\perp$ is the orthogonal complement of W.

This decomposition is the orthogonal decomposition theorem. It says: space splits cleanly into "in" and "out of" components relative to a subspace.

Example in $\mathbb{R}^2$

Let $u = (2, 1)$, and project $x = (3, 4)$ onto $\text{span}\{u\}$.

1. Compute inner product: $\langle x, u \rangle = 3 \cdot 2 + 4 \cdot 1 = 10$.
2. Compute norm squared: $\langle u, u \rangle = 2^2 + 1^2 = 5$.
3. Projection:

$$\text{proj}_u(x) = \frac{10}{5}(2, 1) = 2(2, 1) = (4, 2).$$

4. Orthogonal error:

$$x - \text{proj}_u(x) = (3, 4) - (4, 2) = (-1, 2).$$

Notice: $(4, 2)$ lies on the line through u, and the error vector $(-1, 2)$ is orthogonal to u.

Applications

1. Least Squares Regression: The regression line is the projection of data onto the subspace spanned by predictor variables.
2. Dimensionality Reduction: Principal Component Analysis (PCA) projects data onto the subspace of top eigenvectors.
3. Computer Graphics: 3D objects are projected onto 2D screens.
4. Numerical Methods: Projections solve equations approximately when exact solutions don't exist.
5. Physics: Work and energy are computed via projections of forces and velocities.

Why It Matters

- Projections are the essence of approximation: they give the "best possible" version of a vector inside a chosen subspace.
- They formalize independence: the error vector is always orthogonal to the subspace.
- They provide geometric intuition for statistics, machine learning, and numerical computation.

Try It Yourself

1. Compute the projection of $x = (2, 3, 4)$ onto $u = (1, 1, 1)$.
2. Verify that the residual $x - \text{proj}_u(x)$ is orthogonal to u.
3. Write the projection matrix for the subspace spanned by $\{(1, 0, 0), (0, 1, 0)\}$ in $\mathbb{R}^3$.
4. Challenge: Prove that projection matrices are idempotent and symmetric.

Projections turn vector spaces into cleanly split components: what lies "inside" a subspace and what lies "outside." This idea, simple yet profound, runs through geometry, data analysis, and physics alike.

75. Orthogonal Decomposition Theorem

One of the cornerstones of linear algebra is the orthogonal decomposition theorem, which states that every vector in an inner product space can be uniquely split into two parts: one lying inside a subspace and the other lying in its orthogonal complement. This gives us a clear way to organize information, separate influences, and simplify computations.

Statement of the Theorem

Let V be an inner product space and W a subspace of V. Then for every vector $x \in V$, there exist unique vectors $w \in W$ and $z \in W^\perp$ such that

$$x = w + z.$$

Here:

- $w = \text{proj}_W(x)$, the projection of x onto W.
- $z = x - \text{proj}_W(x)$, the orthogonal component.

This decomposition is unique: no other pair of vectors from W and $W^\perp$ adds up to x.

Example in $\mathbb{R}^2$

Take W to be the line spanned by $u = (1, 2)$. For $x = (4, 1)$:

1. Projection:

$$\text{proj}_u(x) = \frac{\langle x, u \rangle}{\langle u, u \rangle} u.$$

   Compute: $\langle x, u \rangle = 4 \cdot 1 + 1 \cdot 2 = 6$, and $\langle u, u \rangle = 1^2 + 2^2 = 5$. So

$$\text{proj}_u(x) = \frac{6}{5}(1, 2) = \left(\tfrac{6}{5}, \tfrac{12}{5}\right).$$

2. Orthogonal component:

$$z = x - \text{proj}_u(x) = (4, 1) - \left(\tfrac{6}{5}, \tfrac{12}{5}\right) = \left(\tfrac{14}{5}, -\tfrac{7}{5}\right).$$

3. Verify: $\langle u, z \rangle = 1 \cdot \frac{14}{5} + 2 \cdot (-\frac{7}{5}) = 0$. Thus, $z \in W^\perp$.

So we have

$$x = \underbrace{\left(\tfrac{6}{5}, \tfrac{12}{5}\right)}_{\in W} + \underbrace{\left(\tfrac{14}{5}, -\tfrac{7}{5}\right)}_{\in W^\perp}.$$

Geometric Meaning

- The decomposition splits x into its "in-subspace" part and its "out-of-subspace" part.
- w is the closest point in W to x.
- z is the leftover "error," always perpendicular to W.

Geometrically, the shortest path from x to a subspace is always orthogonal.

Orthogonal Complements

- The orthogonal complement $W^\perp$ contains all vectors orthogonal to every vector in W.
- Dimensional relationship:

$$\dim(W) + \dim(W^\perp) = \dim(V).$$

- Together, W and $W^\perp$ partition the space V.

Projection Matrices and Decomposition

If P is the projection matrix onto W:

$$x = Px + (I - P)x,$$

where $Px \in W$ and $(I - P)x \in W^\perp$.

This formulation is used constantly in numerical linear algebra.

Applications

1. Least Squares Approximation: The best-fit solution is the projection; the error lies in the orthogonal complement.
2. Fourier Analysis: Any signal decomposes into a sum of components along orthogonal basis functions plus residuals.
3. Statistics: Regression decomposes data into explained variance (in the subspace of predictors) and residual variance (orthogonal).
4. Engineering: Splitting forces into parallel and perpendicular components relative to a surface.
5. Computer Graphics: Decomposing movement into screen-plane projection and depth (orthogonal direction).

Why It Matters

- Orthogonal decomposition gives clarity: every vector splits into "relevant" and "irrelevant" parts relative to a chosen subspace.
- It provides the foundation for least squares, regression, and signal approximation.
- It ensures uniqueness, stability, and interpretability in vector computations.

Try It Yourself

1. In $\mathbb{R}^3$, decompose $x = (1, 2, 3)$ into components in $\text{span}(1, 0, 0)$ and its orthogonal complement.
2. Show that if W is spanned by $(1, 1, 0)$ and $(0, 1, 1)$, then any vector in $\mathbb{R}^3$ can be uniquely split into W and $W^\perp$.
3. Write down the projection matrix P for $W = \text{span}\{(1, 0, 0), (0, 1, 0)\}$ in $\mathbb{R}^3$. Verify that $I - P$ projects onto $W^\perp$.
4. Challenge: Prove the orthogonal decomposition theorem using projection matrices and the fact that $P^2 = P$.

The orthogonal decomposition theorem guarantees that every vector finds its closest approximation in a chosen subspace and a perfectly perpendicular remainder-an elegant structure that makes analysis and computation possible in countless domains.

76. Orthogonal Projections and Least Squares

One of the deepest connections in linear algebra is between orthogonal projections and the least squares method. When equations don't have an exact solution, least squares finds the "best approximate" one. The theory behind it is entirely geometric: the best solution is the projection of a vector onto a subspace.

The Setup: Overdetermined Systems

Consider a system of equations $Ax = b$, where

- A is an $m \times n$ matrix with $m > n$ (more equations than unknowns).
- $b \in \mathbb{R}^m$ may not lie in the column space of A.

This means:

- There may be no exact solution.
- Instead, we want x that makes Ax as close as possible to b.

Least Squares Problem

The least squares solution minimizes the error:

$$\min_x \|Ax - b\|^2.$$

Here:

- Ax is the projection of b onto the column space of A.
- The error vector $b - Ax$ is orthogonal to the column space.

This is exactly the orthogonal decomposition theorem applied to b.

Derivation of Normal Equations

We want $r = b - Ax$ to be orthogonal to every column of A:

$$A^T(b - Ax) = 0.$$

Rearranging:

$$A^T Ax = A^T b.$$

This system is called the normal equations. Its solution x gives the least squares approximation.

Projection Matrix in Least Squares

The projection of b onto $\text{Col}(A)$ is

$$\hat{b} = A(A^T A)^{-1} A^T b,$$

assuming $A^T A$ is invertible.

Here,

- $P = A(A^T A)^{-1} A^T$ is the projection matrix onto the column space of A.
- The fitted vector is $\hat{b} = Pb$.
- The residual $r = b - \hat{b}$ lies in the orthogonal complement of $\text{Col}(A)$.

Example

Suppose $A = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $b = \begin{bmatrix} 2 \\ 2 \\ 4 \end{bmatrix}$.

- Column space of A: span of $(1, 2, 3)$.
- Projection formula:

$$\hat{b} = \frac{\langle b, A \rangle}{\langle A, A \rangle} A.$$

- Compute: $\langle b, A \rangle = 2 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 = 18$. $\langle A, A \rangle = 1^2 + 2^2 + 3^2 = 14$.
- Projection:

$$\hat{b} = \frac{18}{14}(1, 2, 3) = \left(\tfrac{9}{7}, \tfrac{18}{7}, \tfrac{27}{7}\right).$$

- Residual:

$$r = b - \hat{b} = \left(\tfrac{5}{7}, -\tfrac{4}{7}, \tfrac{1}{7}\right).$$

Check: $\langle r, A \rangle = 0$, so it's orthogonal.

Geometric Meaning

- The least squares solution is the point in $\mathrm{Col}(A)$ closest to b.
- The error vector is orthogonal to the subspace.
- This is like dropping a perpendicular from b to the subspace $\mathrm{Col}(A)$.

Applications

1. Statistics: Linear regression uses least squares to fit models to data.
2. Engineering: Curve fitting, system identification, and calibration.
3. Computer Graphics: Best-fit transformations (e.g., Procrustes analysis).
4. Machine Learning: Optimization of linear models (before moving to nonlinear methods).
5. Numerical Methods: Solving inconsistent systems of equations.

Why It Matters

- Orthogonal projections explain why least squares gives the best approximation.
- They reveal the geometry behind regression: data is projected onto the model space.
- They connect linear algebra with statistics, optimization, and applied sciences.

Try It Yourself

1. Solve $\min_x \|Ax - b\|$ for $A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix}$, $b = (1, 2, 2)^T$. Interpret the result.
2. Derive the projection matrix P for this system.
3. Show that the residual is orthogonal to each column of A.
4. Challenge: Prove that among all possible approximations Ax, the least squares solution is unique if and only if A^TA is invertible.

Orthogonal projections turn the messy, unsolvable world of overdetermined equations into one of best possible approximations. Least squares is not just an algebraic trick-it is the geometric essence of "closeness" in higher-dimensional spaces.

77. QR Decomposition

QR decomposition is a factorization of a matrix into an orthogonal part and a triangular part. It grows directly out of orthogonality and the Gram–Schmidt process, and it plays a central role in numerical linear algebra, providing a stable and efficient way to solve systems, compute least squares solutions, and analyze matrices.

Definition

For a real $m \times n$ matrix A with linearly independent columns:

$$A = QR,$$

where:

- Q is an $m \times n$ matrix with orthonormal columns ($Q^TQ = I$).
- R is an $n \times n$ upper triangular matrix.

This decomposition is unique if we require R to have positive diagonal entries.

Connection to Gram–Schmidt

The Gram–Schmidt process applied to the columns of A produces the orthonormal columns of Q. The coefficients used during the orthogonalization steps naturally form the entries of R.

- Each column of A is expressed as a combination of the orthonormal columns of Q.
- The coefficients of this expression populate the triangular matrix R.

Example

Let

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

1. Apply Gram–Schmidt to the columns:
   - $v_1 = (1, 1, 0)^T$, normalize:

     $$u_1 = \frac{1}{\sqrt{2}}(1, 1, 0)^T.$$

   - Subtract projection from $v_2 = (1, 0, 1)^T$:

     $$w_2 = v_2 - \langle v_2, u_1 \rangle u_1.$$

     Compute $\langle v_2, u_1 \rangle = \frac{1}{\sqrt{2}}(1 + 0 + 0) = \frac{1}{\sqrt{2}}$. So

     $$w_2 = (1, 0, 1)^T - \tfrac{1}{\sqrt{2}}(1, 1, 0)^T = \left(\tfrac{1}{2}, -\tfrac{1}{2}, 1\right)^T.$$

     Normalize:

     $$u_2 = \frac{1}{\sqrt{1.5}} \left(\tfrac{1}{2}, -\tfrac{1}{2}, 1\right)^T.$$

2. Construct $Q = [u_1, u_2]$.
3. Compute $R = Q^T A$.

The result is $A = QR$, with Q orthonormal and R triangular.

Geometric Meaning

- Q represents an orthogonal change of basis-rotations and reflections that preserve length and angle.
- R encodes scaling and shear in the new orthonormal coordinate system.
- Together, they show how A transforms space: first rotate into a clean basis, then apply triangular distortion.

Applications

1. Least Squares: Instead of solving $A^TAx = A^Tb$, we use QR:

   $$Ax = b \quad \Rightarrow \quad QRx = b.$$

   Multiply by Q^T:

   $$Rx = Q^Tb.$$

   Since R is triangular, solving for x is efficient and numerically stable.

2. Eigenvalue Algorithms: The QR algorithm iteratively applies QR factorizations to approximate eigenvalues.

3. Numerical Stability: Orthogonal transformations minimize numerical errors compared to solving normal equations.

4. Machine Learning: Many algorithms (e.g., linear regression, PCA) use QR decomposition for efficiency and stability.

5. Computer Graphics: Orthogonal factors preserve shapes; triangular factors simplify transformations.

Why It Matters

- QR decomposition bridges theory (Gram–Schmidt orthogonalization) and computation (matrix factorization).
- It avoids pitfalls of normal equations, improving numerical stability.
- It underpins algorithms across statistics, engineering, and computer science.

Try It Yourself

1. Compute the QR decomposition of

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}.$$

2. Verify that $Q^TQ = I$ and R is upper triangular.
3. Use QR to solve the least squares problem $Ax \approx b$ with $b = (1, 1, 1)^T$.
4. Challenge: Show that if A is square and orthogonal, then $R = I$ and $Q = A$.

QR decomposition turns the messy process of solving least squares into a clean, geometric procedure-rotating into a better coordinate system before solving. It is one of the most powerful tools in the linear algebra toolkit.

78. Orthogonal Matrices

Orthogonal matrices are square matrices whose rows and columns form an orthonormal set. They are the algebraic counterpart of rigid motions in geometry: transformations that preserve lengths, angles, and orientation (except for reflections).

Definition

A square matrix $Q \in \mathbb{R}^{n \times n}$ is orthogonal if

$$Q^TQ = QQ^T = I.$$

This means:

- The columns of Q are orthonormal.
- The rows of Q are also orthonormal.

Properties

1. Inverse Equals Transpose:

$$Q^{-1} = Q^T.$$

   This makes orthogonal matrices especially easy to invert.

2. Preservation of Norms: For any vector x,

$$\|Qx\| = \|x\|.$$

   Orthogonal transformations never stretch or shrink vectors.

3. Preservation of Inner Products:

$$\langle Qx, Qy \rangle = \langle x, y \rangle.$$

   Angles are preserved.

4. Determinant: $\det(Q) = \pm 1$.

   - If $\det(Q) = 1$, Q is a rotation.
   - If $\det(Q) = -1$, Q is a reflection combined with rotation.

Examples

1. 2D Rotation Matrix:

$$Q = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

   Rotates vectors by angle θ.

2. 2D Reflection Matrix: Reflection across the x-axis:

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

3. Permutation Matrices: Swapping coordinates is orthogonal because it preserves lengths. Example in 3D:

$$Q = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Geometric Meaning

Orthogonal matrices represent isometries: transformations that preserve the shape of objects.

- They can rotate, reflect, or permute axes.
- They never distort lengths or angles.

This is why in computer graphics, orthogonal matrices model pure rotations and reflections without scaling.

Applications

1. Computer Graphics: Rotations of 3D models use orthogonal matrices to avoid distortion.
2. Numerical Linear Algebra: Orthogonal transformations are numerically stable, widely used in QR factorization and eigenvalue algorithms.
3. Data Compression: Orthogonal transforms like the Fourier and cosine transforms preserve energy.
4. Signal Processing: Orthogonal filters separate signals into independent components.
5. Physics: Orthogonal matrices describe rotations in rigid body dynamics.

Why It Matters

- Orthogonal matrices are the building blocks of stable algorithms.
- They describe symmetry, structure, and invariance in physical and computational systems.
- They serve as the simplest and most powerful class of transformations that preserve geometry exactly.

Try It Yourself

1. Verify that

$$Q = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

   is orthogonal. What geometric transformation does it represent?

2. Prove that the determinant of an orthogonal matrix must be ± 1.

3. Show that multiplying two orthogonal matrices gives another orthogonal matrix.

4. Challenge: Prove that eigenvalues of orthogonal matrices lie on the complex unit circle (i.e., $|\lambda| = 1$).

Orthogonal matrices capture the essence of symmetry: transformations that preserve structure exactly. They lie at the heart of geometry, physics, and computation.

79. Fourier Viewpoint

The Fourier viewpoint is one of the most profound connections in linear algebra: the idea that complex signals, data, or functions can be decomposed into sums of simpler, orthogonal waves. Instead of describing information in its raw form (time, space, or coordinates), we express it in terms of frequencies. This perspective reshapes how we analyze, compress, and understand information across mathematics, physics, and engineering.

Fourier Series: The Basic Idea

Suppose we have a periodic function $f(x)$ defined on $[-\pi, \pi]$. The Fourier series expresses $f(x)$ as:

$$f(x) = a_0 + \sum_{n=1}^{\infty} \left(a_n \cos(nx) + b_n \sin(nx) \right).$$

- The coefficients a_n, b_n are found using inner products with sine and cosine functions.
- Each sine and cosine is orthogonal to the others under the inner product

$$\langle f, g \rangle = \int_{-\pi}^{\pi} f(x) g(x)\, dx.$$

Thus, Fourier series is nothing more than expanding a function in an orthonormal basis of trigonometric functions.

Fourier Transform: From Time to Frequency

For non-periodic signals, the Fourier transform generalizes this expansion. For a function $f(t)$,

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

transforms it into frequency space. The inverse transform reconstructs $f(t)$ from its frequencies.

This is again an inner product viewpoint: the exponential functions $e^{i\omega t}$ act as orthogonal basis functions on $\mathbb{R}$.

Orthogonality of Waves

The trigonometric functions $\{\cos(nx), \sin(nx)\}$ and the complex exponentials $\{e^{i\omega t}\}$ form orthogonal families.

- Two different sine waves have zero inner product over a full period.
- Likewise, exponentials with different frequencies are orthogonal.

This is exactly like orthogonal vectors in $\mathbb{R}^n$, except here the space is infinite-dimensional.

Discrete Fourier Transform (DFT)

In computational settings, we don't work with infinite integrals but with finite data. The DFT expresses an n-dimensional vector $x = (x_0, \ldots, x_{n-1})$ as a linear combination of orthogonal complex exponentials:

$$X_k = \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n}, \quad k = 0, \ldots, n-1.$$

This is simply a change of basis: from the standard basis (time domain) to the Fourier basis (frequency domain).

The Fast Fourier Transform (FFT) computes this in $O(n \log n)$ time, making Fourier analysis practical at scale.

Geometric Meaning

- In the time domain, data is expressed as a sequence of raw values.
- In the frequency domain, data is expressed as amplitudes of orthogonal waves.
- The Fourier viewpoint is just a rotation into a new orthogonal coordinate system, exactly like diagonalizing a matrix or changing basis.

Applications

1. Signal Processing: Filtering unwanted noise corresponds to removing high-frequency components.
2. Image Compression: JPEG uses Fourier-like transforms (cosine transforms) to compress images.
3. Data Analysis: Identifying cycles and periodic patterns in time series.
4. Physics: Quantum states are represented in both position and momentum bases, linked by Fourier transform.
5. Partial Differential Equations: Solutions are simplified by moving to frequency space, where derivatives become multipliers.

Why It Matters

- Fourier methods turn difficult problems into simpler ones: convolution becomes multiplication, differentiation becomes scaling.
- They provide a universal language for analyzing periodicity, oscillation, and wave phenomena.
- They are linear algebra at heart: orthogonal expansions in special bases.

Try It Yourself

1. Compute the Fourier series coefficients for $f(x) = x$ on $[-\pi, \pi]$.
2. For the sequence $(1, 0, 0, 0)$, compute the 4-point DFT and interpret the result.
3. Show that $\int_{-\pi}^{\pi} \sin(mx)\cos(nx)dx = 0$.
4. Challenge: Prove that the Fourier basis $\{e^{i2\pi kt}\}_{k=0}^{n-1}$ is orthonormal in $\mathbb{C}^n$.

The Fourier viewpoint reveals that every signal, no matter how complex, can be seen as a combination of simple, orthogonal waves. It is a perfect marriage of geometry, algebra, and analysis, and one of the most important ideas in modern mathematics.

80. Polynomial and Multifeature Least Squares

Least squares problems become especially powerful when extended to fitting polynomials or handling multiple features at once. Instead of a single straight line through data, we can fit curves of higher degree or surfaces in higher dimensions. These generalizations lie at the heart of regression, data analysis, and scientific modeling.

From Line to Polynomial

The simplest least squares model is a straight line:

$$y \approx \beta_0 + \beta_1 x.$$

But many relationships are nonlinear. Polynomial least squares generalizes the model to:

$$y \approx \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_d x^d.$$

Here, each power of x is treated as a new feature. The problem reduces to ordinary least squares on the design matrix:

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^d \\ 1 & x_2 & x_2^2 & \dots & x_2^d \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^d \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_d \end{bmatrix}.$$

The least squares solution minimizes $\|A\beta - y\|$.

Multiple Features

When data involves several predictors, we extend the model to:

$$y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

In matrix form:

$$y \approx A\beta,$$

where A is the design matrix with columns corresponding to features (including a column of ones for the intercept).

The least squares solution is still given by the normal equations:

$$\hat{\beta} = (A^T A)^{-1} A^T y,$$

or more stably by QR or SVD factorizations.

Example: Polynomial Fit

Suppose we have data points $(1, 1), (2, 2.2), (3, 2.9), (4, 4.1)$. Fitting a quadratic model $y \approx \beta_0 + \beta_1 x + \beta_2 x^2$:

1. Construct design matrix:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix}.$$

2. Solve least squares problem $\min \|A\beta - y\|$.
3. The result gives coefficients $\beta_0, \beta_1, \beta_2$ that best approximate the curve.

The same process works for higher-degree polynomials or multiple features.

Geometric Meaning

- In polynomial least squares, the feature space expands: instead of points on a line, data lives in a higher-dimensional feature space $(1, x, x^2, \ldots, x^d)$.
- In multifeature least squares, the column space of A spans all possible linear combinations of features.
- The least squares solution projects the observed output vector y onto this subspace.

Thus, whether polynomial or multifeature, the geometry is the same: projection onto the model space.

Practical Challenges

1. Overfitting: Higher-degree polynomials fit noise, not just signal.
2. Multicollinearity: Features may be correlated, making $A^T A$ nearly singular.
3. Scaling: Features with different magnitudes should be normalized.
4. Regularization: Adding penalty terms (ridge or LASSO) stabilizes the solution.

Applications

1. Regression in Statistics: Extending linear regression to handle multiple predictors or polynomial terms.
2. Machine Learning: Basis expansion for feature engineering (before neural nets, this was the standard).
3. Engineering: Curve fitting for calibration, modeling, and prediction.
4. Economics: Forecasting models with many variables (inflation, interest rates, spending).
5. Physics and Chemistry: Polynomial regression to model experimental data.

Why It Matters

- Polynomial least squares captures curvature in data.
- Multifeature least squares allows multiple predictors to explain outcomes.
- Both generalizations turn linear algebra into a practical modeling tool across science and society.

Try It Yourself

1. Fit a quadratic curve through points $(0, 1), (1, 2), (2, 5), (3, 10)$. Compare to a straight-line fit.
2. Construct a multifeature design matrix for predicting exam scores based on hours studied, sleep, and prior grades.
3. Show that polynomial regression is just linear regression on transformed features.
4. Challenge: Derive the bias–variance tradeoff in polynomial least squares-why higher degrees increase variance.

Polynomial and multifeature least squares extend the reach of linear algebra from straight lines to complex patterns, giving us a universal framework for modeling relationships in data.

Closing

```
Closest lines are drawn,
errors fall away to rest,
angles guard the truth.
```

Chapter 9. SVD, PCA, and conditioning

Opening

```
Closest lines are drawn,
errors fall away to rest,
angles guard the truth.
```

81. Singular Values and SVD

The Singular Value Decomposition (SVD) is one of the most powerful tools in linear algebra. It generalizes eigen-decomposition, works for all rectangular matrices (not just square ones), and provides deep insights into geometry, computation, and data analysis. At its core, the SVD tells us that every matrix can be factored into three pieces: rotations/reflections, scaling, and rotations/reflections again.

Definition of SVD

For any real $m \times n$ matrix A, the SVD is:

$$A = U\Sigma V^T,$$

where:

- U is an $m \times m$ orthogonal matrix (columns = left singular vectors).
- Σ is an $m \times n$ diagonal matrix with nonnegative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ (singular values).
- V is an $n \times n$ orthogonal matrix (columns = right singular vectors).

Even if A is rectangular or not diagonalizable, this factorization always exists.

Geometric Meaning

The SVD describes how A transforms space:

1. First rotation/reflection: Multiply by V^T to rotate or reflect coordinates into the right singular vector basis.
2. Scaling: Multiply by Σ, stretching/shrinking each axis by a singular value.
3. Second rotation/reflection: Multiply by U to reorient into the output space.

Thus, A acts as a rotation, followed by scaling, followed by another rotation.

Singular Values

- The singular values σ_i are the square roots of the eigenvalues of A^TA.
- They measure how much A stretches space in particular directions.
- The largest singular value σ_1 is the operator norm of A: the maximum stretch factor.
- If some singular values are zero, they correspond to directions collapsed by A.

Example in $\mathbb{R}^2$

Let

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}.$$

1. Compute $A^TA = \begin{bmatrix} 9 & 3 \\ 3 & 5 \end{bmatrix}$.
2. Find its eigenvalues: $\lambda_1, \lambda_2 = 10 \pm \sqrt{10}$.
3. Singular values: $\sigma_i = \sqrt{\lambda_i}$.
4. The corresponding eigenvectors form the right singular vectors V.
5. Left singular vectors U are obtained by $U = AV/\Sigma$.

This decomposition reveals how A reshapes circles into ellipses.

Links to Eigen-Decomposition

- Eigen-decomposition works only for square, diagonalizable matrices.
- SVD works for all matrices, square or rectangular, diagonalizable or not.
- Instead of eigenvalues (which may be complex or negative), we get singular values (always real and nonnegative).
- Eigenvectors can fail to exist in a full basis; singular vectors always form orthonormal bases.

Applications

1. Data Compression: Truncate small singular values to approximate matrices with fewer dimensions (used in JPEG).
2. Principal Component Analysis (PCA): SVD on centered data finds principal components, directions of maximum variance.
3. Least Squares Problems: SVD provides stable solutions, even for ill-conditioned or singular systems.
4. Noise Filtering: Discard small singular values to remove noise in signals and images.

5. Numerical Stability: SVD helps diagnose conditioning-how sensitive solutions are to input errors.

Why It Matters

- SVD is the "Swiss army knife" of linear algebra: versatile, always applicable, and rich in interpretation.
- It provides geometric, algebraic, and computational clarity.
- It is indispensable for modern applications in machine learning, statistics, engineering, and physics.

Try It Yourself

1. Compute the SVD of

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 0 & 0 \end{bmatrix}.$$

   Interpret the scaling and rotations.

2. Show that for any vector x, $\|Ax\| \leq \sigma_1 \|x\|$.

3. Use SVD to approximate the matrix

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

   with rank 1.

4. Challenge: Prove that the Frobenius norm of A is the square root of the sum of squares of its singular values.

The singular value decomposition is universal: every matrix can be dissected into rotations and scalings, revealing its structure and enabling powerful techniques across mathematics and applied sciences.

82. Geometry of SVD

The Singular Value Decomposition (SVD) is not just an algebraic factorization-it has a precise geometric meaning. It explains exactly how any linear transformation reshapes space: stretching, rotating, compressing, and possibly collapsing dimensions. Understanding this geometry turns SVD from a formal tool into an intuitive picture of what matrices do.

Transformation of the Unit Sphere

Take the unit sphere (or circle, in 2D) in the input space. When we apply a matrix A:

- The sphere is transformed into an ellipsoid.
- The axes of this ellipsoid correspond to the right singular vectors v_i.
- The lengths of the axes are the singular values σ_i.
- The directions of the axes in the output space are the left singular vectors u_i.

Thus, SVD tells us:

$$Av_i = \sigma_i u_i.$$

Every matrix maps orthogonal basis directions into orthogonal ellipsoid axes, scaled by singular values.

Step-by-Step Geometry

The decomposition $A = U\Sigma V^T$ can be read geometrically:

1. Rotate/reflect by V^T: Align input coordinates with the "principal directions" of A.
2. Scale by Σ: Stretch or compress each axis by its singular value. Some singular values may be zero, flattening dimensions.
3. Rotate/reflect by U: Reorient the scaled axes into the output space.

This process is universal: no matter how irregular a matrix seems, it always reshapes space by rotation $\rightarrow$ scaling $\rightarrow$ rotation.

2D Example

Take

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}.$$

- A circle in $\mathbb{R}^2$ is mapped into an ellipse.
- The ellipse's major and minor axes align with the right singular vectors of A.
- Their lengths equal the singular values.
- The ellipse itself is then oriented in the output plane according to the left singular vectors.

This makes SVD the perfect tool for visualizing how A "distorts" geometry.

Stretching and Rank

- If all singular values are positive, the ellipsoid has full dimension (no collapse).
- If some singular values are zero, A flattens the sphere along certain directions, lowering the rank.
- The rank of A equals the number of nonzero singular values.

Thus, rank-deficient matrices literally squash space into lower dimensions.

Distance and Energy Preservation

- The largest singular value σ_1 is how much A can stretch a vector.
- The smallest nonzero singular value σ_r (where $r = \text{rank}(A)$) measures how much the matrix compresses.
- The condition number $\kappa(A) = \sigma_1/\sigma_r$ measures distortion: small values mean nearly spherical stretching, large values mean extreme elongation.

Applications of the Geometry

1. Data Compression: Keeping only the largest singular values keeps the "major axes" of variation.
2. PCA: Data is analyzed along orthogonal axes of greatest variance (singular vectors).
3. Numerical Analysis: Geometry of SVD shows why ill-conditioned systems amplify errors-because some directions are squashed almost flat.
4. Signal Processing: Elliptical distortions correspond to filtering out certain frequency components.

5. Machine Learning: Dimensionality reduction is essentially projecting data onto the largest singular directions.

Why It Matters

- SVD transforms algebraic equations into geometric pictures.
- It reveals exactly how matrices warp space, offering intuition behind abstract operations.
- By interpreting ellipses, singular values, and orthogonal vectors, we gain visual clarity for problems in data, physics, and computation.

Try It Yourself

1. Draw the unit circle in $\mathbb{R}^2$, apply the matrix

$$A = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix},$$

   and sketch the resulting ellipse. Identify its axes and lengths.

2. Verify numerically that $Av_i = \sigma_i u_i$ for computed singular vectors and singular values.
3. For a rank-1 matrix, sketch how the unit circle collapses to a line segment.
4. Challenge: Prove that the set of vectors with maximum stretch under A are precisely the first right singular vectors.

The geometry of SVD gives us a universal lens: every linear transformation is a controlled distortion of space, built from orthogonal rotations and directional scalings.

83. Relation to Eigen-Decompositions

The Singular Value Decomposition (SVD) is often introduced as something entirely new, but it is deeply tied to eigen-decomposition. In fact, singular values and singular vectors emerge from the eigen-decomposition of certain symmetric matrices constructed from A. Understanding this connection shows why SVD always exists, why singular values are nonnegative, and how it generalizes eigen-analysis to all matrices, even rectangular ones.

Eigen-Decomposition Recap

For a square matrix $M \in \mathbb{R}^{n \times n}$, an eigen-decomposition is:

$$M = X \Lambda X^{-1},$$

where Λ is a diagonal matrix of eigenvalues and the columns of X are eigenvectors.

However:

- Not all matrices are diagonalizable.
- Eigenvalues may be complex.
- Rectangular matrices don't have eigenvalues at all.

This is where SVD provides a universal framework.

From $A^T A$ to Singular Values

For any $m \times n$ matrix A:

1. Consider the symmetric, positive semidefinite matrix $A^T A \in \mathbb{R}^{n \times n}$.
    - Symmetry ensures all eigenvalues are real.
    - Positive semidefiniteness ensures they are nonnegative.
2. The eigenvalues of $A^T A$ are squares of the singular values of A:

$$\lambda_i(A^T A) = \sigma_i^2.$$

3. The eigenvectors of $A^T A$ are the right singular vectors v_i.
4. Similarly, for AA^T, eigenvalues are the same σ_i^2, and eigenvectors are the left singular vectors u_i.

Thus:

$$Av_i = \sigma_i u_i, \quad A^T u_i = \sigma_i v_i.$$

This pair of relationships binds eigen-decomposition and SVD together.

Why Eigen-Decomposition Is Not Enough

- Eigen-decomposition requires a square matrix. SVD works for rectangular matrices.
- Eigenvalues can be negative or complex; singular values are always real and nonnegative.
- Eigenvectors may not exist as a complete basis; singular vectors always form orthonormal bases.

In short, SVD provides the robustness that eigen-decomposition lacks.

Example

Let

$$A = \begin{bmatrix} 3 & 0 \\ 4 & 0 \\ 0 & 5 \end{bmatrix}.$$

1. Compute $A^T A = \begin{bmatrix} 25 & 0 \\ 0 & 25 \end{bmatrix}$.
   - Eigenvalues: $25, 25$.
   - Singular values: $\sigma_1 = \sigma_2 = 5$.
2. Right singular vectors are eigenvectors of $A^T A$. Here, they form the standard basis.
3. Left singular vectors come from Av_i/σ_i.

So the geometry of SVD is fully encoded in eigen-analysis of $A^T A$ and AA^T.

Geometric Picture

- Eigenvectors of $A^T A$ describe directions in input space where A stretches without mixing directions.
- Eigenvectors of AA^T describe the corresponding directions in output space.
- Singular values tell us how much stretching occurs.

Thus, SVD is essentially eigen-decomposition in disguise-but applied to the right symmetric companions.

Applications of the Connection

1. PCA: Data covariance matrix $X^T X$ uses eigen-decomposition, but PCA is implemented with SVD directly.
2. Numerical Methods: Algorithms for SVD rely on eigen-analysis of $A^T A$.
3. Stability Analysis: The relationship ensures singular values are reliable measures of conditioning.
4. Signal Processing: Power in signals (variance) is explained by eigenvalues of covariance, which connect to singular values.
5. Machine Learning: Kernel PCA and related methods depend on this link to handle nonlinear features.

Why It Matters

- SVD explains every matrix transformation in terms of orthogonal bases and scalings.
- Its relationship with eigen-decomposition ensures that SVD is not an alien tool, but a generalization.
- The eigenview shows why SVD is guaranteed to exist and why singular values are always real and nonnegative.

Try It Yourself

1. Prove that if v is an eigenvector of $A^T A$ with eigenvalue λ, then Av is either zero or a left singular vector of A with singular value $\sqrt{\lambda}$.
2. For the matrix

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix},$$

   compute both eigen-decomposition and SVD. Compare the results.
3. Show that $A^T A$ and AA^T always share the same nonzero eigenvalues.
4. Challenge: Explain why an orthogonal diagonalization of $A^T A$ is enough to guarantee existence of the full SVD of A.

The relationship between SVD and eigen-decomposition unifies two of linear algebra's deepest ideas: every matrix transformation is built from eigen-geometry, stretched into a form that always exists and always makes sense.

84. Low-Rank Approximation (Best Small Models)

A central idea in data analysis, scientific computing, and machine learning is that many datasets or matrices are far more complicated in raw form than they truly need to be. Much of the apparent complexity hides redundancy, noise, or low-dimensional patterns. Low-rank approximation is the process of compressing a large, complicated matrix into a smaller, simpler version that preserves the most important information. This concept, grounded in the Singular Value Decomposition (SVD), lies at the heart of dimensionality reduction, recommender systems, and modern AI.

The General Problem

Suppose we have a matrix $A \in \mathbb{R}^{m \times n}$, perhaps representing:

- An image, with rows as pixels and columns as color channels.
- A ratings table, with rows as users and columns as movies.
- A word embedding matrix, with rows as words and columns as features.

Often, A is very large but highly structured. The question is:

Can we find a smaller matrix B of rank k (where $k \ll \min(m, n)$) that approximates A well?

Rank and Complexity

The rank of a matrix is the number of independent directions it encodes. High rank means complexity; low rank means redundancy.

- A rank-1 matrix can be written as an outer product of two vectors: uv^T.
- A rank-k matrix is a sum of k such outer products.
- Limiting rank controls how much structure we allow the approximation to capture.

The SVD Solution

The SVD provides a natural decomposition:

$$A = U\Sigma V^T,$$

where singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$ measure importance.

To approximate A with rank k:

$$A_k = U_k \Sigma_k V_k^T,$$

where we keep only the top k singular values and vectors.

This is not just a heuristic: it is the Eckart–Young theorem:

> Among all rank-k matrices, A_k minimizes the error $\|A - B\|$ (both in Frobenius and spectral norm).

Thus, SVD provides the *best possible* low-rank approximation.

Geometric Intuition

- Each singular value σ_i measures how strongly A stretches in the direction of singular vector v_i.
- Keeping the top k singular values means keeping the most important stretches and ignoring weaker directions.
- The approximation captures the "essence" of A while discarding small, noisy, or redundant effects.

Examples

1. Images A grayscale image can be stored as a matrix of pixel intensities. Using SVD, one can compress it by keeping only the largest singular values:

- $k = 10$: blurry but recognizable image.
- $k = 50$: much sharper, yet storage cost is far less than full.
- $k = 200$: nearly indistinguishable from the original.

This is practical image compression: fewer numbers, same perception.

2. Recommender Systems Consider a user–movie rating matrix. Although it may be huge, the true patterns (genre preferences, popularity trends) live in a low-dimensional subspace. A rank-k approximation captures these patterns, predicting missing ratings by filling in the structure.

3. Natural Language Processing (NLP) Word embeddings often arise from co-occurrence matrices. Low-rank approximation via SVD extracts semantic structure, enabling words like "king," "queen," and "crown" to cluster together.

Error and Trade-Offs

- Error decay: If singular values drop quickly, small k gives a great approximation. If they decay slowly, more terms are needed.
- Energy preserved: The squared singular values σ_i^2 represent variance captured. Keeping the first k terms preserves most of the "energy."
- Balance: Too low rank = oversimplification (loss of structure). Too high rank = no compression.

Practical Computation

For very large matrices, full SVD is expensive ($O(mn^2)$ for $m \geq n$). Alternatives include:

- Truncated SVD algorithms (Lanczos, randomized methods).
- Iterative methods that compute only the top k singular values.
- Incremental approaches that update low-rank models as new data arrives.

These are vital in modern data science, where datasets often have millions of entries.

Analogy

- Music playlist: Imagine a playlist with hundreds of songs, but most are variations on a few themes. A low-rank approximation is like keeping only the core melodies while discarding repetitive riffs.
- Photograph compression: Keeping only the brightest and most important strokes of light, while ignoring faint and irrelevant details.
- Book summary: Instead of the full text, you read the essential plot points. That's low-rank approximation.

Why It Matters

- Reveals hidden structure in high-dimensional data.
- Reduces storage and computational cost.
- Filters noise while preserving the signal.
- Provides the foundation for PCA, recommender systems, and dimensionality reduction.

Try It Yourself

1. Take a small 5×5 random matrix. Compute its SVD. Construct the best rank-1 approximation. Compare to the original.
2. Download a grayscale image (e.g., 256×256). Reconstruct it with 10, 50, and 100 singular values. Visually compare.
3. Prove the Eckart–Young theorem for the spectral norm: why can no other rank-k approximation do better than truncated SVD?
4. For a dataset with many features, compute PCA and explain why it is equivalent to finding a low-rank approximation.

Low-rank approximation shows how linear algebra captures the essence of complexity: most of what matters lives in a small number of dimensions. The art is in finding and using them effectively.

85. Principal Component Analysis (Variance and Directions)

Principal Component Analysis (PCA) is one of the most widely used techniques in statistics, data analysis, and machine learning. It provides a method to reduce the dimensionality of a dataset while retaining as much important information as possible. The central insight is that data often varies more strongly in some directions than others, and by focusing on those directions we can summarize the dataset with fewer dimensions, less noise, and more interpretability.

The Basic Question

Suppose we have data points in high-dimensional space, say $x_1, x_2, \ldots, x_m \in \mathbb{R}^n$. Each point might be:

- A face image flattened into thousands of pixels.
- A customer's shopping history across hundreds of products.
- A gene expression profile across thousands of genes.

Storing and working with all features directly is expensive, and many features may be redundant or correlated. PCA asks:

Can we re-express this data in a smaller set of directions that capture the most variability?

Variance as Information

The guiding principle of PCA is variance.

- Variance measures how spread out the data is along a direction.
- High variance directions capture meaningful structure (e.g., different facial expressions, major spending habits).
- Low variance directions often correspond to noise or unimportant fluctuations.

Thus, PCA searches for the directions (called principal components) along which the variance of the data is maximized.

Centering and Covariance

To begin, we center the data by subtracting the mean vector:

$$X_c = X - \mathbf{1}\mu^T,$$

where μ is the average of all data points.

The covariance matrix is then:

$$C = \frac{1}{m} X_c^T X_c.$$

- The diagonal entries measure variance of each feature.
- Off-diagonal entries measure how features vary together.

Finding principal components is equivalent to finding the eigenvectors of this covariance matrix.

The Eigenview

1. The eigenvectors of C are the directions (principal components).
2. The corresponding eigenvalues tell us how much variance lies along each component.
3. Sorting eigenvalues from largest to smallest gives the most informative to least informative directions.

If we keep the top k eigenvectors, we project data into a k-dimensional subspace that preserves most variance.

The SVD View

Another perspective uses the Singular Value Decomposition (SVD):

$$X_c = U\Sigma V^T.$$

- Columns of V are the principal directions.
- Singular values squared (σ_i^2) correspond to eigenvalues of the covariance matrix.
- Projecting onto the first k columns of V gives the reduced representation.

This makes PCA and SVD essentially the same computation.

A Simple Example

Imagine we measure height and weight of 1000 people. Plotting them shows a strong correlation: taller people are often heavier. The cloud of points stretches along a diagonal.

- PCA's first component is this diagonal line: the direction of maximum variance.
- The second component is perpendicular, capturing the much smaller differences (like people of equal height but slightly different weights).
- Keeping only the first component reduces two features into one while retaining most of the information.

Geometric Picture

- PCA rotates the coordinate system so that axes align with directions of greatest variance.
- Projecting onto the top k components flattens the data into a lower-dimensional space, like flattening a tilted pancake onto its broadest plane.

Applications

1. Data Compression: Reduce storage by keeping only leading components (e.g., compressing images).
2. Noise Reduction: Small-variance directions often correspond to measurement noise; discarding them yields cleaner data.
3. Visualization: Reducing data to 2D or 3D for scatterplots helps us see clusters and patterns.
4. Preprocessing in Machine Learning: Many models train faster and generalize better on PCA-transformed data.
5. Genomics and Biology: PCA finds major axes of variation across thousands of genes.
6. Finance: PCA summarizes correlated movements of stocks into a few principal "factors."

Trade-Offs and Limitations

- Interpretability: Principal components are linear combinations of original features, sometimes hard to explain in plain terms.
- Linearity: PCA only captures linear relationships; nonlinear methods (like kernel PCA, t-SNE, or UMAP) may be better for curved manifolds.
- Scaling: Features must be normalized properly; otherwise, PCA might overemphasize units with large raw variance.
- Global Method: PCA captures overall variance, not local structures (e.g., small clusters within the data).

Mathematical Guarantees

PCA has an optimality guarantee:

- Among all k-dimensional linear subspaces, the PCA subspace minimizes the reconstruction error (squared Euclidean distance between data and its projection).
- This is essentially the low-rank approximation theorem seen earlier, applied to covariance matrices.

Why It Matters

PCA shows how linear algebra transforms raw data into insight. By focusing on variance, it provides a principled way to filter noise, compress information, and reveal hidden patterns. It is simple, computationally efficient, and foundational-almost every modern data pipeline uses PCA, explicitly or implicitly.

Try It Yourself

1. Take a dataset of two correlated features (like height and weight). Compute the covariance matrix, eigenvectors, and project onto the first component. Visualize before and after.
2. For a grayscale image stored as a matrix, flatten it into vectors and apply PCA. How many components are needed to reconstruct it with 90% accuracy?
3. Use PCA on the famous Iris dataset (4 features). Plot the data in 2D using the first two components. Notice how species separate in this reduced space.
4. Prove that the first principal component is the unit vector v that maximizes $\|X_c v\|^2$.

PCA distills complexity into clarity: it tells us not just where the data is, but where it *really goes.*

86. Pseudoinverse (Moore–Penrose) and Solving Ill-Posed Systems

In linear algebra, the inverse of a matrix is a powerful tool: if A is invertible, then solving $Ax = b$ is as simple as $x = A^{-1}b$. But what happens when A is not square, or not invertible? In practice, this is the norm: many problems involve rectangular matrices (more equations than unknowns, or more unknowns than equations), or square matrices that are singular. The Moore–Penrose pseudoinverse, usually denoted A^+, generalizes the idea of an inverse to all matrices, providing a systematic way to find solutions-or best approximations-when ordinary inversion fails.

Why Ordinary Inverses Fail

- Non-square matrices: If A is $m \times n$ with $m \neq n$, no standard inverse exists.
- Singular matrices: Even if A is square, if $\det(A) = 0$, it has no inverse.
- Ill-posed problems: In real-world data, exact solutions may not exist (inconsistent systems) or may not be unique (underdetermined systems).

Despite these obstacles, we still want a systematic way to solve or approximate $Ax = b$.

Definition of the Pseudoinverse

The Moore–Penrose pseudoinverse A^+ is defined as the unique matrix that satisfies four properties:

1. $AA^+A = A$.
2. $A^+AA^+ = A^+$.
3. $(AA^+)^T = AA^+$.
4. $(A^+A)^T = A^+A$.

These conditions ensure A^+ acts as an "inverse" in the broadest consistent sense.

Constructing the Pseudoinverse with SVD

Given the SVD of A:

$$A = U\Sigma V^T,$$

where Σ is diagonal with singular values $\sigma_1, \ldots, \sigma_r$, the pseudoinverse is:

$$A^+ = V\Sigma^+U^T,$$

where Σ^+ is formed by inverting nonzero singular values and transposing the matrix. Specifically:

- If $\sigma_i \neq 0$, replace it with $1/\sigma_i$.
- If $\sigma_i = 0$, leave it as 0.

This definition works for all matrices, square or rectangular.

Solving Linear Systems with A^+

1. Overdetermined systems ($m > n$, more equations than unknowns):
   - Often no exact solution exists.
   - The pseudoinverse gives the least-squares solution:

     $$x = A^+ b,$$

     which minimizes $\|Ax - b\|$.
2. Underdetermined systems ($m < n$, more unknowns than equations):
   - Infinitely many solutions exist.
   - The pseudoinverse chooses the solution with the smallest norm:

     $$x = A^+ b,$$

     which minimizes $\|x\|$ among all solutions.
3. Square but singular systems:
   - Some solutions exist, but not uniquely.
   - The pseudoinverse again picks the least-norm solution.

Example 1: Overdetermined

Suppose we want to solve:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 0 \end{bmatrix} x = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}.$$

This 3×2 system has no exact solution. Using the pseudoinverse, we obtain the least-squares solution that best fits all three equations simultaneously.

Example 2: Underdetermined

For

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} x = \begin{bmatrix} 3 \\ 4 \end{bmatrix},$$

the system has infinitely many solutions because x_3 is free. The pseudoinverse gives:

$$x = \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix},$$

choosing the solution with minimum norm.

Geometric Interpretation

- The pseudoinverse acts like projecting onto subspaces.
- For overdetermined systems, it projects b onto the column space of A, then finds the closest x.
- For underdetermined systems, it picks the point in the solution space closest to the origin.

So A^+ embodies the principle of "best possible inverse" under the circumstances.

Applications

1. Least-Squares Regression: Solving $\min_x \|Ax - b\|^2$ via A^+.
2. Signal Processing: Reconstructing signals from incomplete or noisy data.
3. Control Theory: Designing inputs when exact control is impossible.
4. Machine Learning: Training models with non-invertible design matrices.
5. Statistics: Computing generalized inverses of covariance matrices.

Limitations

- Sensitive to very small singular values: numerical instability may occur.
- Regularization (like ridge regression) is often preferred in noisy settings.
- Computationally expensive for very large matrices, though truncated SVD can help.

Why It Matters

The pseudoinverse is a unifying idea: it handles inconsistent, underdetermined, or singular problems with one formula. It ensures we always have a principled answer, even when classical algebra says "no solution" or "infinitely many solutions." In real data analysis, almost every problem is ill-posed to some degree, making the pseudoinverse a practical cornerstone of modern applied linear algebra.

Try It Yourself

1. Compute the pseudoinverse of a simple 2×2 singular matrix by hand using SVD.
2. Solve both an overdetermined (3×2) and underdetermined (2×3) system using A^+. Compare with intuitive expectations.
3. Explore what happens numerically when singular values are very small. Try truncating them-this connects to regularization.

The Moore–Penrose pseudoinverse shows that even when linear systems are "broken," linear algebra still provides a systematic way forward.

87. Conditioning and Sensitivity (How Errors Amplify)

Linear algebra is not only about exact solutions-it is also about how *stable* those solutions are when data is perturbed. In real-world applications, every dataset contains noise: measurement errors in physics experiments, rounding errors in financial computations, or floating-point precision limits in numerical software. Conditioning is the study of how sensitive the solution

of a problem is to small changes in input. A well-conditioned problem reacts gently to perturbations; an ill-conditioned one amplifies errors dramatically.

The Basic Idea

Suppose we solve the linear system:

$$Ax = b.$$

Now imagine we slightly change b to $b + \delta b$. The new solution is $x + \delta x$.

- If $\|\delta x\|$ is about the same size as $\|\delta b\|$, the problem is well-conditioned.
- If $\|\delta x\|$ is much larger, the problem is ill-conditioned.

Conditioning measures this amplification factor.

Condition Number

The central tool is the condition number of a matrix A:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|,$$

where $\|\cdot\|$ is a matrix norm (often the 2-norm).

- If $\kappa(A)$ is close to 1, the problem is well-conditioned.
- If $\kappa(A)$ is large (say, 10^6 or higher), the problem is ill-conditioned.

Interpretation:

- $\kappa(A)$ estimates the maximum relative error in the solution compared to the relative error in the data.
- In practical terms, every digit of accuracy in b may be lost in x if $\kappa(A)$ is too large.

Singular Values and Conditioning

Condition number in 2-norm can be expressed using singular values:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}},$$

where $\sigma_{\max}$ and $\sigma_{\min}$ are the largest and smallest singular values of A.

- If the smallest singular value is tiny compared to the largest, A nearly collapses some directions, making inversion unstable.
- This explains why nearly singular matrices are so problematic in numerical computation.

Example 1: A Stable System

$$A = \begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}.$$

Here, $\sigma_{\max} = 3, \sigma_{\min} = 2$. So $\kappa(A) = 3/2 = 1.5$. Very well-conditioned: small changes in input produce small changes in output.

Example 2: An Ill-Conditioned System

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix}.$$

The determinant is very small, so the system is nearly singular.

- One singular value is about 2.0.
- The other is about 0.0001.
- Condition number: $\kappa(A) \approx 20000$.

This means even tiny changes in b can wildly change x.

Geometric Intuition

A matrix transforms a unit sphere into an ellipse.

- The longest axis of the ellipse = $\sigma_{\max}$.
- The shortest axis = $\sigma_{\min}$.
- The ratio $\sigma_{\max}/\sigma_{\min}$ shows how stretched the transformation is.

If the ellipse is nearly flat, directions aligned with the short axis almost vanish, and recovering them is highly unstable.

Why Conditioning Matters in Computation

1. Numerical Precision: Computers store numbers with limited precision (floating-point). An ill-conditioned system magnifies rounding errors, leading to unreliable results.
2. Regression: In statistics, highly correlated features make the design matrix ill-conditioned, destabilizing coefficient estimates.
3. Machine Learning: Ill-conditioning leads to unstable training, exploding or vanishing gradients.
4. Engineering: Control systems based on ill-conditioned models may be hypersensitive to measurement errors.

Techniques for Handling Ill-Conditioning

- Regularization: Add a penalty term, like ridge regression (λI), to stabilize inversion.
- Truncated SVD: Ignore tiny singular values that only amplify noise.
- Scaling and Preconditioning: Rescale data or multiply by a well-chosen matrix to improve conditioning.
- Avoiding Explicit Inverses: Use factorizations (LU, QR, SVD) rather than computing A^{-1}.

Connection to Previous Topics

- Pseudoinverse: Ill-conditioning is visible when singular values approach zero, making A^+ unstable.
- Low-rank approximation: Truncating small singular values both compresses data and improves conditioning.
- PCA: Discarding low-variance components is essentially a conditioning improvement step.

Why It Matters

Conditioning bridges abstract algebra and numerical reality. Linear algebra promises solutions, but conditioning tells us whether those solutions are trustworthy. Without it, one might misinterpret noise as signal, or lose all accuracy in computations that look fine on paper.

Try It Yourself

1. Compute the condition number of $\begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix}$. Solve for x in $Ax = b$ for several slightly different b. Watch how solutions swing dramatically.

2. Take a dataset with nearly collinear features. Compute the condition number of its covariance matrix. Relate this to instability in regression coefficients.
3. Simulate numerical errors: Add random noise of size 10^{-6} to an ill-conditioned system and observe solution errors.
4. Prove that $\kappa(A) \geq 1$ always holds.

Conditioning reveals the hidden fragility of problems. It warns us when algebra says "solution exists" but computation whispers "don't trust it."

88. Matrix Norms and Singular Values (Measuring Size Properly)

In linear algebra, we often need to measure the "size" of a matrix. For vectors, this is straightforward: the length (norm) tells us how big the vector is. But for matrices, the question is more subtle: do we measure size by entries, by how much the matrix stretches vectors, or by some invariant property? Different contexts demand different answers, and matrix norms-closely tied to singular values-provide the framework for doing so.

Why Measure the Size of a Matrix?

1. Stability: To know how much error a matrix might amplify.
2. Conditioning: The ratio of largest to smallest stretching.
3. Optimization: Many algorithms minimize some matrix norm.
4. Data analysis: Norms measure complexity or energy of data.

Without norms, we cannot compare matrices, analyze sensitivity, or judge approximation quality.

Matrix Norms from Vector Norms

A natural way to define a matrix norm is to ask: *How much does this matrix stretch vectors?*

Formally, for a given vector norm $\|\cdot\|$:

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

This is called the induced matrix norm.

The 2-Norm and Singular Values

When we use the Euclidean norm ($\|x\|_2$) for vectors, the induced matrix norm becomes:

$$\|A\|_2 = \sigma_{\max}(A),$$

the largest singular value of A.

- This means the 2-norm measures the *maximum stretching factor*.
- Geometrically: A maps the unit sphere into an ellipse; $\|A\|_2$ is the length of the ellipse's longest axis.

This link makes singular values the natural language for matrix size.

Other Common Norms

1. Frobenius Norm

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}.$$

- Equivalent to the Euclidean length of all entries stacked in one big vector.
- Can also be expressed as:

$$\|A\|_F^2 = \sum_i \sigma_i^2.$$

- Often used in data science and machine learning because it is easy to compute and differentiable.

2. 1-Norm

$$\|A\|_1 = \max_j \sum_i |a_{ij}|,$$

the maximum absolute column sum.

3. Infinity Norm

$$\|A\|_\infty = \max_i \sum_j |a_{ij}|,$$

the maximum absolute row sum.

Both are computationally cheap, useful in numerical analysis.

4. Nuclear Norm (Trace Norm)

$$\|A\|_* = \sum_i \sigma_i,$$

the sum of singular values.

- Important in low-rank approximation and machine learning (matrix completion, recommender systems).

Singular Values as the Unifying Thread

- Spectral norm (2-norm): maximum singular value.
- Frobenius norm: root of the sum of squared singular values.
- Nuclear norm: sum of singular values.

Thus, norms capture different ways of summarizing singular values: maximum, sum, or energy.

Example: Small Matrix

Take

$$A = \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}.$$

- Singular values: $\sigma_1 = 5, \sigma_2 = 0$.
- $\|A\|_2 = 5$.
- $\|A\|_F = \sqrt{3^2 + 4^2} = 5$.
- $\|A\|_* = 5$.

Here, different norms coincide, but generally they highlight different aspects of the matrix.

Geometric Intuition

- 2-norm: "How much can A stretch a vector?"
- Frobenius norm: "What is the overall energy in all entries?"
- 1-norm / ∞-norm: "What is the heaviest column or row load?"
- Nuclear norm: "How much total stretching power does A have?"

Each is a lens, giving a different perspective.

Applications

1. Numerical Stability: Condition number $\kappa(A) = \sigma_{\max}/\sigma_{\min}$ uses the spectral norm.
2. Machine Learning: Nuclear norm is used for matrix completion (Netflix Prize).
3. Image Compression: Frobenius norm measures reconstruction error.
4. Control Theory: 1-norm and ∞-norm bound system responses.
5. Optimization: Norms serve as penalties or constraints, shaping solutions.

Why It Matters

Matrix norms provide the language to compare, approximate, and control matrices. Singular values ensure that this language is not arbitrary but grounded in geometry. Together, they explain how matrices distort space, how error grows, and how we can measure complexity.

Try It Yourself

1. For $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, compute $\|A\|_1$, $\|A\|_\infty$, $\|A\|_F$, and $\|A\|_2$ (using SVD for the last). Compare.
2. Prove that $\|A\|_F^2 = \sum \sigma_i^2$.
3. Show that $\|A\|_2 \leq \|A\|_F \leq \|A\|_*$. Interpret geometrically.
4. Consider a rank-1 matrix uv^T. What are its norms? Which are equal?

Matrix norms and singular values are the measuring sticks of linear algebra-they tell us not just how big a matrix is, but how it acts, where it is stable, and when it is fragile.

89. Regularization (Ridge/Tikhonov to Tame Instability)

When solving linear systems or regression problems, instability often arises because the system is ill-conditioned: tiny errors in data lead to huge swings in the solution. Regularization is the strategy of *adding stability* by deliberately modifying the problem, sacrificing exactness for robustness. The two most common approaches-ridge regression and Tikhonov regularization-embody this principle.

The Problem of Instability

Consider the least-squares problem:

$$\min_x \|Ax - b\|_2^2.$$

If A has nearly dependent columns, or if $\sigma_{\min}(A)$ is very small, then:

- Solutions are unstable.
- Coefficients x can explode in magnitude.
- Predictions vary wildly with small changes in b.

Regularization modifies the objective so that the solution prefers stability over exactness.

Ridge / Tikhonov Regularization

The modified problem is:

$$\min_x \left(\|Ax - b\|_2^2 + \lambda \|x\|_2^2\right),$$

where $\lambda > 0$ is the regularization parameter.

- The first term enforces data fit.
- The second term penalizes large coefficients, discouraging unstable solutions.

This is called ridge regression in statistics and Tikhonov regularization in numerical analysis.

The Closed-Form Solution

Expanding the objective and differentiating gives:

$$x_\lambda = (A^T A + \lambda I)^{-1} A^T b.$$

Key points:

- The added λI makes the matrix invertible, even if $A^T A$ is singular.
- As $\lambda \to 0$, the solution approaches the ordinary least-squares solution.
- As $\lambda \to \infty$, the solution shrinks toward 0.

SVD View

If $A = U\Sigma V^T$, then the least-squares solution is:

$$x = \sum_i \frac{u_i^T b}{\sigma_i} v_i.$$

If σ_i is very small, the term $\frac{1}{\sigma_i}$ causes instability.

With regularization:

$$x_\lambda = \sum_i \frac{\sigma_i}{\sigma_i^2 + \lambda} (u_i^T b) v_i.$$

- Small singular values (unstable directions) are suppressed.
- Large singular values (stable directions) are mostly preserved.

This explains why ridge regression stabilizes solutions: it damps noise-amplifying directions.

Geometric Interpretation

- The unregularized problem fits b exactly in the column space of A.
- Regularization tilts the solution toward the origin, shrinking coefficients.
- Geometrically, the feasible region (ellipsoid from Ax) intersects with a ball constraint from $\|x\|_2$. The solution is where these two shapes balance.

Extensions

1. Lasso (ℓ_1 regularization): Replaces $\|x\|_2^2$ with $\|x\|_1$, encouraging sparse solutions.
2. Elastic Net: Combines ridge and lasso penalties.
3. General Tikhonov: Uses $\|Lx\|_2^2$ with some matrix L, tailoring the penalty (e.g., smoothing in signal processing).
4. Bayesian View: Ridge regression corresponds to placing a Gaussian prior on coefficients.

Applications

- Machine Learning: Prevents overfitting in regression and classification.
- Signal Processing: Suppresses noise when reconstructing signals.
- Image Reconstruction: Stabilizes inverse problems like deblurring.
- Numerical PDEs: Adds smoothness constraints to solutions.
- Econometrics and Finance: Controls instability from highly correlated variables.

Why It Matters

Regularization transforms fragile problems into reliable ones. It acknowledges the reality of noise and finite precision, and instead of chasing impossible exactness, it provides usable, stable answers. In modern data-driven fields, almost every large-scale model relies on regularization for robustness.

Try It Yourself

1. Solve the system $Ax = b$ where

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

   Compare the unregularized least-squares solution with ridge-regularized solutions for $\lambda = 0.01, 1, 10$.

2. Using the SVD, show how coefficients for small singular values are shrunk.
3. In regression with many correlated features, compute coefficient paths as λ varies. Observe how they stabilize.
4. Explore image denoising: apply ridge regularization to a blurred/noisy image reconstruction problem.

Regularization shows the wisdom of linear algebra in practice: sometimes the best solution is not the exact one, but the stable one.

90. Rank-Revealing QR and Practical Diagnostics (What Rank Really Is)

Rank-the number of independent directions in a matrix-is central to linear algebra. It tells us about solvability of systems, redundancy of features, and the dimensionality of data. But in practice, computing rank is not as simple as counting pivots or checking determinants. Real-world data is noisy, nearly dependent, or high-dimensional. Rank-revealing QR (RRQR) factorization and related diagnostics provide stable, practical tools for uncovering rank and structure.

Why Rank Matters

- Linear systems: Rank determines if a system has a unique solution, infinitely many, or none.
- Data science: Rank measures intrinsic dimensionality, guiding dimensionality reduction.
- Numerics: Small singular values make effective rank ambiguous-exact vs. numerical rank diverge.

Thus, we need reliable algorithms to decide "how many directions matter" in a matrix.

Exact Rank vs. Numerical Rank

- Exact rank: Defined over exact arithmetic. A column is independent if it cannot be expressed as a linear combination of others.
- Numerical rank: In floating-point computation, tiny singular values cannot be trusted. A threshold ϵ determines when we treat them as zero.

For example, if the smallest singular value of A is 10^{-12}, and computations are in double precision ($\sim 10^{-16}$), we might consider the effective rank smaller than full.

The QR Factorization Recap

The basic QR factorization expresses a matrix $A \in \mathbb{R}^{m \times n}$ as:

$$A = QR,$$

where:

- Q is orthogonal ($Q^T Q = I$), preserving lengths.
- R is upper triangular, holding the "essence" of A.

QR is stable, fast, and forms the backbone of many algorithms.

Rank-Revealing QR (RRQR)

RRQR is an enhancement of QR with column pivoting:

$$AP = QR,$$

where P is a permutation matrix that reorders columns.

- The pivoting ensures that the largest independent directions come first.
- The diagonal entries of R indicate which columns are significant.
- Small values on the diagonal signal dependent (or nearly dependent) directions.

In practice, RRQR allows us to approximate rank by examining the decay of R's diagonal.

Comparing RRQR and SVD

- SVD: Gold standard for determining rank; singular values give exact scaling of each direction.
- RRQR: Faster and cheaper; sufficient when approximate rank is enough.
- Trade-off: SVD is more accurate, RRQR is more efficient.

Both are used depending on the balance of precision and cost.

Example

Let

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1.0001 & 2 \\ 1 & 2 & 3 \end{bmatrix}.$$

- Exact arithmetic: rank = 3.
- Numerically: second column is nearly dependent on the first. SVD shows a singular value near zero.
- RRQR with pivoting identifies the near-dependence by revealing a tiny diagonal in R.

Thus, RRQR "reveals" effective rank without fully computing SVD.

Practical Diagnostics for Rank

1. Condition Number: A high condition number suggests near-rank-deficiency.
2. Diagonal of R in RRQR: Monitors independence of columns.
3. Singular Values in SVD: Most reliable indicator, but expensive.
4. Determinants/Minors: Useful in theory, unstable in practice.

Applications

- Data Compression: Identifying effective rank allows truncation.
- Regression: Detecting multicollinearity by examining rank of the design matrix.
- Control Systems: Rank tests stability and controllability.
- Machine Learning: Dimensionality reduction pipelines (e.g., PCA) start with rank estimation.
- Signal Processing: Identifying number of underlying sources from mixtures.

Why It Matters

Rank is simple in theory, but elusive in practice. RRQR and related diagnostics bridge the gap between exact mathematics and noisy data. They allow practitioners to say, with stability and confidence: *this is how many independent directions really matter.*

Try It Yourself

1. Implement RRQR with column pivoting on a small 5×5 nearly dependent matrix. Compare estimated rank with SVD.
2. Explore the relationship between diagonal entries of R and numerical rank.
3. Construct a dataset with 100 features, where 95 are random noise but 5 are linear combinations. Use RRQR to detect redundancy.
4. Prove that column pivoting does not change the column space of A, only its numerical stability.

Rank-revealing QR shows that linear algebra is not only about exact formulas but also about practical diagnostics-knowing when two directions are truly different and when they are essentially the same.

Closing

```
Noise reduced to still,
singular values unfold space,
essence shines within.
```

Chapter 10. Applications and computation

Opening

```
Worlds in numbers bloom,
graphs and data interlace,
algebra takes flight.
```

91. 2D/3D Geometry Pipelines (Cameras, Rotations, and Transforms)

Linear algebra is the silent backbone of modern graphics, robotics, and computer vision. Every time an image is rendered on a screen, a camera captures a scene, or a robot arm moves in space, a series of matrix multiplications are transforming points from one coordinate system to another. These geometry pipelines map 3D reality into 2D representations, ensuring that objects appear in the correct position, orientation, and scale.

The Geometry of Coordinates

A point in 3D space is represented as a column vector:

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix}.$$

But computers often extend this to homogeneous coordinates, embedding the point in 4D:

$$p_h = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The extra coordinate allows translations to be represented as matrix multiplications, keeping the entire pipeline consistent: every step is just multiplying by a matrix.

Transformations in 2D and 3D

- Translation Moves a point by (t_x, t_y, t_z).

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

- Scaling Expands or shrinks space along each axis.

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

- Rotation In 3D, rotation around the z-axis is:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

  Similar forms exist for rotations around the x- and y-axes.

Each transformation is linear (or affine), and chaining them is just multiplying matrices.

The Camera Pipeline

Rendering a 3D object to a 2D image follows a sequence of steps, each one a matrix multiplication:

1. Model Transform Moves the object from its local coordinates into world coordinates.
2. View Transform Puts the camera at the origin and aligns its axes with the world, effectively changing the point of view.
3. Projection Transform Projects 3D points into 2D. Two types:
   - Orthographic: parallel projection, no perspective.
   - Perspective: distant objects appear smaller, closer to human vision.

Example of perspective projection:

$$P = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where f is focal length.

4. Viewport Transform Maps normalized 2D coordinates to screen pixels.

This sequence-from object to image-is the geometry pipeline.

Example: Rendering a Cube

- Start with cube vertices in local coordinates ($[-1, 1]^3$).
- Apply a scaling matrix to stretch it.
- Apply a rotation matrix to tilt it.
- Apply a translation matrix to move it into the scene.
- Apply a projection matrix to flatten it onto the screen.

Every step is linear algebra, and the final picture is the result of multiplying many matrices in sequence.

Robotics Connection

Robotic arms use similar pipelines: each joint contributes a rotation or translation, encoded as a matrix. By multiplying them, we get the forward kinematics-the position and orientation of the hand given the joint angles.

Why It Matters

Geometry pipelines unify graphics, robotics, and vision. They show how linear algebra powers the everyday visuals of video games, animations, simulations, and even self-driving cars. Without the consistency of matrix multiplication, the complexity of managing transformations would be unmanageable.

Try It Yourself

1. Write down the sequence of matrices that rotate a square by 45°, scale it by 2, and translate it by $(3, 1)$. Multiply them to get the combined transformation.
2. Construct a cube in 3D and simulate a perspective projection by hand for one vertex.
3. For a simple 2-joint robotic arm, represent each joint with a rotation matrix and compute the final hand position.
4. Prove that composing affine transformations is closed under multiplication-why does this make pipelines possible?

Geometry pipelines are the bridge between abstract linear algebra and tangible visual and mechanical systems. They are how math becomes movement, light, and image.

92. Computer Graphics and Robotics (Homogeneous Tricks in Action)

Linear algebra doesn't just stay on the chalkboard-it drives the engines of computer graphics and robotics. Both fields need to describe and manipulate objects in space, often moving between multiple coordinate systems. The homogeneous coordinate trick-adding one extra dimension-makes this elegant: translations, scalings, and rotations all fit into a single framework of matrix multiplication. This uniformity allows efficient computation and consistent pipelines.

Homogeneous Coordinates Recap

In 2D, a point (x, y) becomes $[x, y, 1]^T$. In 3D, a point (x, y, z) becomes $[x, y, z, 1]^T$.

Why add the extra 1? Because then translations-normally not linear-become linear in the higher-dimensional embedding. Every affine transformation (rotations, scalings, shears, reflections, and translations) is just a single multiplication by a homogeneous matrix.

Example:

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad p'_h = T p_h.$$

This trick makes pipelines modular: just multiply the matrices in order.

Computer Graphics Pipelines

Graphics engines (like OpenGL or DirectX) rely entirely on homogeneous transformations:

1. Model Matrix: Puts the object in the scene.
    - Example: Rotate a car 90° and translate it 10 units forward.
2. View Matrix: Positions the virtual camera.
    - Equivalent to moving the world so the camera sits at the origin.
3. Projection Matrix: Projects 3D points to 2D.
    - Perspective projection shrinks faraway objects, orthographic doesn't.
4. Viewport Matrix: Converts normalized 2D coordinates into screen pixels.

Every pixel you see in a video game has passed through this stack of matrices.

Robotics Pipelines

In robotics, the same principle applies:

- A robot arm with joints is modeled as a chain of rigid-body transformations.
- Each joint contributes a rotation or translation matrix.
- Multiplying them gives the final pose of the robot's end-effector (hand, tool, or gripper).

This is called forward kinematics.

Example: A 2D robotic arm with two joints:

$$p = R(\theta_1)T(l_1)R(\theta_2)T(l_2)\begin{bmatrix}0\\0\\1\end{bmatrix}.$$

Here $R(\theta_i)$ are rotation matrices and $T(l_i)$ are translations along the arm length. Multiplying them gives the position of the hand.

Shared Challenges in Graphics and Robotics

1. Precision: Numerical round-off errors can accumulate; stable algorithms are critical.
2. Speed: Both fields demand real-time computation-60 frames per second for graphics, millisecond reaction times for robots.
3. Hierarchy: Objects in graphics may be nested (a car's wheel rotates relative to the car), just like robot joints. Homogeneous transforms naturally handle these hierarchies.
4. Inverse Problems: Graphics uses inverse transforms for camera movement; robotics uses them for inverse kinematics (finding joint angles to reach a point).

Why Homogeneous Tricks Are Powerful

- Uniformity: One system (matrix multiplication) handles all transformations.
- Efficiency: Hardware (GPUs, controllers) can optimize matrix operations directly.
- Scalability: Works the same in 2D, 3D, or higher.
- Composability: Long pipelines are just products of matrices, avoiding special cases.

Applications

- Graphics: Rendering engines, VR/AR, CAD software, motion capture.
- Robotics: Arm manipulators, drones, autonomous vehicles, humanoid robots.
- Crossover: Simulation platforms use the same math to test robots and render virtual environments.

Try It Yourself

1. Build a 2D transformation pipeline: rotate a triangle, translate it, and project it into screen space. Write down the final transformation matrix.
2. Model a simple 2-joint robotic arm. Derive the forward kinematics using homogeneous matrices.
3. Implement a camera transform: place a cube at $(0, 0, 5)$, move the camera to $(0, 0, 0)$, and compute its 2D screen projection.
4. Show that composing a rotation and translation directly is equivalent to embedding them into a homogeneous matrix and multiplying.

Homogeneous coordinates are the hidden secret that lets graphics and robots share the same mathematical DNA. They unify how we move pixels, machines, and virtual worlds.

93. Graphs, Adjacency, and Laplacians (Networks via Matrices)

Linear algebra provides a powerful language for studying graphs-networks of nodes connected by edges. From social networks to electrical circuits, from the internet's structure to biological pathways, graphs appear everywhere. Matrices give graphs a numerical form, making it possible to analyze their structure using algebraic techniques.

Graph Basics Recap

- A graph $G = (V, E)$ has a set of vertices V (nodes) and edges E (connections).
- Graphs may be undirected or directed, weighted or unweighted.
- Many graph properties-connectivity, flow, clusters-can be studied through matrices.

The Adjacency Matrix

For a graph with n vertices, the adjacency matrix $A \in \mathbb{R}^{n \times n}$ encodes connections:

$$A_{ij} = \begin{cases} w_{ij}, & \text{if there is an edge from node } i \text{ to node } j \\ 0, & \text{otherwise} \end{cases}$$

- Unweighted graphs: entries are 0 or 1.
- Weighted graphs: entries are edge weights (distances, costs, capacities).
- Undirected graphs: A is symmetric.
- Directed graphs: A may be asymmetric.

The adjacency matrix is the algebraic fingerprint of the graph.

Powers of the Adjacency Matrix

The entry $(A^k)_{ij}$ counts the number of walks of length k from node i to node j.

- A^2 tells how many two-step connections exist.
- This property is used in algorithms for detecting paths, clustering, and network flow.

The Degree Matrix

The degree of a vertex is the number of edges connected to it (or the sum of weights in weighted graphs).

The degree matrix D is diagonal:

$$D_{ii} = \sum_j A_{ij}.$$

This matrix measures how "connected" each node is.

The Graph Laplacian

The combinatorial Laplacian is defined as:

$$L = D - A.$$

Key properties:

- L is symmetric (for undirected graphs).
- Each row sums to zero.
- The smallest eigenvalue is always 0, with eigenvector $[1, 1, ..., 1]^T$.

The Laplacian encodes connectivity: if the graph splits into k connected components, then L has exactly k zero eigenvalues.

Normalized Laplacians

Two common normalized versions are:

$$L_{sym} = D^{-1/2} L D^{-1/2}, \quad L_{rw} = D^{-1} L.$$

These rescale the Laplacian for applications like spectral clustering.

Spectral Graph Theory

Eigenvalues and eigenvectors of A or L reveal structure:

- Algebraic connectivity: The second-smallest eigenvalue of L measures how well connected the graph is.
- Spectral clustering: Eigenvectors of L partition graphs into communities.
- Random walks: Transition probabilities relate to $D^{-1}A$.

Example: A Simple Graph

Take a triangle graph with 3 nodes, each connected to the other two.

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}, \quad L = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}.$$

- Eigenvalues of L: $0, 3, 3$.
- The single zero eigenvalue confirms the graph is connected.

Applications

1. Community Detection: Spectral clustering finds natural divisions in social or biological networks.
2. Graph Drawing: Eigenvectors of L provide coordinates for visually embedding graphs.
3. Random Walks & PageRank: Transition matrices from adjacency define importance scores.
4. Physics: Laplacians appear in discrete versions of diffusion and vibration problems.
5. Machine Learning: Graph neural networks (GNNs) use Laplacians to propagate signals across graph structure.

Why It Matters

Graphs and matrices are two sides of the same coin: one combinatorial, one algebraic. By turning a network into a matrix, linear algebra gives us access to the full toolbox of eigenvalues, norms, and factorizations, enabling deep insights into connectivity, flow, and structure.

Try It Yourself

1. Compute adjacency, degree, and Laplacian matrices for a square graph (4 nodes in a cycle). Find eigenvalues of L.
2. Prove that the Laplacian always has at least one zero eigenvalue.
3. Show that if a graph has k components, then the multiplicity of zero as an eigenvalue is exactly k.
4. For a random walk on a graph, derive the transition matrix $P = D^{-1}A$. Interpret its eigenvectors.

Graphs demonstrate how linear algebra stretches beyond geometry and data tables-it becomes a universal language for networks, from molecules to megacities.

94. Data Preprocessing as Linear Operations (Centering, Whitening, Scaling)

Before any sophisticated model can be trained, raw data must be preprocessed. Surprisingly, many of the most common preprocessing steps-centering, scaling, whitening-are nothing more than linear algebra operations in disguise. Understanding them this way not only clarifies why they work, but also shows how they connect to broader concepts like covariance, eigenvalues, and singular value decomposition.

The Nature of Preprocessing

Most datasets are stored as a matrix: rows correspond to samples (observations) and columns correspond to features (variables). For instance, in a dataset of 1,000 people with height, weight, and age recorded, we'd have a 1000×3 matrix. Linear algebra allows us to systematically reshape, scale, and rotate this matrix to prepare it for downstream analysis.

Centering: Shifting the Origin

Centering means subtracting the mean of each column (feature) from all entries in that column.

$$X_{centered} = X - \mathbf{1}\mu^T$$

- Here X is the data matrix, μ is the vector of column means, and $\mathbf{1}$ is a column of ones.
- Effect: moves the dataset so that each feature has mean zero.
- Geometric view: translates the cloud of points so its "center of mass" sits at the origin.
- Why important: covariance and correlation formulas assume data are mean-centered; otherwise, cross-terms are skewed.

Example: If people's heights average 170 cm, subtract 170 from every height. After centering, "height = 0" corresponds to the average person.

Scaling: Normalizing Variability

Raw features can have different units or magnitudes (e.g., weight in kg, income in thousands of dollars). To compare them fairly, we scale:

$$X_{scaled} = XD^{-1}$$

where D is a diagonal matrix of feature standard deviations.

- Each feature now has variance 1.
- Geometric view: rescales axes so all dimensions have equal "spread."
- Common in machine learning: ensures gradient descent does not disproportionately focus on features with large raw values.

Example: If weight varies around 60 kg $\pm$ 15, dividing by 15 makes its spread comparable to that of height ($\pm$10 cm).

Whitening: Removing Correlations

Even after centering and scaling, features can remain correlated (e.g., height and weight). Whitening transforms the data so features become uncorrelated with unit variance.

- Let $\Sigma = \frac{1}{n}X^TX$ be the covariance matrix of centered data.
- Perform eigendecomposition: $\Sigma = Q\Lambda Q^T$.
- Whitening transform:

$$X_{white} = XQ\Lambda^{-1/2}Q^T$$

Result:

1. The covariance matrix of X_{white} is the identity matrix.
2. Each new feature is a rotated combination of old features, with no redundancy.

Geometric view: whitening "spheres" the data cloud, turning an ellipse into a perfect circle.

Covariance Matrix as the Key Player

The covariance matrix itself arises naturally from preprocessing:

$$\Sigma = \frac{1}{n} X^T X \quad \text{(if } X \text{ is centered)}.$$

- Diagonal entries: variances of features.
- Off-diagonal entries: covariances, measuring linear relationships.
- Preprocessing operations (centering, scaling, whitening) are designed to reshape data so Σ becomes easier to interpret and more stable for learning algorithms.

Connections to PCA

- Centering is required before PCA, otherwise the first component just points to the mean.
- Scaling ensures PCA does not overweight large-variance features.
- Whitening is closely related to PCA itself: PCA diagonalizes the covariance, and whitening goes one step further by rescaling eigenvalues to unity.

Thus, PCA can be seen as a preprocessing pipeline plus an analysis step.

Practical Workflows

1. Centering and Scaling (Standardization): The default for many algorithms like logistic regression or SVM.
2. Whitening: Often used in signal processing (e.g., removing correlations in audio or images).
3. Batch Normalization in Deep Learning: A variant of centering + scaling applied layer by layer during training.
4. Whitening in Image Processing: Ensures features like pixel intensities are decorrelated, improving compression and recognition.

Worked Example

Suppose we have three features: height, weight, and age.

1. Raw data:
    - Mean height = 170 cm, mean weight = 65 kg, mean age = 35 years.
    - Variance differs widely: age varies less, weight more.
2. After centering:

- Mean of each feature is zero.
- A person of average height now has value 0 in that feature.

3. After scaling:

   - All features have unit variance.
   - Algorithms can treat age and weight equally.

4. After whitening:

   - Correlation between height and weight disappears.
   - Features become orthogonal directions in feature space.

Why It Matters

Without preprocessing, models may be misled by scale, units, or correlations. Preprocessing makes features comparable, balanced, and independent-a crucial condition for algorithms that rely on geometry (distances, angles, inner products).

In essence, preprocessing is the bridge from messy, real-world data to the clean structures linear algebra expects.

Try It Yourself

1. For a small dataset, compute the covariance matrix before and after centering. What changes?
2. Scale the dataset so each feature has unit variance. Check the new covariance.
3. Perform whitening via eigendecomposition and verify the covariance matrix becomes the identity.
4. Plot the data points in 2D before and after whitening. Notice how an ellipse becomes a circle.

Preprocessing through linear algebra shows that preparing data is not just housekeeping-it's a fundamental reshaping of the problem's geometry.

95. Linear Regression and Classification (From Model to Matrix)

Linear algebra provides the foundation for two of the most widely used tools in data science and applied statistics: linear regression (predicting continuous outcomes) and linear classification (separating categories). Both problems reduce to expressing data in matrix form and then applying linear operations to estimate parameters.

The Regression Setup

Suppose we want to predict an output $y \in \mathbb{R}^n$ from features collected in a data matrix $X \in \mathbb{R}^{n \times p}$, where:

- n = number of observations (samples).
- p = number of features (variables).

We assume a linear model:

$$y \approx X\beta,$$

where $\beta \in \mathbb{R}^p$ is the vector of coefficients (weights). Each entry of β tells us how much its feature contributes to the prediction.

The Normal Equations

We want to minimize the squared error:

$$\min_{\beta} \|y - X\beta\|^2.$$

Differentiating leads to the normal equations:

$$X^T X\beta = X^T y.$$

- If $X^T X$ is invertible:

$$\hat{\beta} = (X^T X)^{-1} X^T y.$$

- If not invertible (multicollinearity, too many features), we use the pseudoinverse via SVD:

$$\hat{\beta} = X^+ y.$$

Geometric Interpretation

- $X\beta$ is the projection of y onto the column space of X.
- The residual $r = y - X\hat{\beta}$ is orthogonal to all columns of X.
- This "closest fit" property is why regression is a projection problem.

Classification with Linear Models

Instead of predicting continuous outputs, sometimes we want to separate categories (e.g., spam vs. not spam).

- Linear classifier: decides based on the sign of a linear function.

$$\hat{y} = \text{sign}(w^T x + b).$$

- Geometric view: w defines a hyperplane in feature space. Points on one side are labeled positive, on the other side negative.
- Relation to regression: logistic regression replaces squared error with a log-likelihood loss, but still solves for weights via iterative linear-algebraic methods.

Multiclass Extension

- For k classes, we use a weight matrix $W \in \mathbb{R}^{p \times k}$.
- Prediction:

$$\hat{y} = \arg\max_j (XW)_{ij}.$$

- Each class has a column of W, and the classifier picks the column with the largest score.

Example: Predicting House Prices

- Features: size, number of rooms, distance to city center.
- Target: price.
- X = matrix of features, y = price vector.
- Regression solves for coefficients showing how strongly each factor influences price.

If we switch to classification (predicting "expensive" vs. "cheap"), we treat price as a label and solve for a hyperplane separating the two categories.

Computational Aspects

- Directly solving normal equations: $O(p^3)$ (matrix inversion).
- QR factorization: numerically more stable.
- SVD: best when X is ill-conditioned or rank-deficient.
- Modern libraries: exploit sparsity or use gradient-based methods for large datasets.

Connections to Other Topics

- Least Squares (Chapter 8): Regression is the canonical least-squares problem.
- SVD (Chapter 9): Pseudoinverse gives regression when columns are dependent.
- Regularization (Chapter 9): Ridge regression adds a penalty $\lambda\|\beta\|^2$ to improve stability.
- Classification (Chapter 10): Forms the foundation of more complex models like support vector machines and neural networks.

Why It Matters

Linear regression and classification show the direct link between linear algebra and real-world decisions. They combine geometry (projection, hyperplanes), algebra (solving systems), and computation (factorizations). Despite their simplicity, they remain indispensable: they are interpretable, fast, and often competitive with more complex models.

Try It Yourself

1. Given three features and five samples, construct X and y. Solve for β using the normal equations.
2. Show that residuals are orthogonal to all columns of X.
3. Write down a linear classifier separating two clusters of points in 2D. Sketch the separating hyperplane.
4. Explore what happens when two features are highly correlated (collinear). Use pseudoinverse to recover a stable solution.

Linear regression and classification are proof that linear algebra is not just abstract-it is the engine of practical prediction.

96. PCA in Practice (Dimensionality Reduction Workflow)

Principal Component Analysis (PCA) is one of the most widely used tools in applied linear algebra. At its heart, PCA identifies the directions (principal components) along which data varies the most, and then re-expresses the data in terms of those directions. In practice, PCA is not just a mathematical curiosity-it is a complete workflow for reducing dimensionality, denoising data, and extracting patterns from high-dimensional datasets.

The Motivation

Modern datasets often have thousands or even millions of features:

- Images: each pixel is a feature.
- Genomics: each gene expression level is a feature.
- Text: each word in a vocabulary becomes a dimension.

Working in such high dimensions is expensive (computationally) and fragile (noise accumulates). PCA provides a systematic way to reduce the feature space to a smaller set of dimensions that still captures most of the variability.

Step 1: Organizing the Data

We start with a data matrix $X \in \mathbb{R}^{n \times p}$:

- n: number of samples (observations).
- p: number of features (variables).

Each row is a sample; each column is a feature.

Centering is the first preprocessing step: subtract the mean of each column so the dataset has mean zero. This ensures that PCA describes variance rather than being biased by offsets.

$$X_{centered} = X - \mathbf{1}\mu^T$$

Step 2: Covariance Matrix

Next, compute the covariance matrix:

$$\Sigma = \frac{1}{n} X_{centered}^T X_{centered}.$$

- Diagonal entries: variance of each feature.
- Off-diagonal entries: how features co-vary.

The structure of Σ determines the directions of maximal variation in the data.

Step 3: Eigen-Decomposition or SVD

Two equivalent approaches:

1. Eigen-decomposition: Solve $\Sigma v = \lambda v$.
    - Eigenvectors v are the principal components.
    - Eigenvalues λ measure variance along those directions.
2. Singular Value Decomposition (SVD): Directly decompose the centered data matrix:

$$X_{centered} = U\Sigma V^T.$$

    - Columns of $V =$ principal directions.
    - Squared singular values correspond to variances.

SVD is preferred in practice for numerical stability and efficiency, especially when p is very large.

Step 4: Choosing the Number of Components

We order eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p$.

- Explained variance ratio:

$$\text{EVR}(k) = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}.$$

- We choose k such that EVR exceeds some threshold (e.g., 90–95%).
- This balances dimensionality reduction with information preservation.

Graphically, a scree plot shows eigenvalues, and we look for the "elbow" point where additional components add little variance.

Step 5: Projecting Data

Once we select k components, we project onto them:

$$X_{PCA} = X_{centered} V_k,$$

where V_k contains the top k eigenvectors.

Result:

- $X_{PCA} \in \mathbb{R}^{n \times k}$.
- Each row is now a k-dimensional representation of the original sample.

Worked Example: Face Images

Suppose we have a dataset of grayscale images, each 100×100 pixels ($p = 10,000$).

1. Center each pixel value.
2. Compute covariance across all images.
3. Find eigenvectors = eigenfaces. These are characteristic patterns like "glasses," "mouth shape," or "lighting direction."
4. Keep top 50 components. Each face is now represented as a 50-dimensional vector instead of 10,000.

This drastically reduces storage and speeds up recognition while keeping key features.

Practical Considerations

- Standardization: If features have different scales (e.g., age in years vs. income in thousands), we must scale them before PCA.
- Computational shortcuts: For very large p, it's often faster to compute PCA via truncated SVD on X directly.
- Noise filtering: Small eigenvalues often correspond to noise; truncating them denoises the dataset.
- Interpretability: Principal components are linear combinations of features. Sometimes these combinations are interpretable, sometimes not.

Connections to Other Concepts

- Whitening (Chapter 94): PCA followed by scaling eigenvalues to 1 is whitening.
- SVD (Chapter 9): PCA is essentially an application of SVD.
- Regression (Chapter 95): PCA can be used before regression to reduce collinearity among predictors (PCA regression).
- Machine learning pipelines: PCA is often used before clustering, classification, or neural networks.

Why It Matters

PCA turns raw, unwieldy data into a compact form without losing essential structure. It enables visualization (2D/3D plots of high-dimensional data), faster learning, and noise reduction. Many breakthroughs-from face recognition to gene expression analysis-rely on PCA as the first preprocessing step.

Try It Yourself

1. Take a dataset with 3 features. Manually compute covariance, eigenvalues, and eigenvectors.
2. Project the data onto the first two principal components and plot. Compare to the original 3D scatter.
3. Download an image dataset and apply PCA to compress it. Reconstruct the images with 10, 50, 100 components. Observe the trade-off between compression and fidelity.
4. Compute explained variance ratios and decide how many components to keep.

PCA is the bridge between raw data and meaningful representation: it reduces complexity while sharpening patterns. It shows how linear algebra can reveal hidden order in high-dimensional chaos.

97. Recommender Systems and Low-Rank Models (Fill the Missing Entries)

Recommender systems-such as those used by Netflix, Amazon, or Spotify-are built on the principle that preferences can be captured by low-dimensional structures hidden inside large, sparse data. Linear algebra gives us the machinery to expose and exploit these structures, especially through low-rank models.

The Matrix of Preferences

We begin with a user–item matrix $R \in \mathbb{R}^{m \times n}$:

- Rows represent users.
- Columns represent items (movies, books, songs).
- Entries R_{ij} store the rating (say 1–5 stars) or interaction (clicks, purchases).

In practice, most entries are missing-users rate only a small subset of items. The central challenge: predict the missing entries.

Why Low-Rank Structure?

Despite its size, R often lies close to a low-rank approximation:

$$R \approx UV^T$$

- $U \in \mathbb{R}^{m \times k}$: user factors.
- $V \in \mathbb{R}^{n \times k}$: item factors.
- $k \ll \min(m, n)$.

Here, each user and each item is represented in a shared latent feature space.

- Example: For movies, latent dimensions might capture "action vs. romance," "old vs. new," or "mainstream vs. indie."
- A user's preference vector in this space interacts with an item's feature vector to generate a predicted rating.

This factorization explains correlations: if you liked Movie A and B, and Movie C shares similar latent features, the system predicts you'll like C too.

Singular Value Decomposition (SVD) Approach

If R were complete (no missing entries), we could compute the SVD:

$$R = U\Sigma V^T.$$

- Keep the top k singular values to form a rank-k approximation.
- This captures the dominant patterns in user preferences.
- Geometric view: project the massive data cloud onto a smaller k-dimensional subspace where structure is clearer.

But real data is incomplete. That leads to matrix completion problems.

Matrix Completion

Matrix completion tries to infer missing entries of R by assuming low rank. The optimization problem is:

$$\min_X \ \text{rank}(X) \quad \text{s.t. } X_{ij} = R_{ij} \text{ for observed entries.}$$

Since minimizing rank is NP-hard, practical algorithms instead minimize the nuclear norm (sum of singular values) or use alternating minimization:

- Initialize U, V randomly.
- Iteratively solve for one while fixing the other.
- Converge to a low-rank factorization that fits the observed ratings.

Alternating Least Squares (ALS)

ALS is a standard approach:

1. Fix V, solve least squares for U.
2. Fix U, solve least squares for V.
3. Repeat until convergence.

Each subproblem is straightforward linear regression, solvable with normal equations or QR decomposition.

Stochastic Gradient Descent (SGD)

Another approach: treat each observed rating as a training sample. Update latent vectors by minimizing squared error:

$$\ell = (R_{ij} - u_i^T v_j)^2.$$

Iteratively adjust user vector u_i and item vector v_j along gradients. This scales well to huge datasets, making it common in practice.

Regularization

To prevent overfitting:

$$\ell = (R_{ij} - u_i^T v_j)^2 + \lambda(\|u_i\|^2 + \|v_j\|^2).$$

- Regularization shrinks factors, discouraging extreme values.
- Geometrically, it keeps latent vectors within a reasonable ball in feature space.

Cold Start Problem

- New users: Without ratings, u_i is unknown. Solutions: use demographic features or ask for a few initial ratings.
- New items: Similarly, items need side information (metadata, tags) to generate initial latent vectors.

This is where hybrid models combine matrix factorization with content-based features.

Example: Movie Ratings

Imagine 1,000 users and 5,000 movies.

- The raw R matrix has 5 million entries, but each user has rated only ~50 movies.
- Matrix completion with rank $k = 20$ recovers a dense approximation.
- Each user is represented by 20 latent "taste" factors; each movie by 20 latent "theme" factors.
- Prediction: the dot product of user and movie vectors.

Beyond Ratings: Implicit Feedback

In practice, systems often lack explicit ratings. Instead, they use:

- Views, clicks, purchases, skips.
- These signals are indirect but abundant.
- Factorization can handle them by treating interactions as weighted observations.

Connections to Other Linear Algebra Tools

- SVD (Chapter 9): The backbone of factorization methods.
- Pseudoinverse (Chapter 9): Useful when solving small regression subproblems in ALS.
- Conditioning (Chapter 9): Factorization stability depends on well-scaled latent factors.
- PCA (Chapter 96): PCA is essentially a low-rank approximation, so PCA and recommenders share the same mathematics.

Why It Matters

Recommender systems personalize the modern internet. Every playlist suggestion, book recommendation, or ad placement is powered by linear algebra hidden in a massive sparse matrix. Low-rank modeling shows how even incomplete, noisy data can be harnessed to reveal patterns of preference and behavior.

Try It Yourself

1. Take a small user–item matrix with missing entries. Apply rank-2 approximation via SVD to fill in gaps.
2. Implement one step of ALS: fix movie factors and update user factors with least squares.
3. Compare predictions with and without regularization. Notice how regularization stabilizes results.
4. Explore the cold-start problem: simulate a new user and try predicting preferences from minimal data.

Low-rank models reveal a powerful truth: behind the enormous variety of human choices lies a surprisingly small set of underlying patterns-and linear algebra is the key to uncovering them.

98. PageRank and Random Walks (Ranking with Eigenvectors)

PageRank, the algorithm that once powered Google's search engine dominance, is a striking example of how linear algebra and eigenvectors can measure importance in a network. At its core, it models the web as a graph and asks a simple question: if you randomly surf the web forever, which pages will you visit most often?

The Web as a Graph

- Each web page is a node.
- Each hyperlink is a directed edge.
- The adjacency matrix A encodes which pages link to which:

$$A_{ij} = 1 \quad \text{if page } j \text{ links to page } i.$$

Why columns instead of rows? Because links flow from source to destination, and PageRank naturally arises when analyzing column-stochastic transition matrices.

Transition Matrix

To model random surfing, we define a column-stochastic matrix P:

$$P_{ij} = \frac{1}{\text{outdeg}(j)} \quad \text{if } j \to i.$$

- Each column sums to 1.
- P_{ij} is the probability of moving from page j to page i.
- This defines a Markov chain: a random process where the next state depends only on the current one.

If a user is on page j, they pick one outgoing link uniformly at random.

Random Walk Interpretation

Imagine a web surfer moving page by page according to P. After many steps, the fraction of time spent on each page converges to a steady-state distribution vector π:

$$\pi = P\pi.$$

This is an eigenvector equation: π is the stationary eigenvector of P with eigenvalue 1.

- π_i is the long-run probability of being on page i.
- A higher π_i means greater importance.

The PageRank Adjustment: Teleportation

The pure random walk has problems:

1. Dead ends: Pages with no outgoing links trap the surfer.
2. Spider traps: Groups of pages linking only to each other hoard probability mass.

Solution: add a teleportation mechanism:

- With probability α (say 0.85), follow a link.
- With probability $1 - \alpha$, jump to a random page.

This defines the PageRank matrix:

$$M = \alpha P + (1 - \alpha)\frac{1}{n}ee^T,$$

where e is the all-ones vector.

- M is stochastic, irreducible, and aperiodic.
- By the Perron–Frobenius theorem, it has a unique stationary distribution π.

Solving the Eigenproblem

The PageRank vector π satisfies:

$$M\pi = \pi.$$

- Computing π directly via eigen-decomposition is infeasible for billions of pages.
- Instead, use power iteration: repeatedly multiply a vector by M until convergence.

This works because the largest eigenvalue is 1, and the method converges to its eigenvector.

Worked Example: A Tiny Web

Suppose 3 pages with links:

- Page 1 $\to$ Page 2
- Page 2 $\to$ Page 3
- Page 3 $\to$ Page 1 and Page 2

Adjacency matrix (columns = source):

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

Transition matrix:

$$P = \begin{bmatrix} 0 & 0 & 1/2 \\ 1 & 0 & 1/2 \\ 0 & 1 & 0 \end{bmatrix}.$$

With teleportation ($\alpha = 0.85$), we form M. Power iteration quickly converges to $\pi = [0.37, 0.34, 0.29]^T$. Page 1 is ranked highest.

Beyond the Web

Although born in search engines, PageRank's mathematics applies broadly:

- Social networks: Rank influential users by their connections.
- Citation networks: Rank scientific papers by how they are referenced.
- Biology: Identify key proteins in protein–protein interaction networks.
- Recommendation systems: Rank products or movies via link structures.

In each case, importance is defined not by how many connections a node has, but by the importance of the nodes that point to it.

Computational Challenges

- Scale: Billions of pages mean M cannot be stored fully; sparse matrix techniques are essential.
- Convergence: Power iteration may take hundreds of steps; preconditioning and parallelization speed it up.
- Personalization: Instead of uniform teleportation, adjust probabilities to bias toward user interests.

Why It Matters

PageRank illustrates a deep principle: importance emerges from connectivity. Linear algebra captures this by identifying the dominant eigenvector of a transition matrix. This idea-ranking nodes in a network by stationary distributions-has transformed search engines, social media, and science itself.

Try It Yourself

1. Construct a 4-page web graph and compute its PageRank manually with $\alpha = 0.85$.
2. Implement power iteration in Python or MATLAB for a small adjacency matrix.
3. Compare PageRank to simple degree counts. Notice how PageRank rewards links from important nodes more heavily.
4. Modify teleportation to bias toward a subset of pages (personalized PageRank). Observe how rankings change.

PageRank is not only a milestone in computer science history-it is a living example of how eigenvectors can capture global importance from local structure.

99. Numerical Linear Algebra Essentials (Floating Point, BLAS/LAPACK)

Linear algebra in theory is exact: numbers behave like real numbers, operations are deterministic, and results are precise. In practice, computations are carried out on computers, where numbers are represented in finite precision and algorithms must balance speed, accuracy, and stability. This intersection-numerical linear algebra-is what makes linear algebra usable at modern scales.

Floating-Point Representation

Real numbers cannot be stored exactly on a digital machine. Instead, they are approximated using the IEEE 754 floating-point standard.

- A floating-point number is stored as:

$$x = \pm(1.m_1m_2m_3\ldots) \times 2^e$$

  where m is the mantissa and e is the exponent.

- Single precision (float32): 32 bits $\rightarrow$ ~7 decimal digits of precision.
- Double precision (float64): 64 bits $\rightarrow$ ~16 decimal digits.
- Machine epsilon (ϵ): The smallest gap between 1 and the next representable number. For double precision, $\epsilon \approx 2.22 \times 10^{-16}$.

Implication: operations like subtraction of nearly equal numbers cause catastrophic cancellation, where significant digits vanish.

Conditioning of Problems

A linear algebra problem may be well-posed mathematically but still numerically difficult.

- The condition number of a matrix A:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|.$$

- If $\kappa(A)$ is large, small input errors cause large output errors.
- Example: solving $Ax = b$. With ill-conditioned A, the computed solution may be unstable even with perfect algorithms.

Geometric intuition: ill-conditioned matrices stretch vectors unevenly, so small perturbations in direction blow up under inversion.

Stability of Algorithms

- An algorithm is numerically stable if it controls the growth of errors from finite precision.
- Gaussian elimination with partial pivoting is stable; without pivoting, it may fail catastrophically.
- Orthogonal factorizations (QR, SVD) are usually more stable than elimination methods.

Numerical analysis focuses on designing algorithms that guarantee accuracy within a few multiples of machine epsilon.

Direct vs. Iterative Methods

1. Direct methods: Solve in a finite number of steps (e.g., Gaussian elimination, LU decomposition, Cholesky for positive definite systems).
    - Reliable for small/medium problems.
    - Complexity $\sim O(n^3)$.
2. Iterative methods: Generate successive approximations (e.g., Jacobi, Gauss–Seidel, Conjugate Gradient).
    - Useful for very large, sparse systems.
    - Complexity per iteration $\sim O(n^2)$ or less, often leveraging sparsity.

Matrix Factorizations in Computation

Many algorithms rely on factorizing a matrix once, then reusing it:

- LU decomposition: Efficient for solving multiple right-hand sides.
- QR factorization: Stable approach for least squares.
- SVD: Gold standard for ill-conditioned problems, though expensive.

These factorizations reduce repeated operations into structured, cache-friendly steps.

Sparse vs. Dense Computations

- Dense matrices: Most entries are nonzero. Use dense linear algebra packages like BLAS and LAPACK.
- Sparse matrices: Most entries are zero. Store only nonzeros, use specialized algorithms to avoid wasted computation.

Large-scale problems (e.g., finite element simulations, web graphs) are feasible only because of sparse methods.

BLAS and LAPACK: Standard Libraries

- BLAS (Basic Linear Algebra Subprograms): Defines kernels for vector and matrix operations (dot products, matrix–vector, matrix–matrix multiplication). Optimized BLAS implementations exploit cache, SIMD, and multi-core parallelism.
- LAPACK (Linear Algebra PACKage): Builds on BLAS to provide algorithms for solving systems, eigenvalue problems, SVD, etc. LAPACK is the backbone of many scientific computing environments (MATLAB, NumPy, Julia).
- MKL, OpenBLAS, cuBLAS: Vendor-specific implementations optimized for Intel CPUs, open-source systems, or NVIDIA GPUs.

These libraries make the difference between code that runs in minutes and code that runs in milliseconds.

Floating-Point Pitfalls

1. Accumulated round-off: Summing numbers of vastly different magnitudes may discard small contributions.
2. Loss of orthogonality: Repeated Gram–Schmidt orthogonalization without reorthogonalization may drift numerically.
3. Overflow/underflow: Extremely large/small numbers exceed representable range.
4. NaNs and Infs: Divide-by-zero or invalid operations propagate errors.

Mitigation: use numerically stable algorithms, scale inputs, and check condition numbers.

Parallel and GPU Computing

Modern numerical linear algebra thrives on parallelism:

- GPUs accelerate dense linear algebra with thousands of cores (cuBLAS, cuSOLVER).
- Distributed libraries (ScaLAPACK, PETSc, Trilinos) allow solving problems with billions of unknowns across clusters.
- Mixed precision methods: compute in float32 or even float16, then refine in float64, balancing speed and accuracy.

Applications in the Real World

- Engineering simulations: Structural mechanics, fluid dynamics rely on sparse solvers.
- Machine learning: Training deep networks depends on optimized BLAS for matrix multiplications.
- Finance: Risk models solve huge regression problems with factorized covariance matrices.
- Big data: Dimensionality reduction (PCA, SVD) requires large-scale, stable algorithms.

Why It Matters

Linear algebra in practice is about more than theorems: it's about turning abstract models into computations that run reliably on imperfect hardware. Numerical linear algebra provides the essential toolkit-floating-point understanding, conditioning analysis, stable algorithms, and optimized libraries-that ensures results are both fast and trustworthy.

Try It Yourself

1. Compute the condition number of a nearly singular matrix (e.g., $\begin{bmatrix} 1 & 1 \\ 1 & 1.0001 \end{bmatrix}$) and solve $Ax = b$. Compare results in single vs. double precision.
2. Implement Gaussian elimination with and without pivoting. Compare errors for ill-conditioned matrices.
3. Use NumPy with OpenBLAS to time large matrix multiplications; compare against a naive Python implementation.
4. Explore iterative solvers: implement Conjugate Gradient for a sparse symmetric positive definite system.

Numerical linear algebra is the bridge between mathematical elegance and computational reality. It teaches us that solving equations on a computer is not just about the equations-it's about the algorithms, representations, and hardware that bring them to life.

100. Capstone Problem Sets and Next Steps (A Roadmap to Mastery)

You've now walked through the major landmarks of linear algebra: vectors, matrices, systems, transformations, determinants, eigenvalues, orthogonality, SVD, and applications to data and networks. The journey doesn't end here. This last section is designed as a capstone, a way to tie things together and show you how to keep practicing, exploring, and deepening your understanding. Think of it as your "next steps" map.

Practicing the Basics Until They Feel Natural

Linear algebra may seem heavy at first, but the simplest drills build lasting confidence. Try solving a few systems of equations by hand using elimination, and notice how pivoting reveals where solutions exist-or don't. Write down a small matrix and practice multiplying it by a vector. This might feel mechanical, but it's how your intuition sharpens: every time you push numbers through the rules, you're learning how the algebra reshapes space.

Even a single concept, like the dot product, can teach a lot. Take two short vectors in the plane, compute their dot product, and then compare it to the cosine of the angle between them. Seeing algebra match geometry is what makes linear algebra come alive.

Moving Beyond Computation: Understanding Structures

Once you're comfortable with the mechanics, try reflecting on the bigger structures. What does it mean for a set of vectors to be a subspace? Can you tell whether a line through the origin is one? What about a line shifted off the origin? This is where the rules and axioms you've seen start to guide your reasoning.

Experiment with bases and coordinates: pick two different bases for the plane and see how a single point looks different depending on the "ruler" you're using. Write out the change-of-basis matrix and check that it transforms coordinates the way you expect. These exercises show that linear algebra isn't just about numbers-it's about perspective.

Bringing Ideas Together in Larger Problems

The real joy comes when different ideas collide. Suppose you have noisy data, like a scatter of points that should lie along a line. Try fitting a line using least squares. What you're really doing is projecting the data onto a subspace. Or take a small Markov chain, like a random walk around three or four nodes, and compute its long-term distribution. That steady state is an eigenvector in disguise. These integrative problems demonstrate how the topics you've studied connect.

Projects make this even more vivid. For example:

- In computer graphics, write simple code that rotates or reflects a shape using a matrix.
- In networks, use the Laplacian to identify clusters in a social graph of friends.
- In recommendation systems, factorize a small user–item table to predict missing ratings.

These aren't abstract puzzles-they show how linear algebra works in the real world.

Looking Ahead: Where Linear Algebra Leads You

By now you know that linear algebra is not an isolated subject; it's a foundation. The next steps depend on your interests.

If you enjoy computation, numerical linear algebra is the natural extension. It digs into how floating-point numbers behave on real machines, how to control round-off errors, and why some algorithms are more stable than others. You'll learn why Gaussian elimination with pivoting is safe while without pivoting it can fail, and why QR and SVD are trusted in sensitive applications.

If abstraction intrigues you, then abstract linear algebra opens the door. Here you'll move beyond $\mathbb{R}^n$ into general vector spaces: polynomials as vectors, functions as vectors, dual spaces, and eventually tensor products. These ideas power much of modern mathematics and physics.

If data excites you, statistics and machine learning are a natural path. Covariance matrices, principal component analysis, regression, and neural networks all rest on linear algebra. Understanding them deeply requires both the computation you've practiced and the geometric insights you've built.

And if your curiosity points toward the sciences, linear algebra is everywhere: in quantum mechanics, where states are vectors and operators are matrices; in engineering, where vibrations and control systems rely on eigenvalues; in computer graphics, where every rotation and projection is a linear transformation.

Why This Capstone Matters

This final step is less about new theorems and more about perspective. The problems you solve now-whether small drills or large projects-train you to see structure, not just numbers. The roadmap is open-ended, because linear algebra itself is open-ended: once you learn to see the world through its lens, you notice it everywhere, from the patterns in networks to the behavior of algorithms to the geometry of space.

Try It Yourself

1. Take a dataset you care about-maybe sports scores, songs you listen to, or spending records. Organize it as a matrix. Compute simple things: averages (centering), a regression line, maybe even principal components. See what structure you uncover.
2. Write a short program that solves systems of equations using elimination. Test it on well-behaved and nearly singular matrices. Notice how stability changes.
3. Draw a 2D scatterplot and fit a line with least squares. Plot the residuals. What does it mean geometrically that the residuals are orthogonal to the line?
4. Try explaining eigenvalues to a friend without formulas-just pictures and stories. Teaching it will make it real.

Linear algebra is both a tool and a way of thinking. You now have enough to stand on your own, but the road continues forward-into deeper math, into practical computation, and into the sciences that rely on these ideas every day. This capstone is an invitation: keep practicing, keep exploring, and let the structures of linear algebra sharpen the way you see the world.

Closing

```
From lines to the stars,
each problem bends, transforms, grows-
paths extend ahead.
```

Finale

A quiet closing, where lessons settle and the music of algebra carries on beyond the final page.

1. Quiet Reflection

```
Lessons intertwining,
the book rests, but vectors stretch-
silence holds their song.
```

2. Infinite Journey

```
One map now complete,
yet beyond each line and plane
new horizons call.
```

3. Structure and Growth

```
Roots beneath the ground,
branches weaving endless skies,
algebra takes flight.
```

4. Light After Study

```
Numbers fade to light,
patterns linger in the mind,
paths remain open.
```

5. Eternal Motion

```
Stillness finds its place,
transformations carry on,
movement without end.
```

6. Gratitude and Closure

```
Steps of thought complete,
spaces carved with gentle care,
thank you, wandering mind.
```

7. Future Echo

```
From shadows to form,
each question births new echoes-
the journey goes on.
```

8. Horizon Beyond

```
The book closes here,
yet the lines refuse to end,
they stretch toward the stars.
```

The LAB

Chapter 1. Vectors, scalars, and geometry

1. Scalars, Vectors, and Coordinate Systems

Let's get our hands dirty! This lab is about playing with the *building blocks* of linear algebra: scalars and vectors. Think of a scalar as just a plain number, like `3` or `-1.5`. A vector is a small list of numbers, which you can picture as an arrow in space.

We'll use Python (with NumPy) to explore them. Don't worry if this is your first time with NumPy - we'll go slowly.

Set Up Your Lab

```
import numpy as np
```

That's it - we're ready! NumPy is the main tool we'll use for linear algebra.

Step-by-Step Code Walkthrough

Scalars are just numbers.

```
a = 5       # a scalar
b = -2.5    # another scalar

print(a + b)   # add them
print(a * b)   # multiply them
```

```
2.5
-12.5
```

Vectors are lists of numbers.

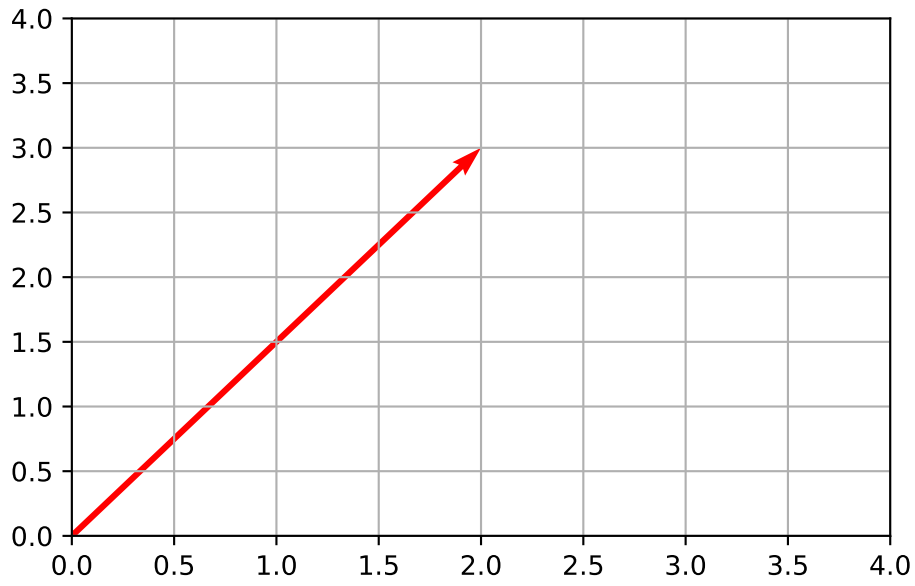
```
v = np.array([2, 3])       # a vector in 2D
w = np.array([1, -1, 4])  # a vector in 3D

print(v)
print(w)
```

```
[2 3]
[ 1 -1  4]
```

Coordinates tell us where we are. Think of `[2, 3]` as "go 2 steps in the x-direction, 3 steps in the y-direction."

We can even *draw* it:

```
import matplotlib.pyplot as plt

# plot vector v
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.xlim(0, 4)
plt.ylim(0, 4)
plt.grid()
plt.show()
```



This makes a little arrow from the origin `(0,0)` to `(2,3)`.

Try It Yourself

1. Change the vector `v` to `[4, 1]`. Where does the arrow point now?
2. Try making a 3D vector with 4 numbers, like `[1, 2, 3, 4]`. What happens?
3. Replace `np.array([2,3])` with `np.array([0,0])`. What does the arrow look like?

2. Vector Notation, Components, and Arrows

In this lab, we'll practice reading, writing, and visualizing vectors in different ways. A vector can look simple at first - just a list of numbers - but how we *write* it and how we *interpret* it really matters. This is where notation and components come into play.

A vector has:

- A symbol (we might call it `v`, `w`, or even `→AB` in geometry).
- Components (the individual numbers, like `2` and `3` in `[2, 3]`).
- An arrow picture (a geometric way to see the vector as a directed line segment).

Let's see all three in action with Python.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Writing vectors in Python

```
# Two-dimensional vector
v = np.array([2, 3])

# Three-dimensional vector
w = np.array([1, -1, 4])

print("v =", v)
print("w =", w)
```

```
v = [2 3]
w = [ 1 -1  4]
```

Here `v` has components `(2, 3)` and `w` has components `(1, -1, 4)`.

2. Accessing components Each number in the vector is a *component*. We can pick them out using indexing.

```
print("First component of v:", v[0])
print("Second component of v:", v[1])
```

```
First component of v: 2
Second component of v: 3
```

Notice: in Python, indices start at `0`, so `v[0]` is the *first* component.

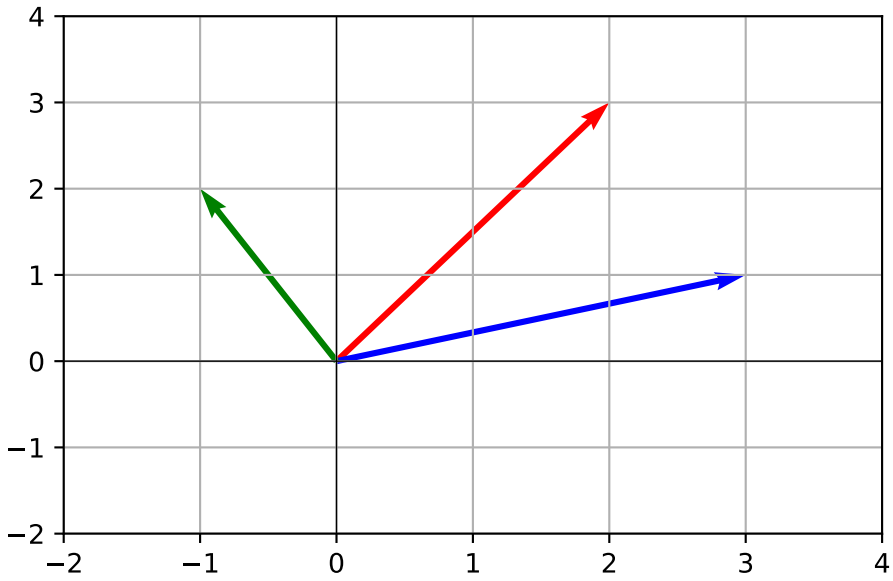
3. Visualizing vectors as arrows In 2D, it's easy to draw a vector from the origin `(0,0)` to its endpoint `(x,y)`.

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.xlim(-1, 4)
plt.ylim(-2, 4)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```

This shows vector v as a red arrow from `(0,0)` to `(2,3)`.

4. Drawing multiple vectors We can plot several arrows at once to compare them.

```
u = np.array([3, 1])
z = np.array([-1, 2])

# Draw v, u, z in different colors
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r', label='v')
plt.quiver(0, 0, u[0], u[1], angles='xy', scale_units='xy', scale=1, color='b', label='u')
plt.quiver(0, 0, z[0], z[1], angles='xy', scale_units='xy', scale=1, color='g', label='z')

plt.xlim(-2, 4)
plt.ylim(-2, 4)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



Now you'll see three arrows starting at the same point, each pointing in a different direction.

Try It Yourself

1. Change `v` to `[5, 0]`. What does the arrow look like now?
2. Try a vector like `[0, -3]`. Which axis does it line up with?
3. Make a new vector `q = np.array([2, 0, 0])`. What happens if you try to plot it with `plt.quiver` in 2D?

3. Vector Addition and Scalar Multiplication

In this lab, we'll explore the two most fundamental operations you can perform with vectors: adding them together and scaling them by a number (a scalar). These operations form the basis of everything else in linear algebra, from geometry to machine learning. Understanding how they work, both in code and visually, is key to building intuition.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Vector addition When you add two vectors, you simply add their components one by one.

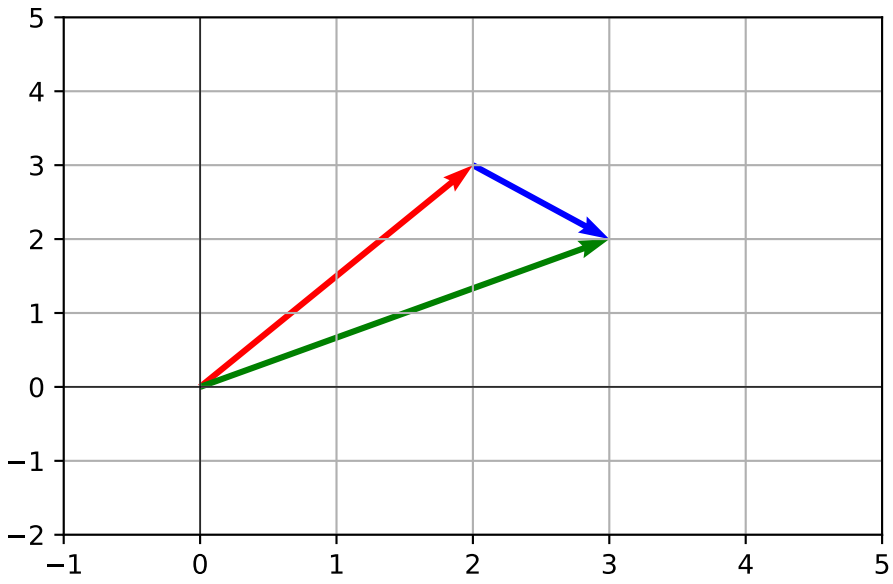
```
v = np.array([2, 3])
u = np.array([1, -1])

sum_vector = v + u
print("v + u =", sum_vector)
```

```
v + u = [3 2]
```

Here, `(2,3) + (1,-1) = (3,2)`.

2. Visualizing vector addition (tip-to-tail method) Graphically, vector addition means placing the tail of one vector at the head of the other. The resulting vector goes from the start of the first to the end of the second.

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r', label='v')
plt.quiver(v[0], v[1], u[0], u[1], angles='xy', scale_units='xy', scale=1, color='b', label=
plt.quiver(0, 0, sum_vector[0], sum_vector[1], angles='xy', scale_units='xy', scale=1, color=

plt.xlim(-1, 5)
plt.ylim(-2, 5)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



The green arrow is the result of adding `v` and `u`.

3. Scalar multiplication Multiplying a vector by a scalar stretches or shrinks it. If the scalar is negative, the vector flips direction.

```
c = 2
scaled_v = c * v
print("2 * v =", scaled_v)

d = -1
scaled_v_neg = d * v
print("-1 * v =", scaled_v_neg)
```

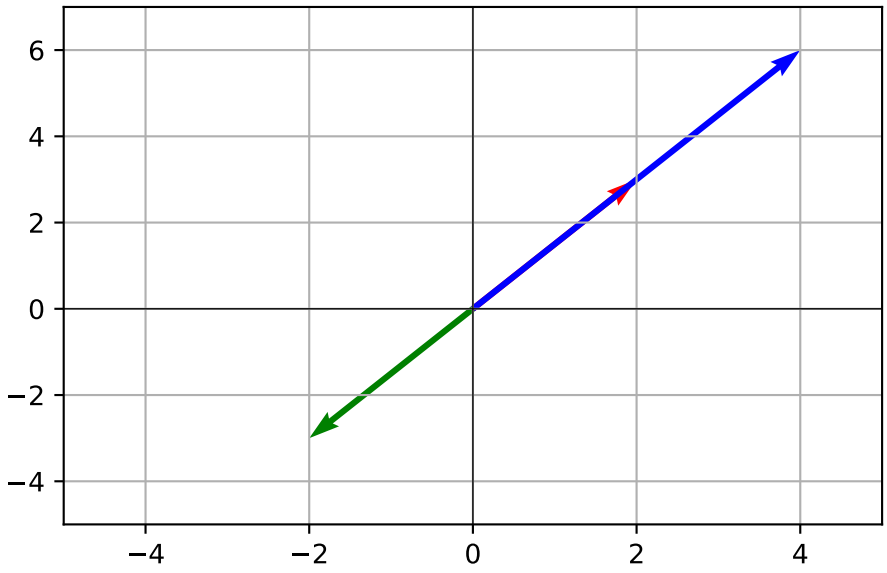
```
2 * v = [4 6]
-1 * v = [-2 -3]
```

So `2 * (2,3) = (4,6)` and `-1 * (2,3) = (-2,-3)`.

4. Visualizing scalar multiplication

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r', label='v')
plt.quiver(0, 0, scaled_v[0], scaled_v[1], angles='xy', scale_units='xy', scale=1, color='b'
plt.quiver(0, 0, scaled_v_neg[0], scaled_v_neg[1], angles='xy', scale_units='xy', scale=1, c
```

```
plt.xlim(-5, 5)
plt.ylim(-5, 7)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



Here, the blue arrow is twice as long as the red arrow, while the green arrow points in the opposite direction.

5. Combining both operations We can scale vectors and then add them. This is called a linear combination (and it's the foundation for the next section).

```
combo = 3*v + (-2)*u
print("3*v - 2*u =", combo)
```

```
3*v - 2*u = [ 4 11]
```

Try It Yourself

1. Replace `c = 2` with `c = 0.5`. What happens to the vector?
2. Try adding three vectors: `v + u + np.array([-1,2])`. Can you predict the result before printing?
3. Visualize `3*v + 2*u` using arrows. How does it compare to just `v + u`?

4. Linear Combinations and Span

Now that we know how to add vectors and scale them, we can combine these two moves to create linear combinations. A linear combination is just a recipe: multiply vectors by scalars, then add them together. The set of all possible results you can get from such recipes is called the span.

This idea is powerful because span tells us what directions and regions of space we can reach using given vectors.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Linear combinations in Python

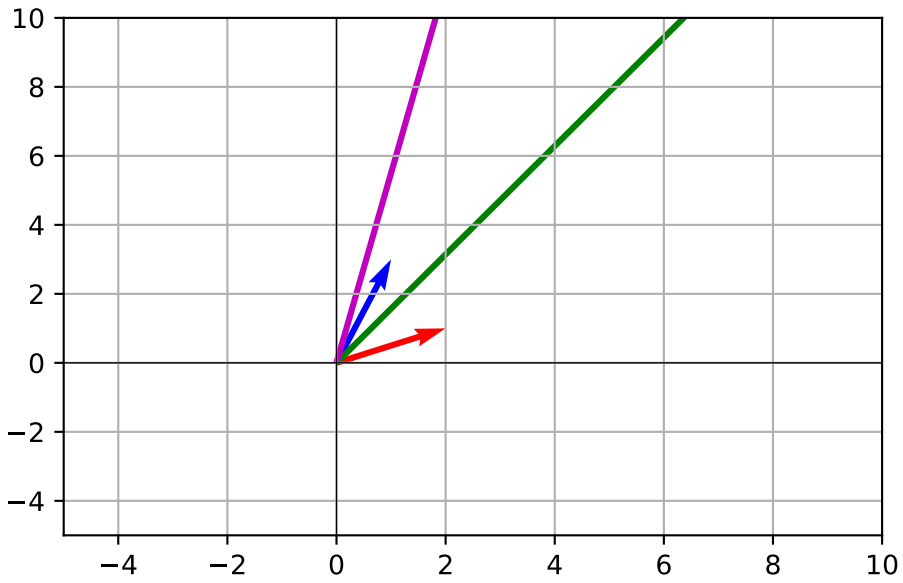
```
v = np.array([2, 1])
u = np.array([1, 3])

combo1 = 2*v + 3*u
combo2 = -1*v + 4*u

print("2*v + 3*u =", combo1)
print("-v + 4*u =", combo2)
```

```
2*v + 3*u = [ 7 11]
-v + 4*u = [ 2 11]
```

Here, we multiplied and added vectors using scalars. Each result is a new vector.

2. Visualizing linear combinations Let's plot `v`, `u`, and their combinations.

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r', label='v')
plt.quiver(0, 0, u[0], u[1], angles='xy', scale_units='xy', scale=1, color='b', label='u')
plt.quiver(0, 0, combo1[0], combo1[1], angles='xy', scale_units='xy', scale=1, color='g', lab
plt.quiver(0, 0, combo2[0], combo2[1], angles='xy', scale_units='xy', scale=1, color='m', lab

plt.xlim(-5, 10)
plt.ylim(-5, 10)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



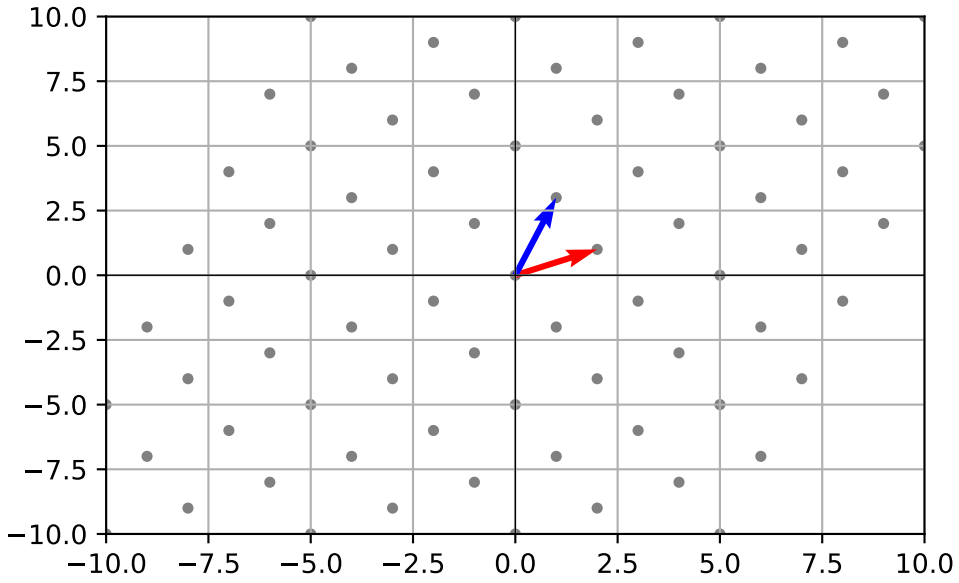
This shows how new arrows can be generated from scaling and adding the original ones.

3. Exploring the span The span of two 2D vectors is either:

- A line (if one is a multiple of the other).
- The whole 2D plane (if they are independent).

```
# Generate many combinations
coeffs = range(-5, 6)
points = []
for a in coeffs:
    for b in coeffs:
        point = a*v + b*u
```

```
        points.append(point)

points = np.array(points)

plt.scatter(points[:,0], points[:,1], s=10, color='gray')
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.quiver(0, 0, u[0], u[1], angles='xy', scale_units='xy', scale=1, color='b')

plt.xlim(-10, 10)
plt.ylim(-10, 10)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```

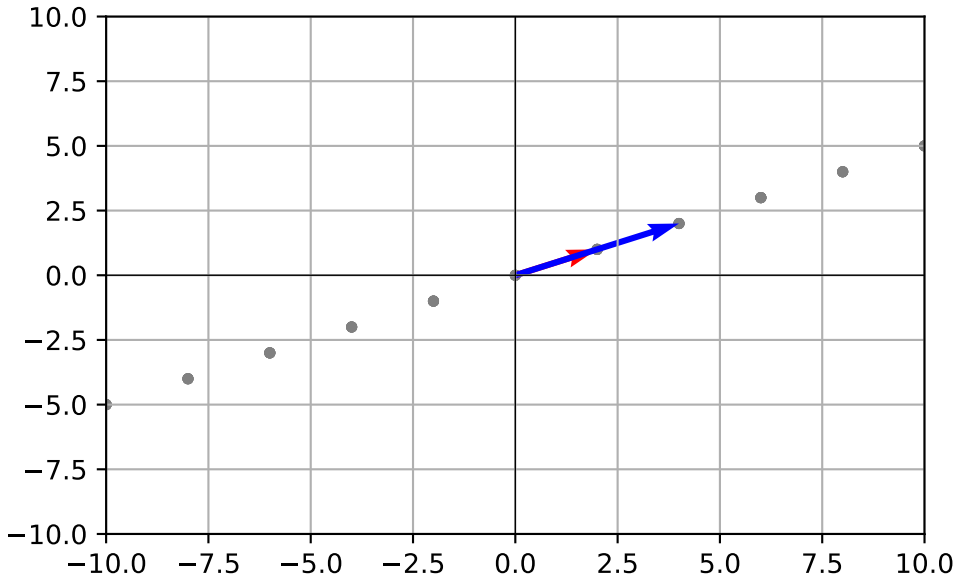


The gray dots show all reachable points with combinations of `v` and `u`.

4. Special case: dependent vectors

```
w = np.array([4, 2])  # notice w = 2*v
coeffs = range(-5, 6)
points = []
for a in coeffs:
    for b in coeffs:
```

```
        points.append(a*v + b*w)

points = np.array(points)

plt.scatter(points[:,0], points[:,1], s=10, color='gray')
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.quiver(0, 0, w[0], w[1], angles='xy', scale_units='xy', scale=1, color='b')

plt.xlim(-10, 10)
plt.ylim(-10, 10)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



Here, the span collapses to a line because `w` is just a scaled copy of `v`.

Try It Yourself

1. Replace `u = [1,3]` with `u = [-1,2]`. What does the span look like?
2. Try three vectors in 2D (e.g., `v, u, w`). Do you get more than the whole plane?
3. Experiment with 3D vectors. Use `np.array([x,y,z])` and check whether different vectors span a plane or all of space.

5. Length (Norm) and Distance

In this lab, we'll measure how big a vector is (its length, also called its norm) and how far apart two vectors are (their distance). These ideas connect algebra to geometry: when we compute a norm, we're measuring the size of an arrow; when we compute a distance, we're measuring the gap between two points in space.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Vector length (norm) in 2D The length of a vector is computed using the Pythagorean theorem. For a vector `(x, y)`, the length is `sqrt(x² + y²)`.

```
v = np.array([3, 4])
length = np.linalg.norm(v)
print("Length of v =", length)
```

```
Length of v = 5.0
```

This prints `5.0`, because `(3,4)` forms a right triangle with sides 3 and 4, and `sqrt(3²+4²)=5`.

2. Manual calculation vs NumPy

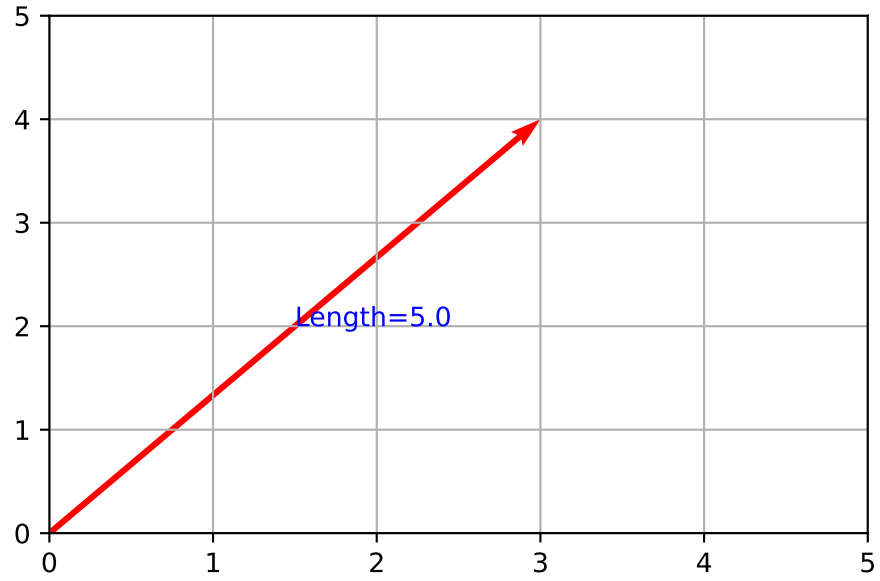
```
manual_length = (v[0]**2 + v[1]**2)**0.5
print("Manual length =", manual_length)
print("NumPy length =", np.linalg.norm(v))
```

```
Manual length = 5.0
NumPy length = 5.0
```

Both give the same result.

3. Visualizing vector length

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.text(v[0]/2, v[1]/2, f"Length={length}", fontsize=10, color='blue')
plt.grid()
plt.show()
```



You'll see the arrow `(3,4)` with its length labeled.

4. Distance between two vectors The distance between `v` and another vector `u` is the length of their difference: `||v - u||`.

```
u = np.array([0, 0])   # the origin
dist = np.linalg.norm(v - u)
print("Distance between v and u =", dist)
```

```
Distance between v and u = 5.0
```

Since `u` is the origin, this is just the length of `v`.

5. A more interesting distance

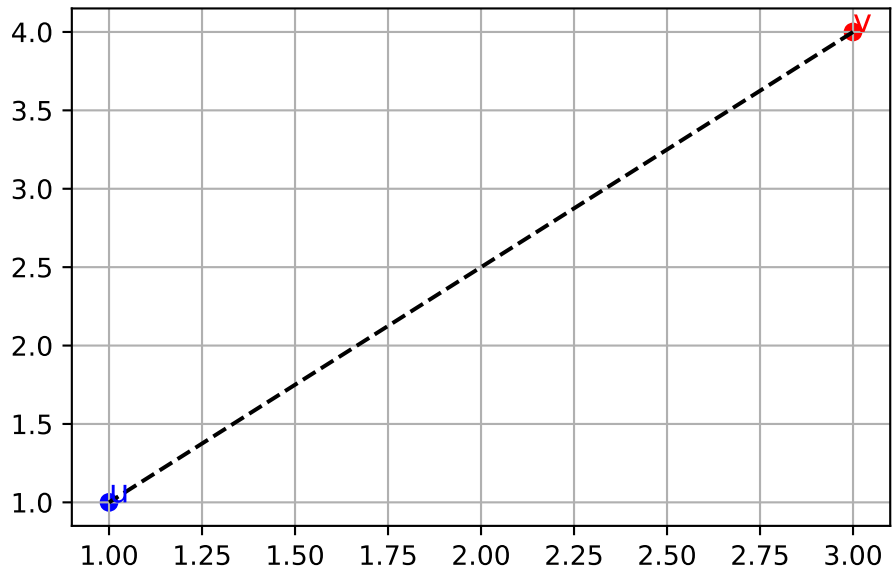
```
u = np.array([1, 1])
dist = np.linalg.norm(v - u)
print("Distance between v and u =", dist)
```

```
Distance between v and u = 3.605551275463989
```

This measures how far `(3,4)` is from `(1,1)`.

6. Visualizing distance between points

```
plt.scatter([v[0], u[0]], [v[1], u[1]], color=['red','blue'])
plt.plot([v[0], u[0]], [v[1], u[1]], 'k--')
plt.text(v[0], v[1], 'v', fontsize=12, color='red')
plt.text(u[0], u[1], 'u', fontsize=12, color='blue')
plt.grid()
plt.show()
```



The dashed line shows the distance between the two points.

7. Higher-dimensional vectors Norms and distances work the same way in any dimension:

```
a = np.array([1,2,3])
b = np.array([4,0,8])
print("‖a‖ =", np.linalg.norm(a))
print("‖b‖ =", np.linalg.norm(b))
print("Distance between a and b =", np.linalg.norm(a-b))
```

```
‖a‖ = 3.7416573867739413
‖b‖ = 8.94427190999916
Distance between a and b = 6.164414002968976
```

Even though we can't draw 3D easily on paper, the formulas still apply.

Try It Yourself

1. Compute the length of `np.array([5,12])`. What do you expect?
2. Find the distance between `(2,3)` and `(7,7)`. Can you sketch it by hand and check?
3. In 3D, try vectors `(1,1,1)` and `(2,2,2)`. Why is the distance exactly `sqrt(3)`?

6. Dot Product

The dot product is one of the most important operations in linear algebra. It takes two vectors and gives you a single number. That number combines both the lengths of the vectors and how much they point in the same direction. In this lab, we'll calculate dot products in several ways, see how they relate to geometry, and visualize their meaning.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Algebraic definition The dot product of two vectors is the sum of the products of their components:

```
v = np.array([2, 3])
u = np.array([4, -1])

dot_manual = v[0]*u[0] + v[1]*u[1]
dot_numpy = np.dot(v, u)

print("Manual dot product:", dot_manual)
print("NumPy dot product:", dot_numpy)
```

```
Manual dot product: 5
NumPy dot product: 5
```

Here, `(2*4) + (3*-1) = 8 - 3 = 5`.

2. Geometric definition The dot product also equals the product of the lengths of the vectors times the cosine of the angle between them:

$$v \cdot u = \|v\|\|u\| \cos\theta$$

We can compute the angle:

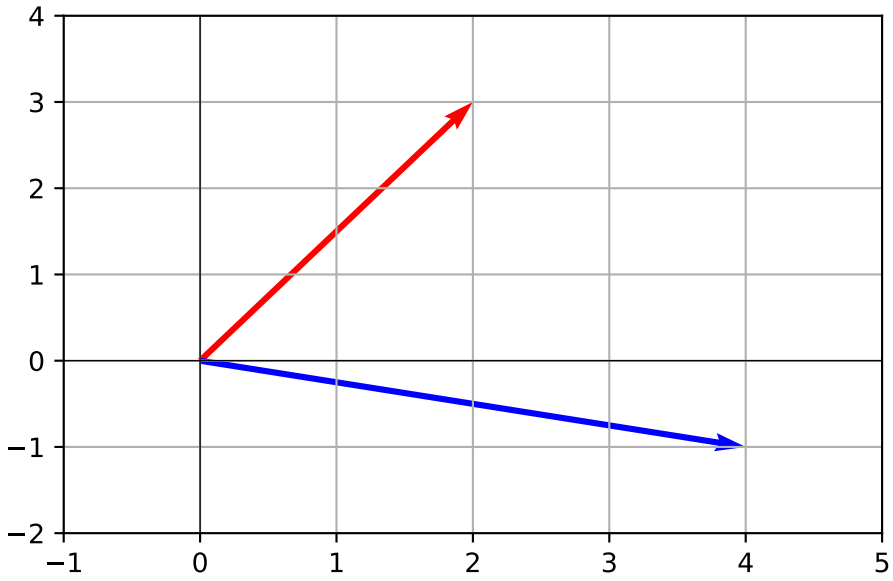
```
norm_v = np.linalg.norm(v)
norm_u = np.linalg.norm(u)

cos_theta = np.dot(v, u) / (norm_v * norm_u)
theta = np.arccos(cos_theta)

print("cos(theta) =", cos_theta)
print("theta (in radians) =", theta)
print("theta (in degrees) =", np.degrees(theta))
```

```
cos(theta) = 0.33633639699815626
theta (in radians) = 1.2277723863741932
theta (in degrees) = 70.3461759419467
```

This gives the angle between `v` and `u`.

3. Visualizing the dot product Let's draw the two vectors:

```
plt.quiver(0,0,v[0],v[1],angles='xy',scale_units='xy',scale=1,color='r',label='v')
plt.quiver(0,0,u[0],u[1],angles='xy',scale_units='xy',scale=1,color='b',label='u')
plt.xlim(-1,5)
plt.ylim(-2,4)
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.show()
```



The dot product is positive if the angle is less than 90°, negative if greater than 90°, and zero if the vectors are perpendicular.

4. Projections and dot product The dot product lets us compute how much of one vector lies in the direction of another.

```
proj_length = np.dot(v, u) / np.linalg.norm(u)
print("Projection length of v onto u:", proj_length)
```

```
Projection length of v onto u: 1.212678125181665
```

This is the length of the shadow of `v` onto `u`.

5. Special cases

- If vectors point in the same direction, the dot product is large and positive.
- If vectors are perpendicular, the dot product is zero.
- If vectors point in opposite directions, the dot product is negative.

```
a = np.array([1,0])
b = np.array([0,1])
c = np.array([-1,0])

print("a · b =", np.dot(a,b))   # perpendicular
print("a · a =", np.dot(a,a))   # length squared
print("a · c =", np.dot(a,c))   # opposite
```

```
a · b = 0
a · a = 1
a · c = -1
```

Try It Yourself

1. Compute the dot product of `(3,4)` with `(4,3)`. Is the result larger or smaller than the product of their lengths?
2. Try `(1,2,3) · (4,5,6)`. Does the geometric meaning still work in 3D?
3. Create two perpendicular vectors (e.g. `(2,0)` and `(0,5)`). Verify the dot product is zero.

7. Angles Between Vectors and Cosine

In this lab, we'll go deeper into the connection between vectors and geometry by calculating angles. Angles tell us how much two vectors "point in the same direction." The bridge between algebra and geometry here is the cosine formula, which comes directly from the dot product.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Formula for the angle The angle θ between two vectors v and u is given by:

$$\cos\theta = \frac{v \cdot u}{\|v\|\,\|u\|}$$

This means:

- If $\cos\theta = 1$, the vectors point in exactly the same direction.
- If $\cos\theta = 0$, they are perpendicular.
- If $\cos\theta = -1$, they point in opposite directions.

2. Computing the angle in Python

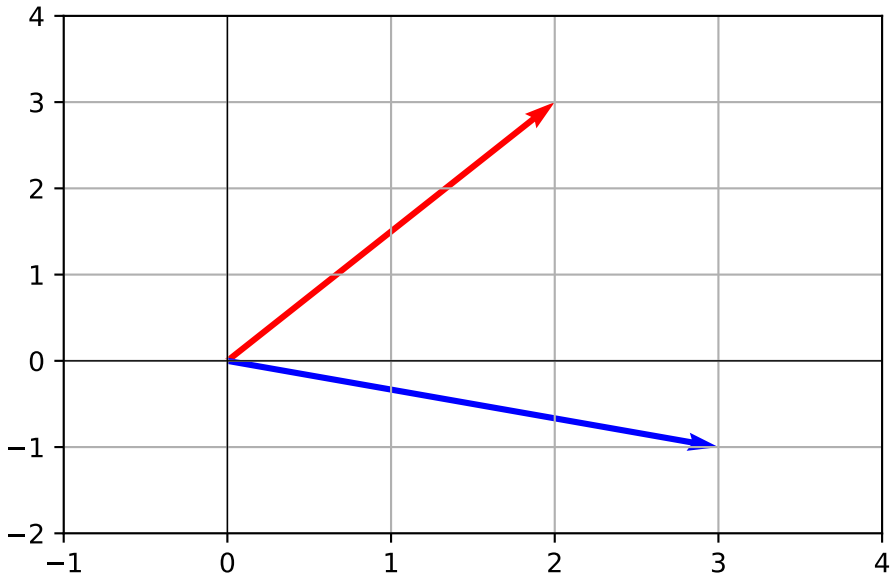
```
v = np.array([2, 3])
u = np.array([3, -1])

dot = np.dot(v, u)
norm_v = np.linalg.norm(v)
norm_u = np.linalg.norm(u)

cos_theta = dot / (norm_v * norm_u)
theta = np.arccos(cos_theta)

print("cos(theta) =", cos_theta)
print("theta in radians =", theta)
print("theta in degrees =", np.degrees(theta))
```

```
cos(theta) = 0.2631174057921088
theta in radians = 1.3045442776439713
theta in degrees = 74.74488129694222
```

This gives both the cosine value and the actual angle.

3. Visualizing the vectors

```
plt.quiver(0,0,v[0],v[1],angles='xy',scale_units='xy',scale=1,color='r',label='v')
plt.quiver(0,0,u[0],u[1],angles='xy',scale_units='xy',scale=1,color='b',label='u')

plt.xlim(-1,4)
plt.ylim(-2,4)
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.show()
```



You can see the angle between v and u as the gap between the red and blue arrows.

4. Checking special cases

```
a = np.array([1,0])
b = np.array([0,1])
c = np.array([-1,0])

print("Angle between a and b =", np.degrees(np.arccos(np.dot(a,b)/(np.linalg.norm(a)*np.linal
print("Angle between a and c =", np.degrees(np.arccos(np.dot(a,c)/(np.linalg.norm(a)*np.linal
```

```
Angle between a and b = 90.0
Angle between a and c = 180.0
```

- Angle between `(1,0)` and `(0,1)` is 90°.
- Angle between `(1,0)` and `(-1,0)` is 180°.

5. Using cosine as a similarity measure In data science and machine learning, people often use cosine similarity instead of raw angles. It's just the cosine value itself:

```
cosine_similarity = np.dot(v,u)/(np.linalg.norm(v)*np.linalg.norm(u))
print("Cosine similarity =", cosine_similarity)
```

```
Cosine similarity = 0.2631174057921088
```

Values close to `1` mean vectors are aligned, values near `0` mean unrelated, and values near `-1` mean opposite.

Try It Yourself

1. Create two random vectors with `np.random.randn(3)` and compute the angle between them.
2. Verify that swapping the vectors gives the same angle (symmetry).
3. Find two vectors where cosine similarity is exactly `0`. Can you come up with an example in 2D?

8. Projections and Decompositions

In this lab, we'll learn how to split one vector into parts: one part that lies *along* another vector, and one part that is *perpendicular*. This process is called projection and decomposition. Projections let us measure "how much of a vector points in a given direction," and decompositions give us a way to break vectors into useful components.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Projection formula The projection of vector v onto vector u is:

$$\text{proj}_u(v) = \frac{v \cdot u}{u \cdot u} u$$

This gives the component of v that points in the direction of u.

2. Computing projection in Python

```
v = np.array([3, 2])
u = np.array([2, 0])

proj_u_v = (np.dot(v, u) / np.dot(u, u)) * u
print("Projection of v onto u:", proj_u_v)
```

```
Projection of v onto u: [3. 0.]
```

Here, $v = (3, 2)$ and $u = (2, 0)$. The projection of `v` onto `u` is a vector pointing along the x-axis.

3. Decomposing into parallel and perpendicular parts

We can write:

$$v = \text{proj}_u(v) + (v - \text{proj}_u(v))$$

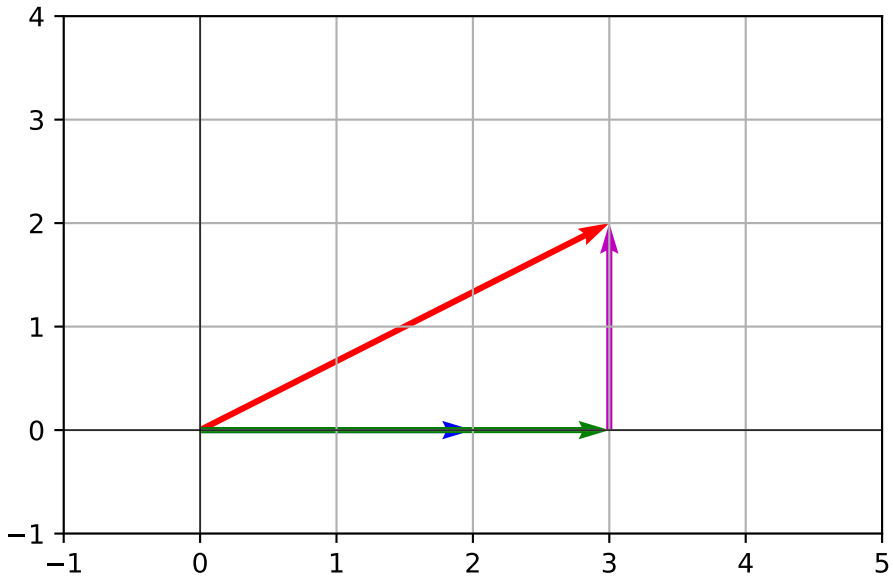
The first part is parallel to `u`, the second part is perpendicular.

```
perp = v - proj_u_v
print("Parallel part:", proj_u_v)
print("Perpendicular part:", perp)
```

```
Parallel part: [3. 0.]
Perpendicular part: [0. 2.]
```

4. Visualizing projection and decomposition

```
plt.quiver(0, 0, v[0], v[1], angles='xy', scale_units='xy', scale=1, color='r', label='v')
plt.quiver(0, 0, u[0], u[1], angles='xy', scale_units='xy', scale=1, color='b', label='u')
plt.quiver(0, 0, proj_u_v[0], proj_u_v[1], angles='xy', scale_units='xy', scale=1, color='g'
plt.quiver(proj_u_v[0], proj_u_v[1], perp[0], perp[1], angles='xy', scale_units='xy', scale=

plt.xlim(-1, 5)
plt.ylim(-1, 4)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid()
plt.show()
```



You'll see `v` (red), `u` (blue), the projection (green), and the perpendicular remainder (magenta).

5. Projection in higher dimensions

This formula works in any dimension:

```
a = np.array([1,2,3])
b = np.array([0,1,0])

proj = (np.dot(a,b)/np.dot(b,b)) * b
perp = a - proj
```

```
print("Projection of a onto b:", proj)
print("Perpendicular component:", perp)
```

```
Projection of a onto b: [0. 2. 0.]
Perpendicular component: [1. 0. 3.]
```

Even in 3D or higher, projections are about splitting into "along" and "across."

Try It Yourself

1. Try projecting `(2,3)` onto `(0,5)`. Where does it land?
2. Take a 3D vector like `(4,2,6)` and project it onto `(1,0,0)`. What does this give you?
3. Change the base vector `u` to something not aligned with the axes, like `(1,1)`. Does the projection still work?

9. Cauchy–Schwarz and Triangle Inequalities

This lab introduces two fundamental inequalities in linear algebra. They may look abstract at first, but they provide guarantees that always hold true for vectors. We'll explore them with small examples in Python to see why they matter.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Cauchy–Schwarz inequality

The inequality states:

$$|v \cdot u| \leq \|v\| \, \|u\|$$

It means the dot product is never "bigger" than the product of the vector lengths. Equality happens only if the two vectors are pointing in exactly the same (or opposite) direction.

```
v = np.array([3, 4])
u = np.array([1, 2])

lhs = abs(np.dot(v, u))
rhs = np.linalg.norm(v) * np.linalg.norm(u)

print("Left-hand side (|v·u|):", lhs)
print("Right-hand side (‖v‖‖u‖):", rhs)
print("Inequality holds?", lhs <= rhs)
```

```
Left-hand side (|v·u|): 11
Right-hand side (‖v‖‖u‖): 11.180339887498949
Inequality holds? True
```

2. Testing Cauchy–Schwarz with different vectors

```
pairs = [
    (np.array([1,0]), np.array([0,1])),  # perpendicular
    (np.array([2,3]), np.array([4,6])),  # multiples
    (np.array([-1,2]), np.array([3,-6])) # opposite multiples
]

for v,u in pairs:
    lhs = abs(np.dot(v, u))
    rhs = np.linalg.norm(v) * np.linalg.norm(u)
    print(f"v={v}, u={u} -> |v·u|={lhs}, ‖v‖‖u‖={rhs}, holds={lhs<=rhs}")
```

```
v=[1 0], u=[0 1] -> |v·u|=0, ‖v‖‖u‖=1.0, holds=True
v=[2 3], u=[4 6] -> |v·u|=26, ‖v‖‖u‖=25.999999999999996, holds=False
v=[-1  2], u=[ 3 -6] -> |v·u|=15, ‖v‖‖u‖=15.000000000000002, holds=True
```

- Perpendicular vectors give `|v·u| = 0`, far less than the product of norms.
- Multiples give equality (`lhs = rhs`).

3. Triangle inequality

The triangle inequality states:

$$\|v + u\| \leq \|v\| + \|u\|$$

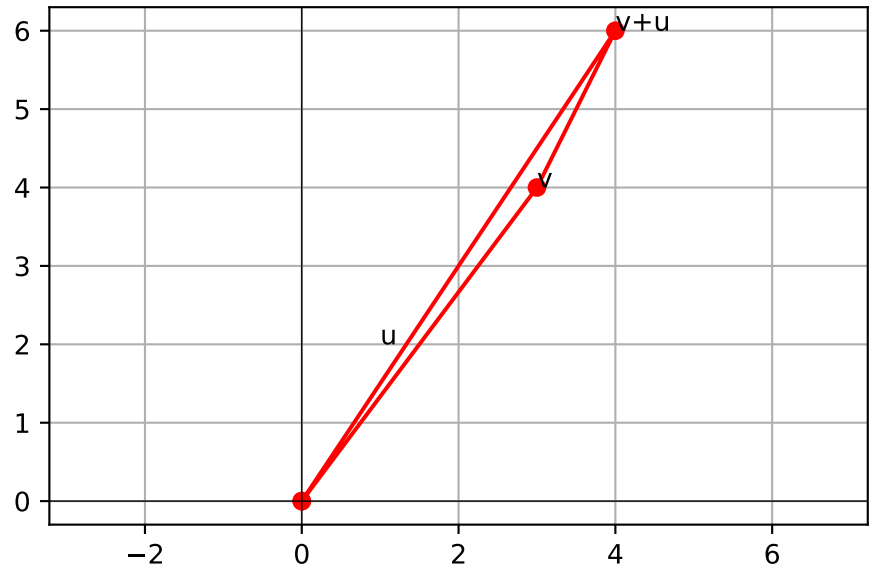
Geometrically, the length of one side of a triangle can never be longer than the sum of the other two sides.

```
v = np.array([3, 4])
u = np.array([1, 2])

lhs = np.linalg.norm(v + u)
rhs = np.linalg.norm(v) + np.linalg.norm(u)

print("‖v+u‖ =", lhs)
print("‖v‖ + ‖u‖ =", rhs)
print("Inequality holds?", lhs <= rhs)
```

```
‖v+u‖ = 7.211102550927978
‖v‖ + ‖u‖ = 7.23606797749979
Inequality holds? True
```

4. Visual demonstration with a triangle

```
import matplotlib.pyplot as plt

origin = np.array([0,0])
points = np.array([origin, v, v+u, origin])

plt.plot(points[:,0], points[:,1], 'ro-')  # triangle outline
plt.text(v[0], v[1], 'v')
plt.text(v[0]+u[0], v[1]+u[1], 'v+u')
plt.text(u[0], u[1], 'u')

plt.grid()
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.axis('equal')
plt.show()
```



This triangle shows why the inequality is called the “triangle” inequality.

5. Testing triangle inequality with random vectors

```
for _ in range(5):
    v = np.random.randn(2)
    u = np.random.randn(2)
    lhs = np.linalg.norm(v+u)
    rhs = np.linalg.norm(v) + np.linalg.norm(u)
    print(f"‖v+u‖={lhs:.3f}, ‖v‖+‖u‖={rhs:.3f}, holds={lhs <= rhs}")
```

```
‖v+u‖=2.500, ‖v‖+‖u‖=2.559, holds=True
‖v+u‖=1.872, ‖v‖+‖u‖=3.020, holds=True
‖v+u‖=2.892, ‖v‖+‖u‖=2.892, holds=True
‖v+u‖=3.578, ‖v‖+‖u‖=4.048, holds=True
‖v+u‖=1.472, ‖v‖+‖u‖=1.926, holds=True
```

No matter what vectors you try, the inequality always holds.

The Takeaway

- Cauchy–Schwarz: The dot product is always bounded by the product of vector lengths.
- Triangle inequality: The length of one side of a triangle can’t exceed the sum of the other two.

- These inequalities form the backbone of geometry, analysis, and many proofs in linear algebra.

10. Orthonormal Sets in $^2/^3$

In this lab, we'll explore orthonormal sets - collections of vectors that are both orthogonal (perpendicular) and normalized (length = 1). These sets are the "nicest" possible bases for vector spaces. In 2D and 3D, they correspond to the coordinate axes we already know, but we can also construct and test new ones.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Orthogonal vectors Two vectors are orthogonal if their dot product is zero.

```
x_axis = np.array([1, 0])
y_axis = np.array([0, 1])

print("x_axis · y_axis =", np.dot(x_axis, y_axis))  # should be 0
```

```
x_axis · y_axis = 0
```

So the standard axes are orthogonal.

2. Normalizing vectors Normalization means dividing a vector by its length to make its norm equal to 1.

```
v = np.array([3, 4])
v_normalized = v / np.linalg.norm(v)

print("Original v:", v)
print("Normalized v:", v_normalized)
print("Length of normalized v:", np.linalg.norm(v_normalized))
```

```
Original v: [3 4]
Normalized v: [0.6 0.8]
Length of normalized v: 1.0
```

Now `v_normalized` points in the same direction as `v` but has unit length.

3. Building an orthonormal set in 2D

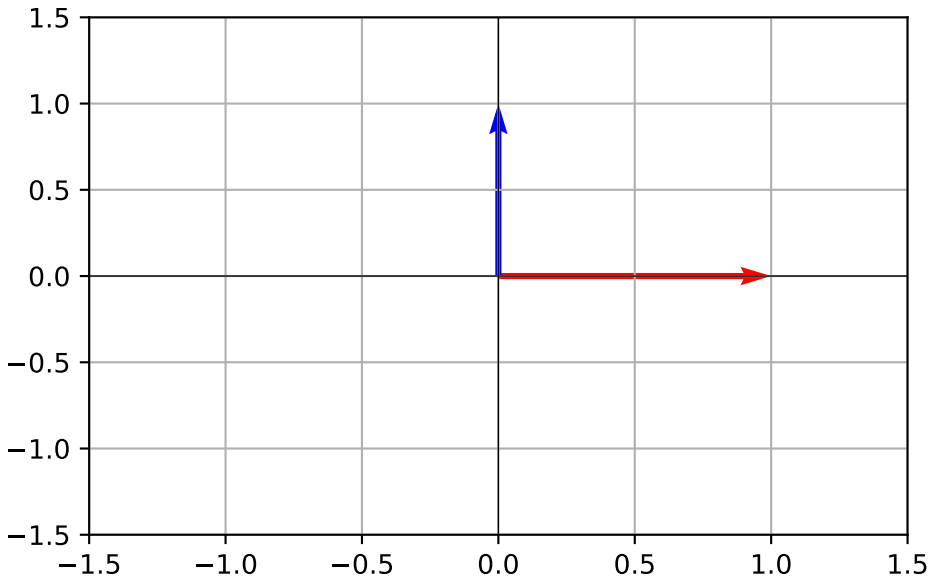
```
u1 = np.array([1, 0])
u2 = np.array([0, 1])

print("u1 length:", np.linalg.norm(u1))
print("u2 length:", np.linalg.norm(u2))
print("u1 · u2 =", np.dot(u1,u2))
```

```
u1 length: 1.0
u2 length: 1.0
u1 · u2 = 0
```

Both have length 1, and their dot product is 0. That makes `{u1, u2}` an orthonormal set in 2D.

4. Visualizing 2D orthonormal vectors

```
plt.quiver(0,0,u1[0],u1[1],angles='xy',scale_units='xy',scale=1,color='r')
plt.quiver(0,0,u2[0],u2[1],angles='xy',scale_units='xy',scale=1,color='b')

plt.xlim(-1.5,1.5)
plt.ylim(-1.5,1.5)
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.show()
```



You'll see the red and blue arrows at right angles, each of length 1.

5. Orthonormal set in 3D In 3D, the standard basis vectors are:

```
i = np.array([1,0,0])
j = np.array([0,1,0])
k = np.array([0,0,1])

print("‖i‖ =", np.linalg.norm(i))
print("‖j‖ =", np.linalg.norm(j))
print("‖k‖ =", np.linalg.norm(k))
print("i·j =", np.dot(i,j))
print("j·k =", np.dot(j,k))
print("i·k =", np.dot(i,k))
```

```
‖i‖ = 1.0
‖j‖ = 1.0
‖k‖ = 1.0
i·j = 0
j·k = 0
i·k = 0
```

Lengths are all 1, and dot products are 0. So `{i, j, k}` is an orthonormal set in ^{3}.

6. Testing if a set is orthonormal We can write a helper function:

```
def is_orthonormal(vectors):
    for i in range(len(vectors)):
        for j in range(len(vectors)):
            dot = np.dot(vectors[i], vectors[j])
            if i == j:
                if not np.isclose(dot, 1): return False
            else:
                if not np.isclose(dot, 0): return False
    return True

print(is_orthonormal([i, j, k]))  # True
```

```
True
```

7. Constructing a new orthonormal pair Not all orthonormal sets look like the axes.

```
u1 = np.array([1,1]) / np.sqrt(2)
u2 = np.array([-1,1]) / np.sqrt(2)

print("u1·u2 =", np.dot(u1,u2))
print("‖u1‖ =", np.linalg.norm(u1))
print("‖u2‖ =", np.linalg.norm(u2))
```

```
u1·u2 = 0.0
‖u1‖ = 0.9999999999999999
‖u2‖ = 0.9999999999999999
```

This gives a rotated orthonormal basis in 2D.

Try It Yourself

1. Normalize `(2,2,1)` to make it a unit vector.
2. Test whether the set `{[1,0,0], [0,2,0], [0,0,3]}` is orthonormal.
3. Construct two vectors in 2D that are not perpendicular. Normalize them and check if the dot product is still zero.

Chapter 2. Matrices and basic operations

11. Matrices as Tables and as Machines

Matrices can feel mysterious at first, but there are two simple ways to think about them:

1. As tables of numbers - just a grid you can store and manipulate.
2. As machines - something that takes a vector in and spits a new vector out.

In this lab, we'll explore both views and see how they connect.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. A matrix as a table of numbers

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

print("Matrix A:\n", A)
print("Shape of A:", A.shape)
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]]
Shape of A: (2, 3)
```

Here, `A` is a 2×3 matrix (2 rows, 3 columns).

- Rows = horizontal slices → `[1,2,3]` and `[4,5,6]`
- Columns = vertical slices → `[1,4]`, `[2,5]`, `[3,6]`

2. Accessing rows and columns

```
first_row = A[0]         # row 0
second_column = A[:,1]   # column 1

print("First row:", first_row)
print("Second column:", second_column)
```

```
First row: [1 2 3]
Second column: [2 5]
```

Rows are whole vectors too, and so are columns.

3. A matrix as a machine

A matrix can "act" on a vector. If `x = [x1, x2, x3]`, then `A·x` is computed by taking linear combinations of the columns of `A`.

```
x = np.array([1, 0, -1])  # a 3D vector
result = A.dot(x)

print("A·x =", result)
```

```
A·x = [-2 -2]
```

Interpretation: multiply `A` by `x` = combine columns of `A` with weights from `x`.

$$A \cdot x = 1 \cdot (\text{col } 1) + 0 \cdot (\text{col } 2) + (-1) \cdot (\text{col } 3)$$

4. Verifying column combination view

```
col1 = A[:,0]
col2 = A[:,1]
col3 = A[:,2]

manual = 1*col1 + 0*col2 + (-1)*col3
print("Manual combination:", manual)
print("A·x result:", result)
```

```
Manual combination: [-2 -2]
A·x result: [-2 -2]
```

They match exactly. This shows the "machine" interpretation is just a shortcut for column combinations.

5. Geometric intuition (2D example)

```
B = np.array([
    [2, 0],
    [0, 1]
])

v = np.array([1,2])
print("B·v =", B.dot(v))
```

```
B·v = [2 2]
```

Here, `B` scales the x-direction by 2 while leaving the y-direction alone. So `(1,2)` becomes `(2,2)`.

Try It Yourself

1. Create a 3×3 identity matrix with `np.eye(3)` and multiply it by different vectors. What happens?
2. Build a matrix `[[0,-1],[1,0]]`. Try multiplying it by `(1,0)` and `(0,1)`. What transformation is this?
3. Create your own 2×2 matrix that flips vectors across the x-axis. Test it on `(1,2)` and `(-3,4)`.

The Takeaway

- A matrix is both a grid of numbers and a machine that transforms vectors.
- Matrix–vector multiplication is the same as combining columns with given weights.
- Thinking of matrices as machines helps build intuition for rotations, scalings, and other transformations later.

12. Matrix Shapes, Indexing, and Block Views

Matrices come in many shapes, and learning to read their structure is essential. Shape tells us how many rows and columns a matrix has. Indexing lets us grab specific entries, rows, or columns. Block views let us zoom in on submatrices, which is extremely useful for both theory and computation.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Matrix shapes

The shape of a matrix is `(rows, columns)`.

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

print("Matrix A:\n", A)
print("Shape of A:", A.shape)
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Shape of A: (3, 3)
```

Here, `A` is a 3×3 matrix.

2. Indexing elements

In NumPy, rows and columns are 0-based. The first entry is `A[0,0]`.

```
print("A[0,0] =", A[0,0])  # top-left element
print("A[1,2] =", A[1,2])  # second row, third column
```

```
A[0,0] = 1
A[1,2] = 6
```

3. Extracting rows and columns

```
row1 = A[0]       # first row
col2 = A[:,1]     # second column

print("First row:", row1)
print("Second column:", col2)
```

```
First row: [1 2 3]
Second column: [2 5 8]
```

Notice: `A[i]` gives a row, `A[:,j]` gives a column.

4. Slicing submatrices (block view)

You can slice multiple rows and columns to form a smaller matrix.

```
block = A[0:2, 1:3]  # rows 0-1, columns 1-2
print("Block submatrix:\n", block)
```

```
Block submatrix:
 [[2 3]
 [5 6]]
```

This block is:

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

5. Modifying parts of a matrix

```
A[0,0] = 99
print("Modified A:\n", A)

A[1,:] = [10, 11, 12]   # replace row 1
print("After replacing row 1:\n", A)
```

```
Modified A:
 [[99  2  3]
 [ 4  5  6]
 [ 7  8  9]]
After replacing row 1:
```

```
 [[99  2  3]
 [10 11 12]
 [ 7  8  9]]
```

6. Non-square matrices

Not all matrices are square. Shapes can be rectangular, too.

```
B = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])

print("Matrix B:\n", B)
print("Shape of B:", B.shape)
```

```
Matrix B:
 [[1 2]
 [3 4]
 [5 6]]
Shape of B: (3, 2)
```

Here, `B` is 3×2 (3 rows, 2 columns).

7. Block decomposition idea

We can think of large matrices as made of smaller blocks. This is common in linear algebra proofs and algorithms.

```
C = np.array([
    [1,2,3,4],
    [5,6,7,8],
    [9,10,11,12],
    [13,14,15,16]
])

top_left = C[0:2, 0:2]
bottom_right = C[2:4, 2:4]

print("Top-left block:\n", top_left)
print("Bottom-right block:\n", bottom_right)
```

```
Top-left block:
 [[1 2]
 [5 6]]
Bottom-right block:
 [[11 12]
 [15 16]]
```

This is the start of block matrix notation.

Try It Yourself

1. Create a 4×5 matrix with values 1–20 using `np.arange(1,21).reshape(4,5)`. Find its shape.
2. Extract the middle row and last column.
3. Cut it into four 2×2 blocks. Can you reassemble them in a different order?

13. Matrix Addition and Scalar Multiplication

Now that we understand matrix shapes and indexing, let's practice two of the simplest but most important operations: adding matrices and scaling them with numbers (scalars). These operations extend the rules we already know for vectors.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Adding two matrices You can add two matrices if (and only if) they have the same shape. Addition happens entry by entry.

```
A = np.array([
    [1, 2],
    [3, 4]
])

B = np.array([
    [5, 6],
```

```
    [7, 8]
])

C = A + B
print("A + B =\n", C)
```

```
A + B =
 [[ 6  8]
 [10 12]]
```

Each element in `C` is the sum of corresponding elements in `A` and `B`.

2. Scalar multiplication Multiplying a matrix by a scalar multiplies every entry by that number.

```
k = 3
D = k * A
print("3 * A =\n", D)
```

```
3 * A =
 [[ 3  6]
 [ 9 12]]
```

Here, each element of `A` is tripled.

3. Combining both operations We can mix addition and scaling, just like with vectors.

```
combo = 2*A - B
print("2A - B =\n", combo)
```

```
2A - B =
 [[-3 -2]
 [-1  0]]
```

This creates new matrices as linear combinations of others.

4. Zero matrix A matrix of all zeros acts like “nothing happens” for addition.

```
zero = np.zeros((2,2))
print("Zero matrix:\n", zero)
print("A + Zero =\n", A + zero)
```

```
Zero matrix:
 [[0. 0.]
 [0. 0.]]
A + Zero =
 [[1. 2.]
 [3. 4.]]
```

5. Shape mismatch (what fails) If shapes don't match, NumPy throws an error.

```
X = np.array([
    [1,2,3],
    [4,5,6]
])

try:
    print(A + X)
except ValueError as e:
    print("Error:", e)
```

```
Error: operands could not be broadcast together with shapes (2,2) (2,3)
```

This shows why shape consistency matters.

Try It Yourself

1. Create two random 3×3 matrices with `np.random.randint(0,10,(3,3))` and add them.
2. Multiply a 4×4 matrix by `-1`. What happens to its entries?
3. Compute `3A + 2B` with the matrices from above. Compare with doing each step manually.

14. Matrix–Vector Product (Linear Combinations of Columns)

This lab introduces the matrix–vector product, one of the most important operations in linear algebra. Multiplying a matrix by a vector doesn't just crunch numbers - it produces a new vector by combining the matrix's columns in a weighted way.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. A simple matrix and vector

```
A = np.array([
    [1, 2],
    [3, 4],
    [5, 6]
])  # 3×2 matrix

x = np.array([2, -1])  # 2D vector
```

Here, `A` has 2 columns, so we can multiply it by a 2D vector `x`.

2. Matrix–vector multiplication in NumPy

```
y = A.dot(x)
print("A·x =", y)
```

```
A·x = [0 2 4]
```

Result: a 3D vector.

3. Interpreting the result as linear combinations

Matrix `A` has two columns:

```
col1 = A[:,0]   # first column
col2 = A[:,1]   # second column

manual = 2*col1 + (-1)*col2
print("Manual linear combination:", manual)
```

```
Manual linear combination: [0 2 4]
```

This matches `A·x`. In words: *multiply each column by the corresponding entry of* $\boldsymbol{x}$ *and then add them up.*

4. Another example (geometry)

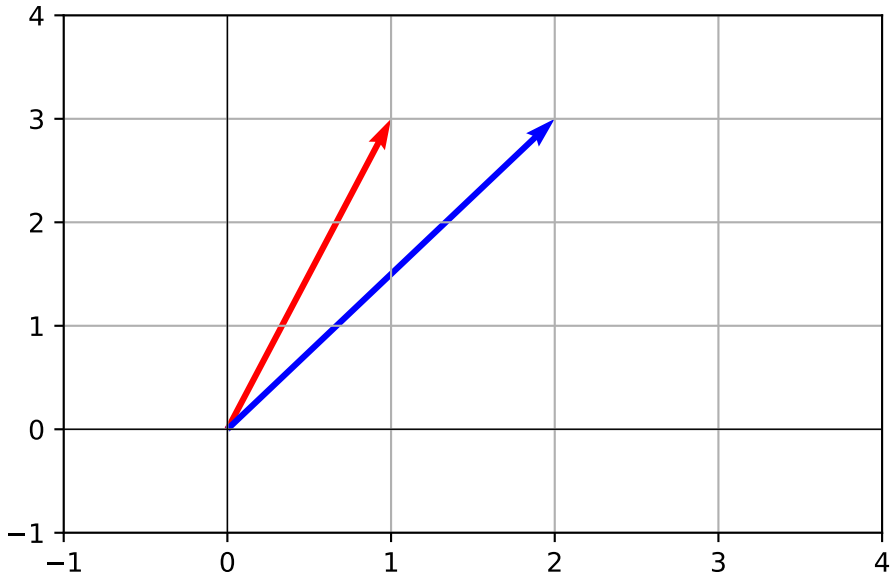
```
B = np.array([
    [2, 0],
    [0, 1]
])  # stretches x-axis by 2

v = np.array([1, 3])
print("B·v =", B.dot(v))
```

```
B·v = [2 3]
```

Here, `(1,3)` becomes `(2,3)`. The x-component was doubled, while y stayed the same.

5. Visualization of matrix action

```
import matplotlib.pyplot as plt

# draw original vector
plt.quiver(0,0,v[0],v[1],angles='xy',scale_units='xy',scale=1,color='r',label='v')

# draw transformed vector
v_transformed = B.dot(v)
plt.quiver(0,0,v_transformed[0],v_transformed[1],angles='xy',scale_units='xy',scale=1,color=

plt.xlim(-1,4)
plt.ylim(-1,4)
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.show()
```



Red arrow = original vector, blue arrow = transformed vector.

Try It Yourself

1. Multiply

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 2 \end{bmatrix}, \; x = [3, 1]$$

   What's the result?

2. Replace `B` with `[[0,-1],[1,0]]`. Multiply it by `(1,0)` and `(0,1)`. What geometric transformation does this represent?

3. For a 4×4 identity matrix (`np.eye(4)`), try multiplying by any 4D vector. What do you observe?

15. Matrix–Matrix Product (Composition of Linear Steps)

Matrix–matrix multiplication is how we combine two linear transformations into one. Instead of applying one transformation, then another, we can multiply their matrices and get a single matrix that does both at once.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Matrix–matrix multiplication in NumPy

```
A = np.array([
    [1, 2],
    [3, 4]
])  # 2×2

B = np.array([
    [2, 0],
    [1, 2]
])  # 2×2

C = A.dot(B)   # or A @ B
print("A·B =\n", C)
```

```
A·B =
 [[ 4  4]
 [10  8]]
```

The result `C` is another 2×2 matrix.

2. Manual computation

Each entry of `C` is computed as a row of A dotted with a column of B:

```
c11 = A[0,:].dot(B[:,0])
c12 = A[0,:].dot(B[:,1])
c21 = A[1,:].dot(B[:,0])
c22 = A[1,:].dot(B[:,1])

print("Manual C =\n", np.array([[c11,c12],[c21,c22]]))
```

```
Manual C =
 [[ 4  4]
 [10  8]]
```

This should match `A·B`.

3. Geometric interpretation

Let's see how two transformations combine.

- Matrix `B` scales x by 2 and stretches y by 2.
- Matrix `A` applies another linear transformation.

Together, `C = A·B` does both in one step.

```
v = np.array([1,1])

print("First apply B:", B.dot(v))
print("Then apply A:", A.dot(B.dot(v)))
print("Directly with C:", C.dot(v))
```

```
First apply B: [2 3]
Then apply A: [ 8 18]
Directly with C: [ 8 18]
```

The result is the same: applying `B` then `A` is equivalent to applying `C`.

4. Non-square matrices

Matrix multiplication also works for rectangular matrices, as long as the inner dimensions match.

```
M = np.array([
    [1, 0, 2],
    [0, 1, 3]
])  # 2×3

N = np.array([
    [1, 2],
    [0, 1],
    [4, 0]
])  # 3×2

P = M.dot(N)  # result is 2×2
print("M·N =\n", P)
```

```
M·N =
 [[ 9  2]
 [12  1]]
```

Shape rule: `(2×3)·(3×2) = (2×2)`.

5. Associativity (but not commutativity)

Matrix multiplication is associative: `(A·B)·C = A·(B·C)`. But it's not commutative: in general, `A·B  B·A`.

```
A = np.array([[1,2],[3,4]])
B = np.array([[0,1],[1,0]])

print("A·B =\n", A.dot(B))
print("B·A =\n", B.dot(A))
```

```
A·B =
 [[2 1]
 [4 3]]
B·A =
 [[3 4]
 [1 2]]
```

The two results are different.

Try It Yourself

1. Multiply

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \; B = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

   What transformation does `A·B` represent?

2. Create a random 3×2 matrix and a 2×4 matrix. Multiply them. What shape is the result?

3. Verify with Python that `(A·B)·C = A·(B·C)` for some 3×3 random matrices.

16. Identity, Inverse, and Transpose

In this lab, we'll meet three special matrix operations and objects: the identity matrix, the inverse, and the transpose. These are the building blocks of matrix algebra, each with a simple meaning but deep importance.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Identity matrix The identity matrix is like the number 1 for matrices: multiplying by it changes nothing.

```
I = np.eye(3)  # 3×3 identity matrix
print("Identity matrix:\n", I)

A = np.array([
    [2, 1, 0],
    [0, 1, 3],
    [4, 0, 1]
])

print("A·I =\n", A.dot(I))
print("I·A =\n", I.dot(A))
```

```
Identity matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A·I =
 [[2. 1. 0.]
 [0. 1. 3.]
 [4. 0. 1.]]
I·A =
 [[2. 1. 0.]
 [0. 1. 3.]
 [4. 0. 1.]]
```

Both equal `A`.

2. Transpose The transpose flips rows and columns.

```
B = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

print("B:\n", B)
print("B.T:\n", B.T)
```

```
B:
 [[1 2 3]
 [4 5 6]]
B.T:
 [[1 4]
 [2 5]
 [3 6]]
```

- Original: 2×3
- Transpose: 3×2

Geometrically, transpose swaps the axes when vectors are viewed in row/column form.

3. Inverse The inverse matrix is like dividing by a number: multiplying a matrix by its inverse gives the identity.

```
C = np.array([
    [2, 1],
    [5, 3]
])

C_inv = np.linalg.inv(C)
print("Inverse of C:\n", C_inv)

print("C·C_inv =\n", C.dot(C_inv))
print("C_inv·C =\n", C_inv.dot(C))
```

```
Inverse of C:
 [[ 3. -1.]
 [-5.  2.]]
```

```
C·C_inv =
 [[ 1.00000000e+00  2.22044605e-16]
 [-8.88178420e-16  1.00000000e+00]]
C_inv·C =
 [[1.00000000e+00 3.33066907e-16]
 [0.00000000e+00 1.00000000e+00]]
```

Both products are (approximately) the identity.

4. Matrices that don't have inverses Not every matrix is invertible. If a matrix is singular (determinant = 0), it has no inverse.

```
D = np.array([
    [1, 2],
    [2, 4]
])

try:
    np.linalg.inv(D)
except np.linalg.LinAlgError as e:
    print("Error:", e)
```

```
Error: Singular matrix
```

Here, the second row is a multiple of the first, so `D` can't be inverted.

5. Transpose and inverse together For invertible matrices,

$$(A^T)^{-1} = (A^{-1})^T$$

We can check this numerically:

```
A = np.array([
    [1, 2],
    [3, 5]
])

lhs = np.linalg.inv(A.T)
rhs = np.linalg.inv(A).T

print("Do they match?", np.allclose(lhs, rhs))
```

```
Do they match? True
```

Try It Yourself

1. Create a 4×4 identity matrix. Multiply it by any 4×1 vector. Does it change?
2. Take a random 2×2 matrix with `np.random.randint`. Compute its inverse and check if multiplying gives identity.
3. Pick a rectangular 3×2 matrix. What happens when you try `np.linalg.inv`? Why?
4. Compute `(A.T).T` for some matrix `A`. What do you notice?

17. Symmetric, Diagonal, Triangular, and Permutation Matrices

In this lab, we'll meet four important families of special matrices. They have patterns that make them easier to understand, compute with, and use in algorithms.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Symmetric matrices A matrix is symmetric if it equals its transpose: $A = A^T$.

```
A = np.array([
    [2, 3, 4],
    [3, 5, 6],
    [4, 6, 8]
])

print("A:\n", A)
print("A.T:\n", A.T)
print("Is symmetric?", np.allclose(A, A.T))
```

```
A:
 [[2 3 4]
 [3 5 6]
 [4 6 8]]
A.T:
 [[2 3 4]
 [3 5 6]
 [4 6 8]]
Is symmetric? True
```

Symmetric matrices appear in physics, optimization, and statistics (e.g., covariance matrices).

2. Diagonal matrices A diagonal matrix has nonzero entries only on the main diagonal.

```
D = np.diag([1, 5, 9])
print("Diagonal matrix:\n", D)

x = np.array([2, 3, 4])
print("D·x =", D.dot(x))  # scales each component
```

```
Diagonal matrix:
 [[1 0 0]
 [0 5 0]
 [0 0 9]]
D·x = [ 2 15 36]
```

Diagonal multiplication simply scales each coordinate separately.

3. Triangular matrices Upper triangular: all entries below the diagonal are zero. Lower triangular: all entries above the diagonal are zero.

```
U = np.array([
    [1, 2, 3],
    [0, 4, 5],
    [0, 0, 6]
])

L = np.array([
    [7, 0, 0],
    [8, 9, 0],
    [1, 2, 3]
])

print("Upper triangular U:\n", U)
print("Lower triangular L:\n", L)
```

```
Upper triangular U:
 [[1 2 3]
 [0 4 5]
 [0 0 6]]
Lower triangular L:
```

```
 [[7 0 0]
 [8 9 0]
 [1 2 3]]
```

These are important in solving linear systems (e.g., Gaussian elimination).

4. Permutation matrices A permutation matrix rearranges the order of coordinates. Each row and each column has exactly one `1`, everything else is `0`.

```
P = np.array([
    [0, 1, 0],
    [0, 0, 1],
    [1, 0, 0]
])

print("Permutation matrix P:\n", P)

v = np.array([10, 20, 30])
print("P·v =", P.dot(v))
```

```
Permutation matrix P:
 [[0 1 0]
 [0 0 1]
 [1 0 0]]
P·v = [20 30 10]
```

Here, `P` cycles `(10,20,30)` into `(20,30,10)`.

5. Checking properties

```
def is_symmetric(M): return np.allclose(M, M.T)
def is_diagonal(M): return np.count_nonzero(M - np.diag(np.diag(M))) == 0
def is_upper_triangular(M): return np.allclose(M, np.triu(M))
def is_lower_triangular(M): return np.allclose(M, np.tril(M))

print("A symmetric?", is_symmetric(A))
print("D diagonal?", is_diagonal(D))
print("U upper triangular?", is_upper_triangular(U))
print("L lower triangular?", is_lower_triangular(L))
```

```
A symmetric? True
D diagonal? True
U upper triangular? True
L lower triangular? True
```

Try It Yourself

1. Create a random symmetric matrix by generating any matrix `M` and computing `(M + M.T)/2`.
2. Build a 4×4 diagonal matrix with diagonal entries `[2,4,6,8]` and multiply it by `[1,1,1,1]`.
3. Make a permutation matrix that swaps the first and last components of a 3D vector.
4. Check whether the identity matrix is diagonal, symmetric, upper triangular, and lower triangular all at once.

18. Trace and Basic Matrix Properties

In this lab, we'll introduce the trace of a matrix and a few quick properties that often appear in proofs, algorithms, and applications. The trace is simple to compute but surprisingly powerful.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. What is the trace? The trace of a square matrix is the sum of its diagonal entries:

$$\operatorname{tr}(A) = \sum_i A_{ii}$$

```
A = np.array([
    [2, 1, 3],
    [0, 4, 5],
    [7, 8, 6]
])
```

```
trace_A = np.trace(A)
print("Matrix A:\n", A)
print("Trace of A =", trace_A)
```

```
Matrix A:
 [[2 1 3]
 [0 4 5]
 [7 8 6]]
Trace of A = 12
```

Here, trace $= 2 + 4 + 6 = 12$.

2. Trace is linear For matrices `A` and `B`:

$$\text{tr}(A + B) = \text{tr}(A) + \text{tr}(B)$$

$$\text{tr}(cA) = c \cdot \text{tr}(A)$$

```
B = np.array([
    [1, 0, 0],
    [0, 2, 0],
    [0, 0, 3]
])

print("tr(A+B) =", np.trace(A+B))
print("tr(A) + tr(B) =", np.trace(A) + np.trace(B))

print("tr(3A) =", np.trace(3*A))
print("3 * tr(A) =", 3*np.trace(A))
```

```
tr(A+B) = 18
tr(A) + tr(B) = 18
tr(3A) = 36
3 * tr(A) = 36
```

3. Trace of a product One important property is:

$$\text{tr}(AB) = \text{tr}(BA)$$

```
C = np.array([
    [0,1],
    [2,3]
])

D = np.array([
    [4,5],
    [6,7]
])

print("tr(CD) =", np.trace(C.dot(D)))
print("tr(DC) =", np.trace(D.dot(C)))
```

```
tr(CD) = 37
tr(DC) = 37
```

Both are equal, even though `CD` and `DC` are different matrices.

4. Trace and eigenvalues The trace equals the sum of eigenvalues of a matrix (counting multiplicities).

```
vals, vecs = np.linalg.eig(A)
print("Eigenvalues:", vals)
print("Sum of eigenvalues =", np.sum(vals))
print("Trace =", np.trace(A))
```

```
Eigenvalues: [12.83286783  2.13019807 -2.9630659 ]
Sum of eigenvalues = 12.000000000000004
Trace = 12
```

The results should match (within rounding error).

5. Quick invariants

- Trace doesn't change under transpose: `tr(A) = tr(A.T)`
- Trace doesn't change under similarity transforms: `tr(P^-1 A P) = tr(A)`

```
print("tr(A) =", np.trace(A))
print("tr(A.T) =", np.trace(A.T))
```

```
tr(A) = 12
tr(A.T) = 12
```

Try It Yourself

1. Create a 2×2 rotation matrix for 90°:

$$R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

   What is its trace? What does that tell you about its eigenvalues?

2. Make a random 3×3 matrix and compare `tr(A)` with the sum of eigenvalues.

3. Test `tr(AB)` and `tr(BA)` with a rectangular matrix `A` (e.g. 2×3) and `B` (3×2). Do they still match?

19. Affine Transforms and Homogeneous Coordinates

Affine transformations let us do more than just linear operations - they include translations (shifting points), which ordinary matrices can't handle alone. To unify rotations, scalings, reflections, and translations, we use homogeneous coordinates.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Linear transformations vs affine transformations

- A linear transformation can rotate, scale, or shear, but always keeps the origin fixed.
- An affine transformation allows translation as well.

For example, shifting every point by `(2,3)` is affine but not linear.

2. Homogeneous coordinates idea We add an extra coordinate (usually `1`) to vectors.

- A 2D point `(x,y)` becomes `(x,y,1)`.
- A 3D point `(x,y,z)` becomes `(x,y,z,1)`.

This trick lets us represent translations using matrix multiplication.

3. 2D translation matrix

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

```
T = np.array([
    [1, 0, 2],
    [0, 1, 3],
    [0, 0, 1]
])

p = np.array([1, 1, 1])  # point at (1,1)
p_translated = T.dot(p)

print("Original point:", p)
print("Translated point:", p_translated)
```

```
Original point: [1 1 1]
Translated point: [3 4 1]
```

This shifts `(1,1)` to `(3,4)`.

4. Combining rotation and translation

A 90° rotation around the origin in 2D:

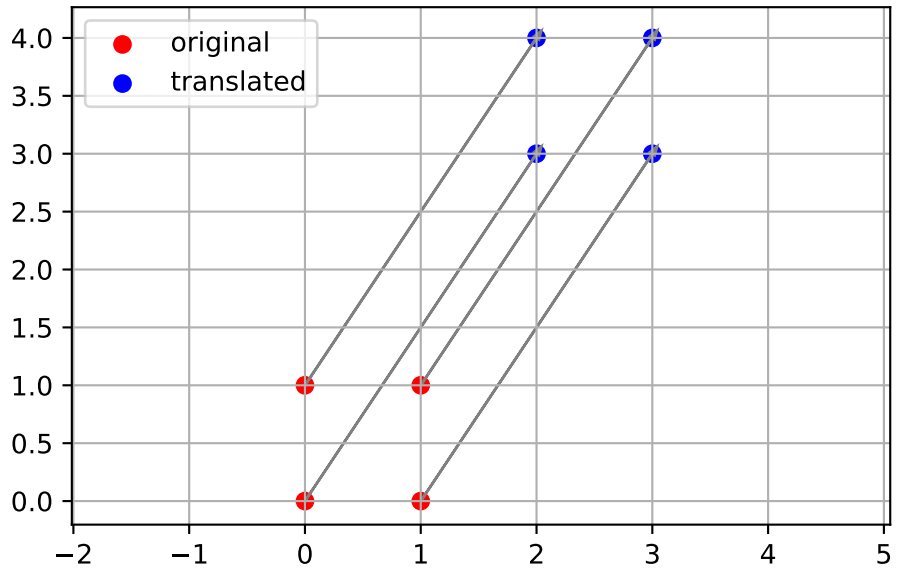
```
R = np.array([
    [0, -1, 0],
    [1,  0, 0],
    [0,  0, 1]
])

M = T.dot(R)  # rotate then translate
print("Combined transform:\n", M)

p = np.array([1, 0, 1])
print("Rotated + translated point:", M.dot(p))
```

```
Combined transform:
 [[ 0 -1  2]
 [ 1  0  3]
 [ 0  0  1]]
Rotated + translated point: [2 4 1]
```

Now we can apply rotation and translation in one step.

5. Visualization of translation

```
points = np.array([
    [0,0,1],
    [1,0,1],
    [1,1,1],
    [0,1,1]
])  # a unit square

transformed = points.dot(T.T)

plt.scatter(points[:,0], points[:,1], color='r', label='original')
plt.scatter(transformed[:,0], transformed[:,1], color='b', label='translated')

for i in range(len(points)):
    plt.arrow(points[i,0], points[i,1],
              transformed[i,0]-points[i,0],
              transformed[i,1]-points[i,1],
              head_width=0.05, color='gray')

plt.legend()
plt.axis('equal')
plt.grid()
plt.show()
```



You'll see the red unit square moved to a blue unit square shifted by `(2,3)`.

6. Extending to 3D In 3D, homogeneous coordinates use 4×4 matrices. Translations, rotations, and scalings all fit the same framework.

```
T3 = np.array([
    [1,0,0,5],
    [0,1,0,-2],
    [0,0,1,3],
    [0,0,0,1]
])

p3 = np.array([1,2,3,1])
print("Translated 3D point:", T3.dot(p3))
```

```
Translated 3D point: [6 0 6 1]
```

This shifts `(1,2,3)` to `(6,0,6)`.

Try It Yourself

1. Build a scaling matrix in homogeneous coordinates that doubles both x and y, and apply it to `(1,1)`.
2. Create a 2D transform that rotates by 90° and then shifts by `(-2,1)`. Apply it to `(0,2)`.
3. In 3D, translate `(0,0,0)` by `(10,10,10)`. What homogeneous matrix did you use?

20. Computing with Matrices (Cost Counts and Simple Speedups)

Working with matrices is not just about theory - in practice, we care about how much computation it takes to perform operations, and how we can make them faster. This lab introduces basic cost analysis (counting operations) and demonstrates simple NumPy optimizations.

Set Up Your Lab

```
import numpy as np
import time
```

Step-by-Step Code Walkthrough

1. Counting operations (matrix–vector multiply)

If `A` is an $m \times n$ matrix and `x` is an n-dimensional vector, computing `A·x` takes about $m \times n$ multiplications and the same number of additions.

```
m, n = 3, 4
A = np.random.randint(1,10,(m,n))
x = np.random.randint(1,10,n)

print("Matrix A:\n", A)
print("Vector x:", x)
print("A·x =", A.dot(x))
```

```
Matrix A:
 [[1 8 3 5]
 [6 8 4 2]
 [3 3 1 5]]
Vector x: [6 3 1 6]
A·x = [63 76 58]
```

Here the cost is $3 \times 4 = 12$ multiplications + 12 additions.

2. Counting operations (matrix–matrix multiply)

For an $m \times n$ times $n \times p$ multiplication, the cost is about $m \times n \times p$.

```
m, n, p = 3, 4, 2
A = np.random.randint(1,10,(m,n))
B = np.random.randint(1,10,(n,p))

C = A.dot(B)
print("A·B =\n", C)
```

```
A·B =
 [[ 60 130]
 [ 62 153]
 [ 70 164]]
```

Here the cost is $3 \times 4 \times 2 = 24$ multiplications + 24 additions.

3. Timing with NumPy (vectorized vs loop)

NumPy is optimized in C and Fortran under the hood. Let's compare matrix multiplication with and without vectorization.

```
n = 50
A = np.random.randn(n,n)
B = np.random.randn(n,n)

# Vectorized
start = time.time()
C1 = A.dot(B)
end = time.time()
print("Vectorized dot:", round(end-start,3), "seconds")

# Manual loops
C2 = np.zeros((n,n))
start = time.time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C2[i,j] += A[i,k]*B[k,j]
end = time.time()
print("Triple loop:", round(end-start,3), "seconds")
```

```
Vectorized dot: 0.0 seconds
Triple loop: 0.063 seconds
```

The vectorized version should be thousands of times faster.

4. Broadcasting tricks

NumPy lets us avoid loops by broadcasting operations across entire rows or columns.

```
A = np.array([
    [1,2,3],
    [4,5,6]
])

# Add 10 to every entry
print("A+10 =\n", A+10)

# Multiply each row by a different scalar
scales = np.array([1,10])[:,None]
print("Row-scaled A =\n", A*scales)
```

```
A+10 =
 [[11 12 13]
 [14 15 16]]
Row-scaled A =
 [[ 1  2  3]
 [40 50 60]]
```

5. Memory and data types

For large computations, data type matters.

```
A = np.random.randn(1000,1000).astype(np.float32)  # 32-bit floats
B = np.random.randn(1000,1000).astype(np.float32)

start = time.time()
C = A.dot(B)
print("Result shape:", C.shape, "dtype:", C.dtype)
print("Time:", round(time.time()-start,3), "seconds")
```

```
Result shape: (1000, 1000) dtype: float32
Time: 0.014 seconds
```

Using `float32` instead of `float64` halves memory use and can speed up computation (at the cost of some precision).

Try It Yourself

1. Compute the cost of multiplying a 200×500 matrix with a 500×1000 matrix. How many multiplications are needed?
2. Time matrix multiplication for sizes 100, 500, 1000 in NumPy. How does the time scale?
3. Experiment with `float32` vs `float64` in NumPy. How do speed and memory change?
4. Try broadcasting: multiply each column of a matrix by `[1,2,3,...]`.

The Takeaway

- Matrix operations have predictable computational costs: `A·x` $\sim m \times n$, `A·B` $\sim m \times n \times p$.
- Vectorized NumPy operations are vastly faster than Python loops.
- Broadcasting and choosing the right data type are simple speedups every beginner should learn.

Chapter 3. Linear Systems and Elimination

21. From Equations to Matrices (Augmenting and Encoding)

Linear algebra often begins with solving systems of linear equations. For example:

$$\begin{cases} x + 2y = 5 \\ 3x - y = 4 \end{cases}$$

Instead of juggling symbols, we can encode the entire system into a matrix. This is the key idea that lets computers handle thousands or millions of equations efficiently.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Write a system of equations

We'll use this small example:

$$\begin{cases} 2x + y = 8 \\ -3x + 4y = -11 \end{cases}$$

2. Encode coefficients and constants

- Coefficient matrix A: numbers multiplying variables.
- Variable vector x: unknowns `[x, y]`.
- Constant vector b: right-hand side.

```
A = np.array([
    [2, 1],
    [-3, 4]
])

b = np.array([8, -11])

print("Coefficient matrix A:\n", A)
print("Constants vector b:", b)
```

```
Coefficient matrix A:
 [[ 2  1]
 [-3  4]]
Constants vector b: [  8 -11]
```

So the system is $A \cdot x = b$.

3. Augmented matrix

We can bundle the system into one compact matrix:

$$[A|b] = \left[\begin{array}{cc|c} 2 & 1 & 8 \\ -3 & 4 & -11 \end{array}\right]$$

```
augmented = np.column_stack((A, b))
print("Augmented matrix:\n", augmented)
```

```
Augmented matrix:
 [[  2   1   8]
 [ -3   4 -11]]
```

This format is useful for elimination algorithms.

4. Solving directly with NumPy

```
solution = np.linalg.solve(A, b)
print("Solution (x,y):", solution)
```

```
Solution (x,y): [3.90909091 0.18181818]
```

Here NumPy solves the system using efficient algorithms.

5. Checking the solution

Always verify:

```
check = A.dot(solution)
print("A·x =", check, "should equal b =", b)
```

```
A·x = [  8. -11.] should equal b = [  8 -11]
```

6. Another example (3 variables)

$$\begin{cases} x + y + z = 6 \\ 2x - y + z = 3 \\ -x + 2y - z = 2 \end{cases}$$

```
A = np.array([
    [1, 1, 1],
    [2, -1, 1],
    [-1, 2, -1]
])

b = np.array([6, 3, 2])

print("Augmented matrix:\n", np.column_stack((A, b)))
print("Solution:", np.linalg.solve(A, b))
```

```
Augmented matrix:
 [[ 1  1  1  6]
 [ 2 -1  1  3]
 [-1  2 -1  2]]
Solution: [2.33333333 2.66666667 1.        ]
```

Try It Yourself

1. Encode the system:

$$\begin{cases} 2x - y = 1 \\ x + 3y = 7 \end{cases}$$

   Write `A` and `b`, then solve.

2. For a 3×3 system, try creating a random coefficient matrix with `np.random.randint(-5,5,(3,3))` and a random `b`. Use `np.linalg.solve`.

3. Modify the constants `b` slightly and see how the solution changes. This introduces the idea of sensitivity.

The Takeaway

- Systems of linear equations can be neatly written as $A \cdot x = b$.
- The augmented matrix $[A|b]$ is a compact way to set up elimination.
- This matrix encoding transforms algebra problems into matrix problems - the gateway to all of linear algebra.

22. Row Operations (Legal Moves That Keep Solutions)

When solving linear systems, we don't want to change the solutions - just simplify the system into an easier form. This is where row operations come in. They are the "legal moves" we can do on an augmented matrix $[A|b]$ without changing the solution set.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Three legal row operations
2. Swap two rows ($R_i \leftrightarrow R_j$)
3. Multiply a row by a nonzero scalar ($R_i \to c \cdot R_i$)
4. Replace a row with itself plus a multiple of another row ($R_i \to R_i + c \cdot R_j$)

These preserve the solution set.

2. Start with an augmented matrix

System:

$$\begin{cases} x + 2y = 5 \\ 3x + 4y = 6 \end{cases}$$

```
A = np.array([
    [1, 2, 5],
    [3, 4, 6]
], dtype=float)

print("Initial augmented matrix:\n", A)
```

```
Initial augmented matrix:
 [[1. 2. 5.]
 [3. 4. 6.]]
```

3. Row swap

Swap row 0 and row 1.

```
A[[0,1]] = A[[1,0]]
print("After swapping rows:\n", A)
```

```
After swapping rows:
 [[3. 4. 6.]
 [1. 2. 5.]]
```

4. Multiply a row by a scalar

Make the pivot in row 0 equal to 1.

```
A[0] = A[0] / A[0,0]
print("After scaling first row:\n", A)
```

```
After scaling first row:
 [[1.         1.33333333 2.        ]
 [1.         2.         5.        ]]
```

5. Add a multiple of another row

Eliminate the first column of row 1.

```
A[1] = A[1] - 3*A[0]
print("After eliminating x from second row:\n", A)
```

```
After eliminating x from second row:
 [[ 1.          1.33333333  2.        ]
 [-2.         -2.         -1.        ]]
```

Now the system is simpler: second row has only y.

6. Solving from the new system

```
y = A[1,2] / A[1,1]
x = (A[0,2] - A[0,1]*y) / A[0,0]
print("Solution: x =", x, ", y =", y)
```

```
Solution: x = 1.3333333333333335 , y = 0.5
```

7. Using NumPy step-by-step vs solver

```
coeff = np.array([[1,2],[3,4]])
const = np.array([5,6])
print("np.linalg.solve result:", np.linalg.solve(coeff,const))
```

```
np.linalg.solve result: [-4.   4.5]
```

Both methods give the same solution.

Try It Yourself

1. Take the system:

$$\begin{cases} 2x + y = 7 \\ x - y = 1 \end{cases}$$

   Write its augmented matrix, then:

   - Swap rows.
   - Scale the first row.
   - Eliminate one variable.

2. Create a random 3×3 system with integers between -5 and 5. Perform at least one of each row operation manually in code.

3. Experiment with multiplying a row by `0`. What happens, and why is this not allowed as a legal operation?

The Takeaway

- The three legal row operations are row swap, row scaling, and row replacement.
- These steps preserve the solution set while moving toward a simpler form.
- They are the foundation of Gaussian elimination, the standard algorithm for solving linear systems.

23. Row-Echelon and Reduced Row-Echelon Forms (Target Shapes)

When solving systems, our goal is to simplify the augmented matrix into a standard shape where the solutions are easy to read. These shapes are called row-echelon form (REF) and reduced row-echelon form (RREF).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

We'll use NumPy for basic work and SymPy for exact RREF (since NumPy doesn't have it built-in).

Step-by-Step Code Walkthrough

1. Row-Echelon Form (REF)

- All nonzero rows are above any zero rows.
- Each leading entry (pivot) is to the right of the pivot in the row above.
- Pivots are usually scaled to 1, but not strictly required.

Example system:

$$\begin{cases} x + 2y + z = 7 \\ 2x + 4y + z = 12 \\ 3x + 6y + 2z = 17 \end{cases}$$

```
A = np.array([
    [1, 2, 1, 7],
    [2, 4, 1, 12],
    [3, 6, 2, 17]
], dtype=float)

print("Augmented matrix:\n", A)
```

```
Augmented matrix:
 [[ 1.  2.  1.  7.]
 [ 2.  4.  1. 12.]
 [ 3.  6.  2. 17.]]
```

Perform elimination manually:

```
# eliminate first column entries below pivot
A[1] = A[1] - 2*A[0]
A[2] = A[2] - 3*A[0]
print("After eliminating first column:\n", A)
```

```
After eliminating first column:
 [[ 1.  2.  1.  7.]
 [ 0.  0. -1. -2.]
 [ 0.  0. -1. -4.]]
```

Now the pivots move diagonally across the matrix - this is row-echelon form.

2. Reduced Row-Echelon Form (RREF) In RREF, we go further:

- Every pivot = 1.
- Every pivot is the only nonzero in its column.

Instead of coding manually, we'll let SymPy handle it:

```
M = Matrix([
    [1, 2, 1, 7],
    [2, 4, 1, 12],
    [3, 6, 2, 17]
])

M_rref = M.rref()
print("RREF form:\n", M_rref[0])
```

```
RREF form:
 Matrix([[1, 2, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]])
```

SymPy shows the final canonical form.

3. Reading solutions from RREF

If the RREF looks like:

$$\begin{bmatrix} 1 & 0 & a & b \\ 0 & 1 & c & d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

It means:

- The first two variables are leading (pivots).
- The third variable is free.
- Solutions can be written in terms of the free variable.

4. A quick example with free variables

System:

$$x + y + z = 32x + y - z = 0$$

```
M2 = Matrix([
    [1,1,1,3],
    [2,1,-1,0]
])

M2_rref = M2.rref()
print("RREF form:\n", M2_rref[0])
```

```
RREF form:
 Matrix([[1, 0, -2, -3], [0, 1, 3, 6]])
```

Here, one column will not have a pivot → that variable is free.

Try It Yourself

1. Take the system:

$$2x + 3y = 6, \quad 4x + 6y = 12$$

   Write the augmented matrix and compute its RREF. What does it tell you about solutions?

2. Create a random 3×4 matrix in NumPy. Use SymPy's `Matrix.rref()` to compute its reduced form. Identify the pivot columns.

3. For the system:

$$x + 2y + 3z = 4, \quad 2x + 4y + 6z = 8$$

   Check if the equations are independent or multiples of each other by looking at the RREF.

The Takeaway

- REF organizes equations into a staircase shape.
- RREF goes further, making each pivot the only nonzero in its column.
- These canonical forms make it easy to identify pivot variables, free variables, and the solution set structure.

24. Pivots, Free Variables, and Leading Ones (Reading Solutions)

Once a matrix is in row-echelon or reduced row-echelon form, the solutions to the system become visible. The key is identifying pivots, leading ones, and free variables.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. What are pivots?

- A pivot is the first nonzero entry in a row (after elimination).
- In RREF, pivots are scaled to `1` and are called leading ones.
- Pivot columns correspond to basic variables.

2. Example system

$$\begin{cases} x + y + z = 6 \\ 2x + 3y + z = 10 \end{cases}$$

```
M = Matrix([
    [1,1,1,6],
    [2,3,1,10]
])

M_rref = M.rref()
print("RREF form:\n", M_rref[0])
```

```
RREF form:
 Matrix([[1, 0, 2, 8], [0, 1, -1, -2]])
```

3. Interpreting the RREF

Suppose the RREF comes out as:

$$\begin{bmatrix} 1 & 0 & -2 & 4 \\ 0 & 1 & 1 & 2 \end{bmatrix}$$

This means:

- Pivot columns: 1 and 2 → variables x and y are basic.
- Free variable: z.
- Equations:

$$x - 2z = 4, \quad y + z = 2$$

- Solution in terms of z:

$$x = 4 + 2z, \quad y = 2 - z, \quad z = z$$

4. Coding the solution extraction

```
rref_matrix, pivots = M_rref
print("Pivot columns:", pivots)

# free variables are the columns not in pivots
all_vars = set(range(rref_matrix.shape[1]-1))  # exclude last column (constants)
free_vars = all_vars - set(pivots)
print("Free variable indices:", free_vars)
```

```
Pivot columns: (0, 1)
Free variable indices: {2}
```

5. Another example with infinitely many solutions

$$x + 2y + 3z = 4, \quad 2x + 4y + 6z = 8$$

```
M2 = Matrix([
    [1,2,3,4],
    [2,4,6,8]
])

M2_rref = M2.rref()
print("RREF form:\n", M2_rref[0])
```

```
RREF form:
 Matrix([[1, 2, 3, 4], [0, 0, 0, 0]])
```

The second row becomes all zeros, showing redundancy. Pivot in column 1, free variables in columns 2 and 3.

6. Solving underdetermined systems

If you have more variables than equations, expect free variables. Example:

$$x + y = 3$$

```
M3 = Matrix([[1,1,3]])
print("RREF form:\n", M3.rref()[0])
```

```
RREF form:
 Matrix([[1, 1, 3]])
```

Here, $x = 3 - y$. Variable y is free.

Try It Yourself

1. Take the system:

   $$x + y + z = 2, \quad y + z = 1$$

   Compute its RREF and identify pivot and free variables.

2. Create a random 3×4 system and compute its pivots. How many free variables do you get?

3. For the system:

   $$x - y = 0, \quad 2x - 2y = 0$$

   Verify that the system has infinitely many solutions and describe them in terms of a free variable.

The Takeaway

- Pivots / leading ones mark the basic variables.
- Free variables correspond to non-pivot columns.
- Solutions are written in terms of free variables, showing whether the system has a unique, infinite, or no solution.

25. Solving Consistent Systems (Unique vs. Infinite Solutions)

Now that we can spot pivots and free variables, we can classify systems of equations as having a unique solution or infinitely many solutions (assuming they're consistent). In this lab, we'll practice solving both types.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Unique solution example

System:

$$x + y = 3, \quad 2x - y = 0$$

```
from sympy import Matrix

M = Matrix([
    [1, 1, 3],
    [2, -1, 0]
])

M_rref = M.rref()
print("RREF form:\n", M_rref[0])

# Split into coefficient matrix A and right-hand side b
A = M[:, :2]
b = M[:, 2]
```

```
solution = A.solve_least_squares(b)
print("Solution:", solution)
```

```
RREF form:
 Matrix([[1, 0, 1], [0, 1, 2]])
Solution: Matrix([[1], [2]])
```

2. Infinite solution example

System:

$$x + y + z = 2, \quad 2x + 2y + 2z = 4$$

```
M2 = Matrix([
    [1, 1, 1, 2],
    [2, 2, 2, 4]
])

M2_rref = M2.rref()
print("RREF form:\n", M2_rref[0])
```

```
RREF form:
 Matrix([[1, 1, 1, 2], [0, 0, 0, 0]])
```

Only one pivot $\rightarrow$ two free variables.

Interpretation:

- $x = 2 - y - z$
- y, z are free
- Infinite solutions described by parameters.

3. Classifying consistency

A system is consistent if the RREF does *not* have a row like:

$$[0, 0, 0, c] \quad (c \neq 0)$$

Example consistent system:

```
M3 = Matrix([
    [1, 2, 3],
    [0, 1, 4]
])
print("RREF:\n", M3.rref()[0])
```

```
RREF:
 Matrix([[1, 0, -5], [0, 1, 4]])
```

Example inconsistent system (no solution):

```
M4 = Matrix([
    [1, 1, 2],
    [2, 2, 5]
])
print("RREF:\n", M4.rref()[0])
```

```
RREF:
 Matrix([[1, 1, 0], [0, 0, 1]])
```

The second one ends with `[0,0,1]`, meaning contradiction ($0 = 1$).

4. Quick NumPy comparison

For systems with unique solutions:

```
A = np.array([[1,1],[2,-1]], dtype=float)
b = np.array([3,0], dtype=float)
print("Unique solution with np.linalg.solve:", np.linalg.solve(A,b))
```

```
Unique solution with np.linalg.solve: [1. 2.]
```

For systems with infinite solutions, `np.linalg.solve` will fail, but SymPy handles parametric solutions.

Try It Yourself

1. Solve:

$$x + y + z = 1, \quad 2x + 3y + z = 2$$

   Is the solution unique or infinite?

2. Check consistency of:

$$x + 2y = 3, \quad 2x + 4y = 8$$

3. Build a random 3×4 augmented matrix and compute its RREF. Identify:
   - Does it have a unique solution, infinitely many, or none?

The Takeaway

- Unique solution: pivot in every variable column.
- Infinite solutions: free variables remain, system is still consistent.
- No solution: an inconsistent row appears.

Understanding pivots and free variables gives a complete picture of the solution set.

26. Detecting Inconsistency (When No Solution Exists)

Not all systems of linear equations can be solved. Some are inconsistent, meaning the equations contradict each other. In this lab, we'll learn how to recognize inconsistency using augmented matrices and RREF.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. An inconsistent system

$$x + y = 2, \quad 2x + 2y = 5$$

Notice the second equation looks like a multiple of the first, but the constant doesn't match - contradiction.

```
M = Matrix([
    [1, 1, 2],
    [2, 2, 5]
])

M_rref = M.rref()
print("RREF:\n", M_rref[0])
```

```
RREF:
 Matrix([[1, 1, 0], [0, 0, 1]])
```

RREF gives:

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

The last row means $0 = 1$, so no solution exists.

2. A consistent system (for contrast)

$$x + y = 2, \quad 2x + 2y = 4$$

```
M2 = Matrix([
    [1, 1, 2],
    [2, 2, 4]
])

print("RREF:\n", M2.rref()[0])
```

```
RREF:
 Matrix([[1, 1, 2], [0, 0, 0]])
```

This reduces to one equation and a redundant row of zeros → infinitely many solutions.

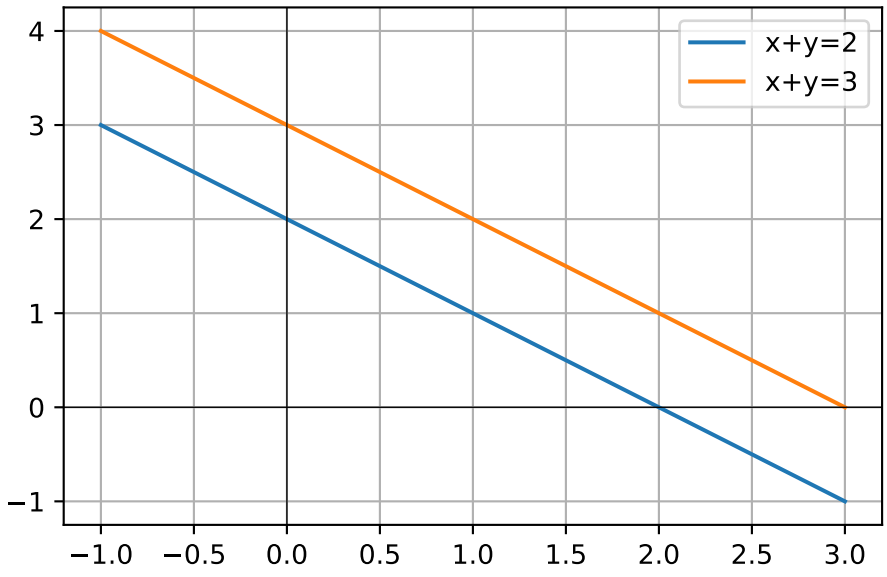
3. Visualizing inconsistency (2D case)

System:

$$x + y = 2 \quad \text{and} \quad x + y = 3$$

These are parallel lines that never meet.

```
import matplotlib.pyplot as plt

x_vals = np.linspace(-1, 3, 100)
y1 = 2 - x_vals
y2 = 3 - x_vals

plt.plot(x_vals, y1, label="x+y=2")
plt.plot(x_vals, y2, label="x+y=3")

plt.legend()
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.show()
```



The two lines are parallel → no solution.

4. Detecting inconsistency automatically

We can scan the RREF for a row of the form $[0, 0, ..., c]$ with $c \neq 0$.

```
def is_inconsistent(M):
    rref_matrix, _ = M.rref()
    for row in rref_matrix.tolist():
        if all(v == 0 for v in row[:-1]) and row[-1] != 0:
            return True
    return False

print("System 1 inconsistent?", is_inconsistent(M))
print("System 2 inconsistent?", is_inconsistent(M2))
```

```
System 1 inconsistent? True
System 2 inconsistent? False
```

Try It Yourself

1. Test the system:

$$x + 2y = 4, \quad 2x + 4y = 10$$

   Write the augmented matrix and check if it's inconsistent.

2. Build a random 2×3 augmented matrix with integer entries. Use `is_inconsistent` to check.

3. Plot two linear equations in 2D. Adjust constants to see when they intersect (consistent) vs when they are parallel (inconsistent).

The Takeaway

- A system is inconsistent if RREF contains a row like $[0, 0, ..., c]$ with $c \neq 0$.
- Geometrically, this means the equations describe parallel lines (2D), parallel planes (3D), or higher-dimensional contradictions.
- Recognizing inconsistency quickly saves time and avoids chasing impossible solutions.

27. Gaussian Elimination by Hand (A Disciplined Procedure)

Gaussian elimination is the systematic way to solve linear systems using row operations. It transforms the augmented matrix into row-echelon form (REF) and then uses back substitution to find solutions. In this lab, we'll walk step by step through the process.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Example system

$$\begin{cases} x + y + z = 6 \\ 2x + 3y + z = 14 \\ x + 2y + 3z = 14 \end{cases}$$

```
A = np.array([
    [1, 1, 1, 6],
    [2, 3, 1, 14],
    [1, 2, 3, 14]
], dtype=float)

print("Initial augmented matrix:\n", A)
```

```
Initial augmented matrix:
 [[ 1.  1.  1.  6.]
 [ 2.  3.  1. 14.]
 [ 1.  2.  3. 14.]]
```

2. Step 1: Get a pivot in the first column

Make the pivot at (0,0) into 1 (it already is). Now eliminate below it.

```
A[1] = A[1] - 2*A[0]    # Row2 → Row2 - 2*Row1
A[2] = A[2] - A[0]      # Row3 → Row3 - Row1
print("After eliminating first column:\n", A)
```

```
After eliminating first column:
 [[ 1.  1.  1.  6.]
 [ 0.  1. -1.  2.]
 [ 0.  1.  2.  8.]]
```

3. Step 2: Pivot in the second column

Make the pivot in row 1, col 1 equal to 1.

```
A[1] = A[1] / A[1,1]
print("After scaling second row:\n", A)
```

```
After scaling second row:
 [[ 1.  1.  1.  6.]
 [ 0.  1. -1.  2.]
 [ 0.  1.  2.  8.]]
```

Now eliminate below:

```
A[2] = A[2] - A[2,1]*A[1]
print("After eliminating second column:\n", A)
```

```
After eliminating second column:
 [[ 1.  1.  1.  6.]
 [ 0.  1. -1.  2.]
 [ 0.  0.  3.  6.]]
```

4. Step 3: Pivot in the third column

Make the bottom-right entry into 1.

```
A[2] = A[2] / A[2,2]
print("After scaling third row:\n", A)
```

```
After scaling third row:
 [[ 1.  1.  1.  6.]
 [ 0.  1. -1.  2.]
 [ 0.  0.  1.  2.]]
```

At this point, the matrix is in row-echelon form (REF).

5. Back substitution

Now solve from the bottom up:

```
z = A[2,3]
y = A[1,3] - A[1,2]*z
x = A[0,3] - A[0,1]*y - A[0,2]*z

print(f"Solution: x={x}, y={y}, z={z}")
```

```
Solution: x=0.0, y=4.0, z=2.0
```

6. Verification

```
coeff = np.array([
    [1,1,1],
    [2,3,1],
    [1,2,3]
], dtype=float)
const = np.array([6,14,14], dtype=float)

print("Check with np.linalg.solve:", np.linalg.solve(coeff,const))
```

```
Check with np.linalg.solve: [0. 4. 2.]
```

The results match.

Try It Yourself

1. Solve:

$$2x + y = 5, \quad 4x - 6y = -2$$

   using Gaussian elimination manually in code.

2. Create a random 3×4 augmented matrix and reduce it step by step, printing after each row operation.

3. Compare your manual elimination to SymPy's RREF with `Matrix.rref()`.

The Takeaway

- Gaussian elimination is a disciplined sequence of row operations.
- It reduces the matrix to row-echelon form, from which back substitution is straightforward.
- This method is the backbone of solving systems by hand and underlies many numerical algorithms.

28. Back Substitution and Solution Sets (Finishing Cleanly)

Once Gaussian elimination reduces a system to row-echelon form (REF), the final step is back substitution. This means solving variables starting from the last equation and working upward. In this lab, we'll practice both unique and infinite solution cases.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Unique solution example

System:

$$x + y + z = 6, \quad 2y + 5z = -4, \quad z = 3$$

Row-echelon form looks like:

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

Solve bottom-up:

```
z = 3
y = (-4 - 5*z)/2
x = 6 - y - z
print(f"Solution: x={x}, y={y}, z={z}")
```

```
Solution: x=12.5, y=-9.5, z=3
```

2. Infinite solution example

System:

$$x + y + z = 2, \quad 2x + 2y + 2z = 4$$

After elimination:

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

This means:

- Equation: $x + y + z = 2$.
- Free variables: choose y and z.

Let $y = s, z = t$. Then:

$$x = 2 - s - t$$

So the solution set is:

```
from sympy import symbols
s, t = symbols('s t')
x = 2 - s - t
y = s
z = t
print("General solution:")
print("x =", x, ", y =", y, ", z =", z)
```

```
General solution:
x = -s - t + 2 , y = s , z = t
```

3. Consistency check with RREF

We can use SymPy to confirm solution sets:

```
M = Matrix([
    [1,1,1,2],
    [2,2,2,4]
])

print("RREF form:\n", M.rref()[0])
```

```
RREF form:
 Matrix([[1, 1, 1, 2], [0, 0, 0, 0]])
```

The second row disappears, showing infinite solutions.

4. Encoding solution sets

General solutions are often written in parametric vector form.

For the infinite solution above:

$$(x, y, z) = (2, 0, 0) + s(-1, 1, 0) + t(-1, 0, 1)$$

This shows the solution space is a plane in $\mathbb{R}^3$.

Try It Yourself

1. Solve:

$$x + 2y = 5, \quad y = 1$$

   Do back substitution by hand and check with NumPy.

2. Take the system:

$$x + y + z = 1, \quad 2x + 2y + 2z = 2$$

   Write its solution set in parametric form.

3. Use `Matrix.rref()` on a 3×4 random augmented matrix. Identify pivot and free variables, then describe the solution set.

The Takeaway

- Back substitution is the cleanup step after Gaussian elimination.
- It reveals whether the system has a unique solution or infinitely many.
- Solutions can be expressed explicitly (unique case) or parametrically (infinite case).

29. Rank and Its First Meaning (Pivots as Information)

The rank of a matrix tells us how much independent information it contains. Rank is one of the most important concepts in linear algebra because it connects to pivots, independence, dimension, and the number of solutions to a system.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Rank definition The rank is the number of pivots (leading ones) in the row-echelon form of a matrix.

Example:

```
A = Matrix([
    [1, 2, 3],
    [2, 4, 6],
    [1, 1, 1]
])

print("RREF:\n", A.rref()[0])
print("Rank of A:", A.rank())
```

```
RREF:
 Matrix([[1, 0, -1], [0, 1, 2], [0, 0, 0]])
Rank of A: 2
```

- The second row is a multiple of the first, so the rank is less than 3.
- Only two independent rows $\rightarrow$ rank $= 2$.

2. Rank and solutions to $A \cdot x = b$

Consider:

$$\begin{cases} x + y + z = 3 \\ 2x + 2y + 2z = 6 \\ x - y = 0 \end{cases}$$

```
M = Matrix([
    [1, 1, 1, 3],
    [2, 2, 2, 6],
    [1, -1, 0, 0]
])

print("RREF:\n", M.rref()[0])
print("Rank of coefficient matrix:", M[:, :-1].rank())
print("Rank of augmented matrix:", M.rank())
```

```
RREF:
 Matrix([[1, 0, 1/2, 3/2], [0, 1, 1/2, 3/2], [0, 0, 0, 0]])
Rank of coefficient matrix: 2
Rank of augmented matrix: 2
```

- If rank(A) = rank([A|b]) = number of variables → unique solution.
- If rank(A) = rank([A|b]) < number of variables → infinite solutions.
- If rank(A) < rank([A|b]) → no solution.

3. NumPy comparison

```
A = np.array([
    [1, 2, 3],
    [2, 4, 6],
    [1, 1, 1]
], dtype=float)

print("Rank with NumPy:", np.linalg.matrix_rank(A))
```

```
Rank with NumPy: 2
```

4. Rank as “dimension of information”

The rank equals:

- The number of independent rows.
- The number of independent columns.
- The dimension of the column space.

```
B = Matrix([
    [1,2],
    [2,4],
    [3,6]
])

print("Rank of B:", B.rank())
```

```
Rank of B: 1
```

All columns are multiples → only one independent direction → rank = 1.

Try It Yourself

1. Compute the rank of:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

   What do you expect?

2. Create a random 4×4 matrix with `np.random.randint`. Compute its rank with both SymPy and NumPy.

3. Test solution consistency using rank: build a system where rank(A) rank([A|b]) and show it has no solution.

The Takeaway

- Rank = number of pivots = dimension of independent information.
- Rank reveals whether a system has no solution, one solution, or infinitely many.
- Rank connects algebra (pivots) with geometry (dimension of subspaces).

30. LU Factorization (Elimination Captured as L and U)

Gaussian elimination can be recorded in a neat factorization:

$$A = LU$$

where L is a lower triangular matrix (recording the multipliers we used) and U is an upper triangular matrix (the result of elimination). This is called LU factorization. It's a powerful tool for solving systems efficiently.

Set Up Your Lab

```
import numpy as np
from scipy.linalg import lu
```

Step-by-Step Code Walkthrough

1. Example matrix

```
A = np.array([
    [2, 3, 1],
    [4, 7, 7],
    [6, 18, 22]
], dtype=float)

print("Matrix A:\n", A)
```

```
Matrix A:
 [[ 2.  3.  1.]
 [ 4.  7.  7.]
 [ 6. 18. 22.]]
```

2. LU decomposition with SciPy

```
P, L, U = lu(A)

print("Permutation matrix P:\n", P)
print("Lower triangular L:\n", L)
print("Upper triangular U:\n", U)
```

```
Permutation matrix P:
 [[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
Lower triangular L:
 [[1.         0.         0.        ]
 [0.66666667 1.         0.        ]
 [0.33333333 0.6        1.        ]]
Upper triangular U:
 [[ 6.         18.         22.        ]
 [ 0.         -5.         -7.66666667]
 [ 0.          0.         -1.73333333]]
```

Here, P handles row swaps (partial pivoting), L is lower triangular, and U is upper triangular.

3. Verifying the factorization

```
reconstructed = P @ L @ U
print("Does P·L·U equal A?\n", np.allclose(reconstructed, A))
```

```
Does P·L·U equal A?
 True
```

4. Solving a system with LU

Suppose we want to solve $Ax = b$. Instead of working directly with A, we solve in two steps:

1. Solve $Ly = Pb$ (forward substitution).
2. Solve $Ux = y$ (back substitution).

```
b = np.array([1, 2, 3], dtype=float)

# Step 1: Pb
Pb = P @ b

# Step 2: forward substitution Ly = Pb
y = np.linalg.solve(L, Pb)

# Step 3: back substitution Ux = y
x = np.linalg.solve(U, y)

print("Solution x:", x)
```

```
Solution x: [ 0.5 -0.  -0. ]
```

5. Efficiency advantage

If we have to solve many systems with the same A but different b, we only compute LU once, then reuse it. This saves a lot of computation.

6. NumPy's built-in rank-revealing factorization

While NumPy doesn't have `lu` directly, it works seamlessly with SciPy. For large matrices, LU decomposition is the backbone of solvers like `np.linalg.solve`.

Try It Yourself

1. Compute LU decomposition for

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{bmatrix}$$

   Verify $P \cdot L \cdot U = A$.

2. Solve $Ax = b$ with

$$b = [3, 7, 8]$$

   using LU factorization.

3. Compare solving with LU factorization vs directly using `np.linalg.solve(A,b)`. Are the answers the same?

The Takeaway

- LU factorization captures Gaussian elimination in matrix form: $A = P \cdot L \cdot U$.
- It allows fast repeated solving of systems with different right-hand sides.
- LU decomposition is a core technique in numerical linear algebra and the basis of many solvers.

Chapter 4. Vector Spaces and Subspaces

31. Axioms of Vector Spaces (What "Space" Really Means)

Vector spaces generalize what we've been doing with vectors and matrices. Instead of just $\mathbb{R}^n$, a vector space is any collection of objects (vectors) where addition and scalar multiplication follow specific axioms (rules). In this lab, we'll explore these axioms concretely with Python.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Vector space example: $\mathbb{R}^2$

Let's check two rules (axioms): closure under addition and scalar multiplication.

```
u = np.array([1, 2])
v = np.array([3, -1])

# Closure under addition
print("u + v =", u + v)

# Closure under scalar multiplication
k = 5
print("k * u =", k * u)
```

```
u + v = [4 1]
k * u = [ 5 10]
```

Both results are still in $\mathbb{R}^2$.

2. Zero vector and additive inverses

Every vector space must contain a zero vector, and every vector must have an additive inverse.

```
zero = np.array([0, 0])
inverse_u = -u
print("Zero vector:", zero)
print("u + (-u) =", u + inverse_u)
```

```
Zero vector: [0 0]
u + (-u) = [0 0]
```

3. Distributive and associative properties

Check:

- $a(u+v) = au + av$
- $(a+b)u = au + bu$

```
a, b = 2, 3

lhs1 = a * (u + v)
rhs1 = a*u + a*v
print("a(u+v) =", lhs1, ", au+av =", rhs1)

lhs2 = (a+b) * u
rhs2 = a*u + b*u
print("(a+b)u =", lhs2, ", au+bu =", rhs2)
```

```
a(u+v) = [8 2] , au+av = [8 2]
(a+b)u = [ 5 10] , au+bu = [ 5 10]
```

Both equalities hold → distributive laws confirmed.

4. A set that fails to be a vector space

Consider only positive numbers with normal addition and scalar multiplication.

```
positive_numbers = [1, 2, 3]
try:
    print("Closure under negatives?", -1 * np.array(positive_numbers))
except Exception as e:
    print("Error:", e)
```

```
Closure under negatives? [-1 -2 -3]
```

Negative results leave the set → not a vector space.

5. Python helper to check axioms

We can quickly check if a set of vectors is closed under addition and scalar multiplication.

```
def check_closure(vectors, scalars):
    for v in vectors:
        for u in vectors:
            if not any(np.array_equal(v+u, w) for w in vectors):
                return False
        for k in scalars:
            if not any(np.array_equal(k*v, w) for w in vectors):
                return False
    return True

vectors = [np.array([0,0]), np.array([1,0]), np.array([0,1]), np.array([1,1])]
scalars = [0,1,-1]
print("Closed under addition and scalar multiplication?", check_closure(vectors, scalars))
```

```
Closed under addition and scalar multiplication? False
```

This small set is closed → it forms a vector space (a subspace of $\mathbb{R}^2$).

Try It Yourself

1. Verify that $\mathbb{R}^3$ satisfies the vector space axioms using random vectors.
2. Test whether the set of all 2×2 matrices forms a vector space under normal addition and scalar multiplication.
3. Find an example of a set that fails closure (e.g., integers under division).

The Takeaway

- A vector space is any set where addition and scalar multiplication satisfy 10 standard axioms.
- These rules ensure consistent algebraic behavior.
- Many objects beyond arrows in $\mathbb{R}^n$ (like polynomials or matrices) are vector spaces too.

32. Subspaces, Column Space, and Null Space (Where Solutions Live)

A subspace is a smaller vector space sitting inside a bigger one. For matrices, two subspaces show up all the time:

- Column space: all combinations of the matrix's columns (possible outputs of Ax).
- Null space: all vectors x such that $Ax = 0$ (inputs that vanish).

This lab explores both in Python.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Column space basics

Take:

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

```
A = Matrix([
    [1,2],
    [2,4],
    [3,6]
])

print("Matrix A:\n", A)
print("Column space basis:\n", A.columnspace())
print("Rank (dimension of column space):", A.rank())
```

```
Matrix A:
 Matrix([[1, 2], [2, 4], [3, 6]])
Column space basis:
 [Matrix([
[1],
```

```
[2],
[3]])]
Rank (dimension of column space): 1
```

- The second column is a multiple of the first → column space has dimension 1.
- All outputs of Ax lie on a line in $\mathbb{R}^3$.

2. Null space basics

```
print("Null space basis:\n", A.nullspace())
```

```
Null space basis:
 [Matrix([
[-2],
[ 1]])]
```

The null space contains all x where $Ax = 0$. Here, the null space is 1-dimensional (vectors like $[-2, 1]$).

3. A full-rank example

```
B = Matrix([
    [1,0,0],
    [0,1,0],
    [0,0,1]
])

print("Column space basis:\n", B.columnspace())
print("Null space basis:\n", B.nullspace())
```

```
Column space basis:
 [Matrix([
[1],
[0],
[0]]), Matrix([
[0],
[1],
[0]]), Matrix([
[0],
[0],
[1]])]
Null space basis:
 []
```

- Column space = all of $\mathbb{R}^3$.
- Null space = only the zero vector.

4. Geometry link

For A (rank 1, 2 columns):

- Column space: line in $\mathbb{R}^3$.
- Null space: line in $\mathbb{R}^2$.

Together they explain the system $Ax = b$:

- If b is outside the column space, no solution exists.
- If b is inside, solutions differ by a vector in the null space.

5. Quick NumPy version

NumPy doesn't directly give null space, but we can compute it with SVD.

```
from numpy.linalg import svd

A = np.array([[1,2],[2,4],[3,6]], dtype=float)
U, S, Vt = svd(A)

tol = 1e-10
null_mask = (S <= tol)
null_space = Vt.T[:, null_mask]
print("Null space (via SVD):\n", null_space)
```

```
Null space (via SVD):
 [[-0.89442719]
 [ 0.4472136 ]]
```

Try It Yourself

1. Find the column space and null space of

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

   How many dimensions does each have?

2. Generate a random 3×3 matrix. Compute its rank, column space, and null space.

3. Solve $Ax = b$ with

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

and describe why it has infinitely many solutions.

The Takeaway

- The column space = all possible outputs of a matrix.
- The null space = all inputs that map to zero.
- These subspaces give the complete picture of what a matrix does.

33. Span and Generating Sets (Coverage of a Space)

The span of a set of vectors is all the linear combinations you can make from them. If a set of vectors can "cover" a whole space, we call it a generating set. This lab shows how to compute and visualize spans.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Span in $\mathbb{R}^2$

Two vectors that aren't multiples span the whole plane.

```
u = np.array([1, 0])
v = np.array([0, 1])

M = Matrix.hstack(Matrix(u), Matrix(v))
print("Rank:", M.rank())
```

```
Rank: 2
```

Rank $= 2 \rightarrow$ the span of $\{u, v\}$ is all of $\mathbb{R}^2$.

2. Dependent vectors (smaller span)

```
u = np.array([1, 2])
v = np.array([2, 4])

M = Matrix.hstack(Matrix(u), Matrix(v))
print("Rank:", M.rank())
```

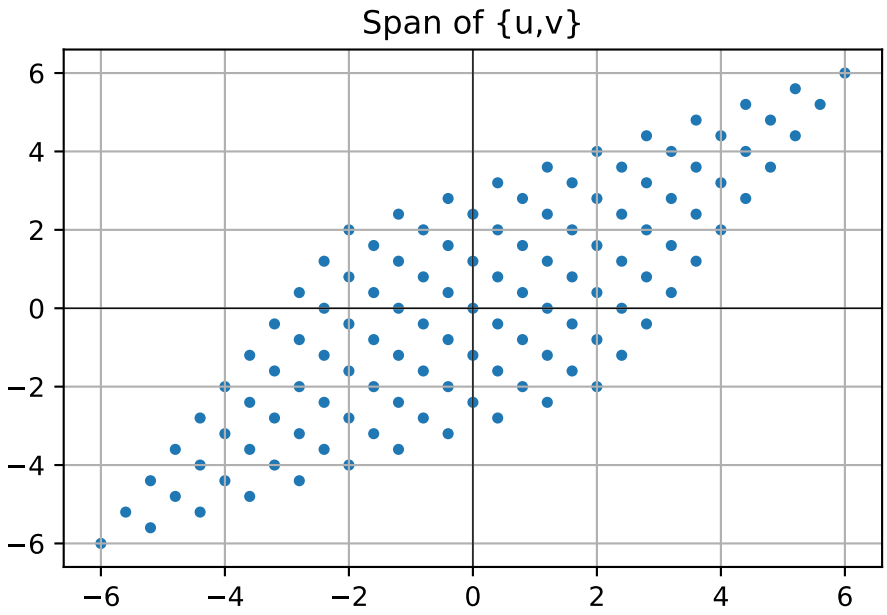
```
Rank: 1
```

Rank $= 1 \rightarrow$ these vectors only span a line.

3. Visualizing a span

Let's see what the span of two vectors looks like.

```
u = np.array([1, 2])
v = np.array([2, 1])

coeffs = np.linspace(-2, 2, 11)
points = []
for a in coeffs:
    for b in coeffs:
        points.append(a*u + b*v)
points = np.array(points)

plt.scatter(points[:,0], points[:,1], s=10)
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.title("Span of {u,v}")
plt.grid()
plt.show()
```



You'll see a filled grid - the entire plane, because the two vectors are independent.

4. Generating set of a space

For $\mathbb{R}^3$:

```
basis = [Matrix([1,0,0]), Matrix([0,1,0]), Matrix([0,0,1])]
M = Matrix.hstack(*basis)
print("Rank:", M.rank())
```

```
Rank: 3
```

Rank $= 3 \rightarrow$ this set spans the whole space.

5. Testing if a vector is in the span

Example: Is $[3, 5]$ in the span of $[1, 2]$ and $[2, 1]$?

```
u = Matrix([1,2])
v = Matrix([2,1])
target = Matrix([3,5])

M = Matrix.hstack(u,v)
solution = M.gauss_jordan_solve(target)
print("Coefficients (a,b):", solution)
```

```
Coefficients (a,b): (Matrix([
[7/3],
[1/3]]), Matrix(0, 1, []))
```

If a solution exists, the target is in the span.

Try It Yourself

1. Test if $[4, 6]$ is in the span of $[1, 2]$.
2. Visualize the span of $[1, 0, 0]$ and $[0, 1, 0]$ in $\mathbb{R}^3$. What does it look like?
3. Create a random 3×3 matrix. Use `rank()` to check if its columns span $\mathbb{R}^3$.

The Takeaway

- Span = all linear combinations of a set of vectors.
- Independent vectors span bigger spaces; dependent ones collapse to smaller spaces.
- Generating sets are the foundation of bases and coordinate systems.

34. Linear Independence and Dependence (No Redundancy vs. Redundancy)

A set of vectors is linearly independent if none of them can be written as a combination of the others. If at least one can, the set is dependent. This distinction tells us whether a set of vectors has redundancy.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Independent vectors example

```
v1 = Matrix([1, 0, 0])
v2 = Matrix([0, 1, 0])
v3 = Matrix([0, 0, 1])

M = Matrix.hstack(v1, v2, v3)
print("Rank:", M.rank(), " Number of vectors:", M.shape[1])
```

```
Rank: 3  Number of vectors: 3
```

Rank = 3, number of vectors = 3 → all independent.

2. Dependent vectors example

```
v1 = Matrix([1, 2, 3])
v2 = Matrix([2, 4, 6])
v3 = Matrix([3, 6, 9])

M = Matrix.hstack(v1, v2, v3)
print("Rank:", M.rank(), " Number of vectors:", M.shape[1])
```

```
Rank: 1  Number of vectors: 3
```

Rank = 1, number of vectors = 3 → they're dependent (multiples of each other).

3. Checking dependence automatically

A quick test: if rank < number of vectors → dependent.

```
def check_independence(vectors):
    M = Matrix.hstack(*vectors)
    return M.rank() == M.shape[1]

print("Independent?", check_independence([Matrix([1,0]), Matrix([0,1])]))
print("Independent?", check_independence([Matrix([1,2]), Matrix([2,4])]))
```

```
Independent? True
Independent? False
```

4. Solving for dependence relation

If vectors are dependent, we can find coefficients $c_1, c_2, ...$ such that

$$c_1 v_1 + c_2 v_2 + ... + c_k v_k = 0$$

with some $c_i \neq 0$.

```
M = Matrix.hstack(Matrix([1,2]), Matrix([2,4]))
null_space = M.nullspace()
print("Dependence relation (coefficients):", null_space)
```

```
Dependence relation (coefficients): [Matrix([
[-2],
[ 1]])]
```

This shows the exact linear relation.

5. Random example

```
np.random.seed(0)
R = Matrix(np.random.randint(-3, 4, (3,3)))
print("Random matrix:\n", R)
print("Rank:", R.rank())
```

```
Random matrix:
 Matrix([[1, 2, -3], [0, 0, 0], [-2, 0, 2]])
Rank: 2
```

Depending on the rank, the columns may be independent (rank = 3) or dependent (rank < 3).

Try It Yourself

1. Test if $[1, 1, 0], [0, 1, 1], [1, 2, 1]$ are independent.
2. Generate 4 random vectors in $\mathbb{R}^3$. Can they ever be independent? Why or why not?
3. Find the dependence relation for $[2, 4], [3, 6]$.

The Takeaway

- Independent set: no redundancy, each vector adds a new direction.
- Dependent set: at least one vector is unnecessary (it lies in the span of others).
- Independence is the key to defining basis and dimension.

35. Basis and Coordinates (Naming Every Vector Uniquely)

A basis is a set of independent vectors that span a space. It's like choosing a coordinate system: every vector in the space can be expressed uniquely as a combination of basis vectors. In this lab, we'll see how to find bases and compute coordinates relative to them.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Standard basis in $\mathbb{R}^3$

```
e1 = Matrix([1,0,0])
e2 = Matrix([0,1,0])
e3 = Matrix([0,0,1])

M = Matrix.hstack(e1, e2, e3)
print("Rank:", M.rank())
```

```
Rank: 3
```

These three independent vectors form the standard basis of $\mathbb{R}^3$. Any vector like $[2, 5, -1]$ can be expressed as

$$2e_1 + 5e_2 - 1e_3$$

2. Finding a basis from dependent vectors

```
v1 = Matrix([1,2,3])
v2 = Matrix([2,4,6])
v3 = Matrix([1,0,1])

M = Matrix.hstack(v1,v2,v3)
print("Column space basis:", M.columnspace())
```

```
Column space basis: [Matrix([
[1],
[2],
[3]]), Matrix([
[1],
[0],
[1]])]
```

SymPy extracts independent columns automatically. This gives a basis for the column space.

3. Coordinates relative to a basis

Suppose basis $= \{[1,0],[1,1]\}$. Express vector $[3,5]$ in this basis.

```
B = Matrix.hstack(Matrix([1,0]), Matrix([1,1]))
target = Matrix([3,5])

coords = B.solve_least_squares(target)
print("Coordinates in basis B:", coords)
```

```
Coordinates in basis B: Matrix([[-2], [5]])
```

So $[3,5] = 3 \cdot [1,0] + 2 \cdot [1,1]$.

4. Basis change

If we switch to a different basis, coordinates change but the vector stays the same.

```
new_basis = Matrix.hstack(Matrix([2,1]), Matrix([1,2]))
coords_new = new_basis.solve_least_squares(target)
print("Coordinates in new basis:", coords_new)
```

```
Coordinates in new basis: Matrix([[1/3], [7/3]])
```

5. Random example

Generate 3 random vectors in $\mathbb{R}^3$. Check if they form a basis.

```
np.random.seed(1)
R = Matrix(np.random.randint(-3,4,(3,3)))
print("Random matrix:\n", R)
print("Rank:", R.rank())
```

```
Random matrix:
 Matrix([[2, 0, 1], [-3, -2, 0], [2, -3, -3]])
Rank: 3
```

If rank $= 3 \rightarrow$ basis for $\mathbb{R}^3$. Otherwise, only span a subspace.

Try It Yourself

1. Check if $[1, 2], [3, 4]$ form a basis of $\mathbb{R}^2$.
2. Express vector $[7, 5]$ in that basis.
3. Create 4 random vectors in $\mathbb{R}^3$. Find a basis for their span.

The Takeaway

- A basis = minimal set of vectors that span a space.
- Every vector has a unique coordinate representation in a given basis.
- Changing bases changes the coordinates, not the vector itself.

36. Dimension (How Many Directions)

The dimension of a vector space is the number of independent directions it has. Formally, it's the number of vectors in any basis of the space. Dimension tells us the "size" of a space in terms of degrees of freedom.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Dimension of $\mathbb{R}^n$

The dimension of $\mathbb{R}^n$ is n.

```
n = 4
basis = [Matrix.eye(n)[:,i] for i in range(n)]
print("Basis for R^4:", basis)
print("Dimension of R^4:", len(basis))
```

```
Basis for R^4: [Matrix([
[1],
[0],
[0],
[0]]), Matrix([
[0],
[1],
[0],
[0]]), Matrix([
[0],
[0],
[1],
[0]]), Matrix([
[0],
[0],
[0],
[1]])]
Dimension of R^4: 4
```

Each standard unit vector adds one independent direction → dimension = 4.

2. Dimension via rank

The rank of a matrix equals the dimension of its column space.

```
A = Matrix([
    [1,2,3],
    [2,4,6],
    [1,0,1]
])

print("Rank (dimension of column space):", A.rank())
```

```
Rank (dimension of column space): 2
```

Here, rank $= 2 \rightarrow$ the column space is a 2D plane inside $\mathbb{R}^3$.

3. Null space dimension

The null space dimension is given by:

$$\dim(\text{Null}(A)) = \#\text{variables} - \text{rank}(A)$$

```
print("Null space basis:", A.nullspace())
print("Dimension of null space:", len(A.nullspace()))
```

```
Null space basis: [Matrix([
[-1],
[-1],
[ 1]])]
Dimension of null space: 1
```

This is the number of free variables in a solution.

4. Dimension in practice

- A line through the origin in $\mathbb{R}^3$ has dimension 1.
- A plane through the origin has dimension 2.
- The whole $\mathbb{R}^3$ has dimension 3.

Example:

```
v1 = Matrix([1,2,3])
v2 = Matrix([2,4,6])
span = Matrix.hstack(v1,v2)
print("Dimension of span:", span.rank())
```

```
Dimension of span: 1
```

Result $= 1 \rightarrow$ they only generate a line.

5. Random example

```
np.random.seed(2)
R = Matrix(np.random.randint(-3,4,(4,4)))
print("Random 4x4 matrix:\n", R)
print("Column space dimension:", R.rank())
```

```
Random 4x4 matrix:
 Matrix([[-3, 2, -3, 3], [0, -1, 0, -3], [-1, -2, 0, 2], [-1, 1, 1, 1]])
Column space dimension: 4
```

Rank may be 4 (full space) or smaller (collapsed).

Try It Yourself

1. Find the dimension of the column space of

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

2. Compute the dimension of the null space of a 3×3 singular matrix.
3. Generate a 5×3 random matrix and compute its column space dimension.

The Takeaway

- Dimension = number of independent directions.
- Found by counting basis vectors (or rank).
- Dimensions describe lines (1D), planes (2D), and higher subspaces inside larger spaces.

37. Rank–Nullity Theorem (Dimensions That Add Up)

The rank–nullity theorem ties together the dimension of the column space and the null space of a matrix. It says:

$$\text{rank}(A) + \text{nullity}(A) = \text{number of columns of } A$$

This is a powerful consistency check in linear algebra.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Simple 3×3 example

```
A = Matrix([
    [1, 2, 3],
    [2, 4, 6],
    [1, 0, 1]
])

rank = A.rank()
nullity = len(A.nullspace())
print("Rank:", rank)
print("Nullity:", nullity)
print("Rank + Nullity =", rank + nullity)
print("Number of columns =", A.shape[1])
```

```
Rank: 2
Nullity: 1
Rank + Nullity = 3
Number of columns = 3
```

You should see that rank + nullity = 3, the number of columns.

2. Full-rank case

```
B = Matrix([
    [1,0,0],
    [0,1,0],
    [0,0,1]
])

print("Rank:", B.rank())
print("Nullity:", len(B.nullspace()))
```

```
Rank: 3
Nullity: 0
```

- Rank = 3 (all independent).
- Nullity = 0 (only zero solution to $Bx = 0$).
- Rank + Nullity = 3 columns.

3. Wide matrix (more columns than rows)

```
C = Matrix([
    [1,2,3,4],
    [0,1,1,2],
    [0,0,0,0]
])

rank = C.rank()
nullity = len(C.nullspace())
print("Rank:", rank, " Nullity:", nullity, " Columns:", C.shape[1])
```

```
Rank: 2  Nullity: 2  Columns: 4
```

Here, nullity > 0 because there are more variables than independent equations.

4. Verifying with random matrices

```
np.random.seed(3)
R = Matrix(np.random.randint(-3,4,(4,5)))
print("Random 4x5 matrix:\n", R)
print("Rank + Nullity =", R.rank() + len(R.nullspace()))
print("Number of columns =", R.shape[1])
```

```
Random 4x5 matrix:
 Matrix([[-1, -3, -2, 0, -3], [-3, -3, 2, 2, 0], [-1, 0, -2, -2, -1], [2, 3, -
3, 1, 1]])
Rank + Nullity = 5
Number of columns = 5
```

Always consistent: rank + nullity = number of columns.

5. Geometric interpretation

For an $m \times n$ matrix:

- Rank(A) = dimension of outputs (column space).
- Nullity(A) = dimension of hidden directions that collapse to 0.
- Together, they use up all the "input dimensions" (n).

Try It Yourself

1. Compute rank and nullity of

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

   Check the theorem.

2. Create a 2×4 random integer matrix. Confirm that rank + nullity = 4.
3. Explain why a tall full-rank 5×3 matrix must have nullity = 0.

The Takeaway

- Rank + Nullity = number of columns (always true).
- Rank measures independent outputs; nullity measures hidden freedom.
- This theorem connects solutions of $Ax = 0$ with the structure of A.

38. Coordinates Relative to a Basis (Changing the "Ruler")

Once we choose a basis, every vector can be described with coordinates relative to that basis. This is like changing the "ruler" we use to measure vectors. In this lab, we'll practice computing coordinates in different bases.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Standard basis coordinates

Vector $v = [4, 5]$ in $\mathbb{R}^2$:

```
v = Matrix([4,5])
e1 = Matrix([1,0])
e2 = Matrix([0,1])

B = Matrix.hstack(e1,e2)
coords = B.solve_least_squares(v)
print("Coordinates in standard basis:", coords)
```

```
Coordinates in standard basis: Matrix([[4], [5]])
```

Result is just $[4, 5]$. Easy - the standard basis matches the components directly.

2. Non-standard basis

Suppose basis $= \{[1, 1], [1, -1]\}$. Express $v = [4, 5]$ in this basis.

```
B2 = Matrix.hstack(Matrix([1,1]), Matrix([1,-1]))
coords2 = B2.solve_least_squares(v)
print("Coordinates in new basis:", coords2)
```

```
Coordinates in new basis: Matrix([[9/2], [-1/2]])
```

Now v has different coordinates.

3. Changing coordinates back

To reconstruct the vector from coordinates:

```
reconstructed = B2 * coords2
print("Reconstructed vector:", reconstructed)
```

```
Reconstructed vector: Matrix([[4], [5]])
```

It matches the original $[4, 5]$.

4. Random basis in $\mathbb{R}^3$

```
basis = Matrix.hstack(
    Matrix([1,0,1]),
    Matrix([0,1,1]),
    Matrix([1,1,0])
)
v = Matrix([2,3,4])

coords = basis.solve_least_squares(v)
print("Coordinates of v in random basis:", coords)
```

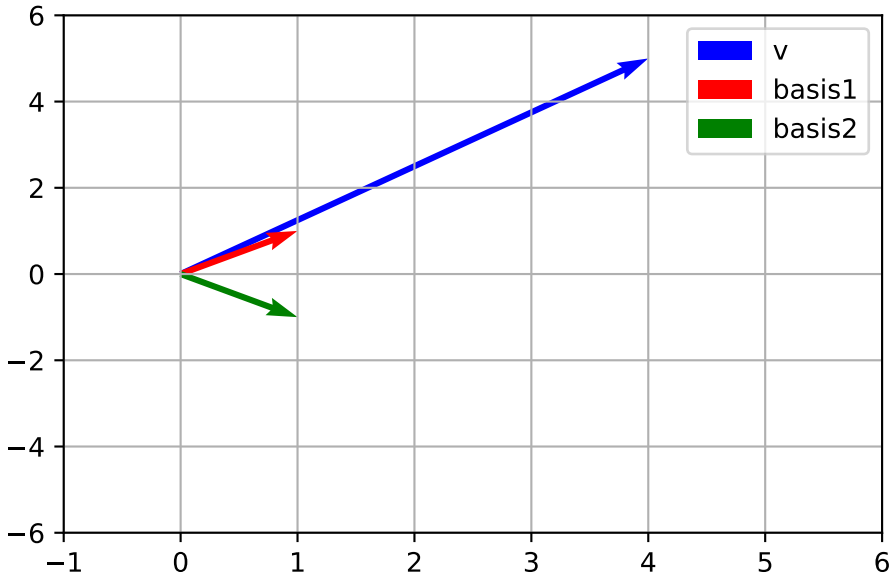
```
Coordinates of v in random basis: Matrix([[3/2], [5/2], [1/2]])
```

Any independent set of 3 vectors in $\mathbb{R}^3$ works as a basis.

5. Visualization in 2D

Let's compare coordinates in two bases.

```
import matplotlib.pyplot as plt

v = np.array([4,5])
b1 = np.array([1,1])
b2 = np.array([1,-1])

plt.quiver(0,0,v[0],v[1],angles='xy',scale_units='xy',scale=1,color='blue',label='v')
plt.quiver(0,0,b1[0],b1[1],angles='xy',scale_units='xy',scale=1,color='red',label='basis1')
plt.quiver(0,0,b2[0],b2[1],angles='xy',scale_units='xy',scale=1,color='green',label='basis2')

plt.xlim(-1,6)
plt.ylim(-6,6)
plt.legend()
plt.grid()
plt.show()
```



Even though the basis vectors look different, they span the same space, and v can be expressed in terms of them.

Try It Yourself

1. Express $[7, 3]$ in the basis $\{[2, 0], [0, 3]\}$.
2. Pick three independent random vectors in $\mathbb{R}^3$. Write down the coordinates of $[1, 2, 3]$ in that basis.
3. Verify that reconstructing always gives the original vector.

The Takeaway

- A basis provides a coordinate system for vectors.
- Coordinates depend on the basis, but the underlying vector doesn't change.
- Changing the basis is like changing the "ruler" you measure vectors with.

39. Change-of-Basis Matrices (Moving Between Coordinate Systems)

When we switch from one basis to another, we need a change-of-basis matrix. This matrix acts like a translator: it converts coordinates in one system to coordinates in another.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Two bases in $\mathbb{R}^2$

Let's define:

- Basis $B = \{[1, 0], [0, 1]\}$ (standard basis).
- Basis $C = \{[1, 1], [1, -1]\}$.

```
B = Matrix.hstack(Matrix([1,0]), Matrix([0,1]))
C = Matrix.hstack(Matrix([1,1]), Matrix([1,-1]))
```

2. Change-of-basis matrix

The matrix that converts C-coordinates → standard coordinates is just C.

```
print("C (basis matrix):\n", C)
```

```
C (basis matrix):
 Matrix([[1, 1], [1, -1]])
```

To go the other way (standard → C), we compute the inverse of C.

```
C_inv = C.inv()
print("C inverse:\n", C_inv)
```

```
C inverse:
 Matrix([[1/2, 1/2], [1/2, -1/2]])
```

3. Converting coordinates

Vector $v = [4, 5]$.

- In standard basis:

```
v = Matrix([4,5])
coords_in_standard = v
print("Coordinates in standard basis:", coords_in_standard)
```

```
Coordinates in standard basis: Matrix([[4], [5]])
```

- In basis C:

```
coords_in_C = C_inv * v
print("Coordinates in C basis:", coords_in_C)
```

```
Coordinates in C basis: Matrix([[9/2], [-1/2]])
```

- Convert back:

```
reconstructed = C * coords_in_C
print("Reconstructed vector:", reconstructed)
```

```
Reconstructed vector: Matrix([[4], [5]])
```

The reconstruction matches the original vector.

4. General formula

If P is the change-of-basis matrix from basis B to basis C:

$$[v]_C = P^{-1}[v]_B$$

$$[v]_B = P[v]_C$$

Here, P is the matrix of new basis vectors written in terms of the old basis.

5. Random 3D example

```
B = Matrix.eye(3)  # standard basis
C = Matrix.hstack(
    Matrix([1,0,1]),
    Matrix([0,1,1]),
    Matrix([1,1,0])
)

v = Matrix([2,3,4])

C_inv = C.inv()
coords_in_C = C_inv * v
print("Coordinates in new basis C:", coords_in_C)

print("Back to standard:", C * coords_in_C)
```

```
Coordinates in new basis C: Matrix([[3/2], [5/2], [1/2]])
Back to standard: Matrix([[2], [3], [4]])
```

Try It Yourself

1. Convert $[7, 3]$ from the standard basis to the basis $\{[2, 0], [0, 3]\}$.
2. Pick a random invertible 3×3 matrix as a basis. Write a vector in that basis, then convert it back to the standard basis.
3. Prove that converting back and forth always returns the same vector.

The Takeaway

- A change-of-basis matrix converts coordinates between bases.
- Going from new basis $\rightarrow$ old basis uses the basis matrix.
- Going from old basis $\rightarrow$ new basis requires its inverse.
- The vector itself never changes - only the description of it does.

40. Affine Subspaces (Lines and Planes Not Through the Origin)

So far, subspaces always passed through the origin. But many familiar objects - like lines offset from the origin or planes floating in space - are affine subspaces. They look like subspaces, just shifted away from zero.

Set Up Your Lab

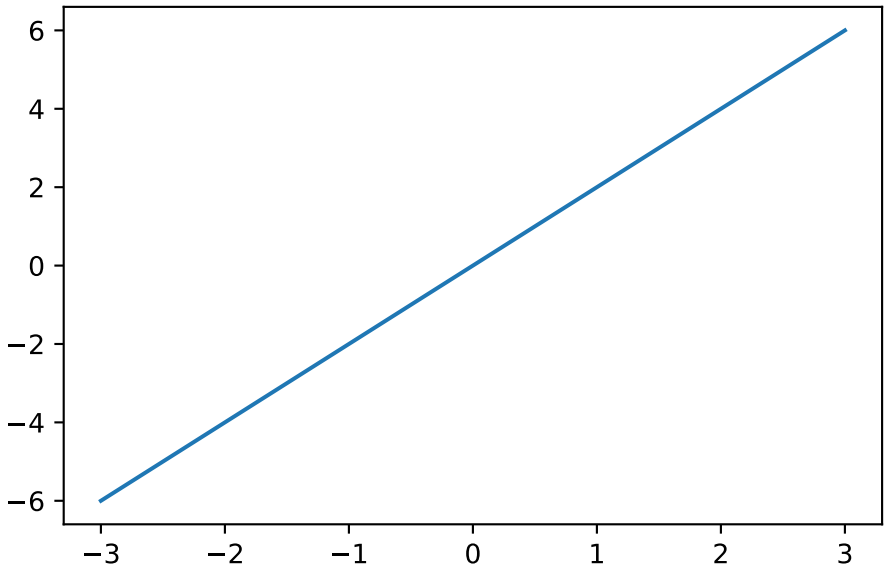
```
import numpy as np
import matplotlib.pyplot as plt
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Line through the origin (a subspace)

$$L = \{t \cdot [1, 2] : t \in \mathbb{R}\}$$

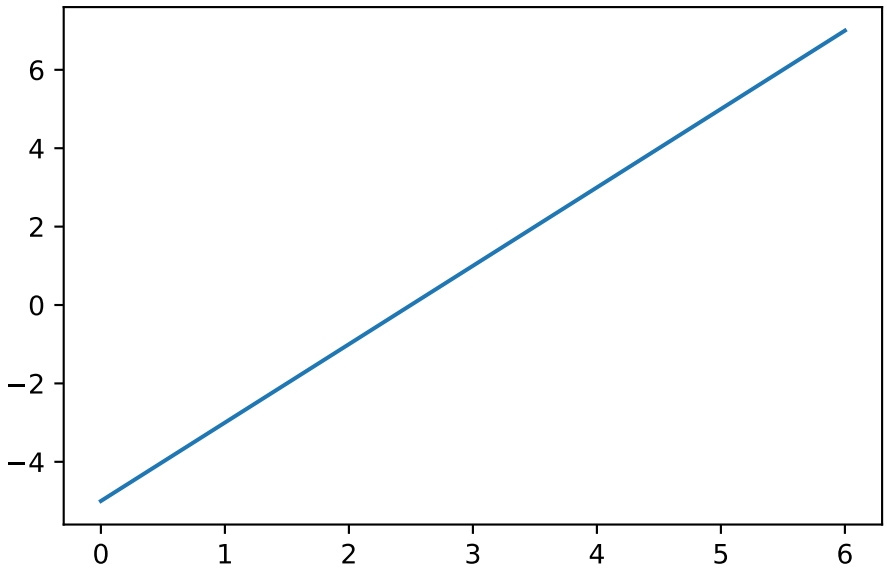
```
t = np.linspace(-3,3,20)
line_origin = np.array([t, 2*t]).T
plt.plot(line_origin[:,0], line_origin[:,1], label="Through origin")
```



2. Line not through the origin (affine subspace)

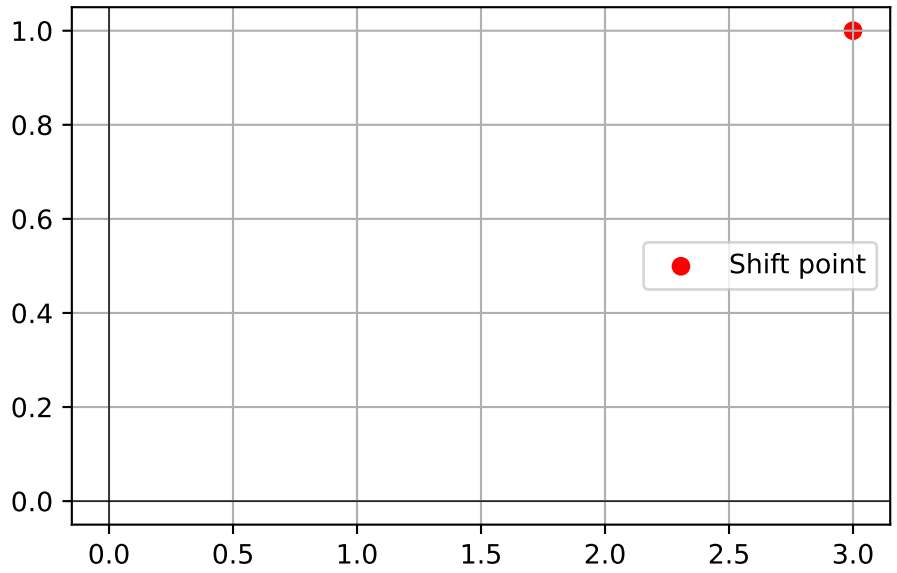
$$L' = \{[3, 1] + t \cdot [1, 2] : t \in \mathbb{R}\}$$

```
point = np.array([3,1])
direction = np.array([1,2])
line_shifted = np.array([point + k*direction for k in t])
plt.plot(line_shifted[:,0], line_shifted[:,1], label="Shifted line")
```



3. Visualizing together

```
plt.scatter(*point, color="red", label="Shift point")
plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.legend()
plt.grid()
plt.show()
```



One line passes through the origin, the other is parallel but shifted.

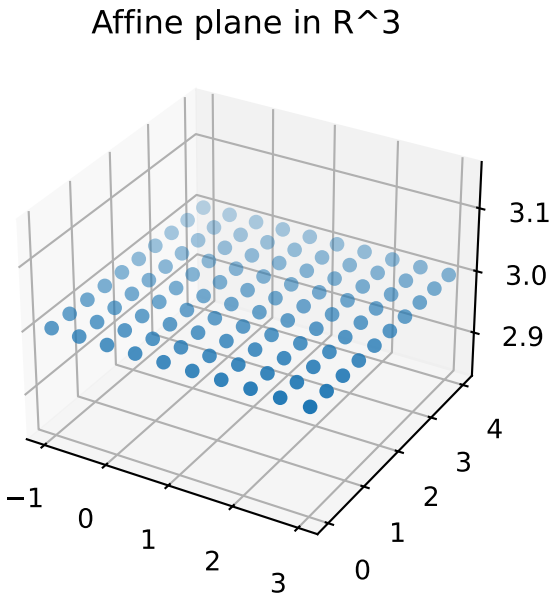
4. Plane example

A plane in $\mathbb{R}^3$:

$$P = \{[1,2,3] + s[1,0,0] + t[0,1,0] : s,t \in \mathbb{R}\}$$

This is an affine plane parallel to the xy-plane, but shifted.

```
s_vals = np.linspace(-2,2,10)
t_vals = np.linspace(-2,2,10)

points = []
for s in s_vals:
    for t in t_vals:
        points.append([1,2,3] + s*np.array([1,0,0]) + t*np.array([0,1,0]))

points = np.array(points)

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(points[:,0], points[:,1], points[:,2])
ax.set_title("Affine plane in R^3")
plt.show()
```



5. Algebraic difference

- A subspace must satisfy closure under addition and scalar multiplication, and must include 0.
- An affine subspace is just a subspace plus a fixed shift vector.

Try It Yourself

1. Define a line in $\mathbb{R}^2$:

$$(x, y) = (2, 3) + t(1, -1)$$

   Plot it and compare with the subspace spanned by $(1, -1)$.

2. Construct an affine plane in $\mathbb{R}^3$ shifted by vector $(5, 5, 5)$.
3. Show algebraically that subtracting the shift point turns an affine subspace back into a regular subspace.

The Takeaway

- Subspaces go through the origin.
- Affine subspaces are shifted copies of subspaces.
- They're essential in geometry, computer graphics, and optimization (e.g., feasible regions in linear programming).

Chapter 5. Linear Transformation and Structure

41. Linear Transformations (Preserving Lines and Sums)

A linear transformation is a function between vector spaces that preserves two key properties:

1. Additivity: $T(u + v) = T(u) + T(v)$
2. Homogeneity: $T(cu) = cT(u)$

In practice, every linear transformation can be represented by a matrix. This lab will help you understand and experiment with linear transformations in Python.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Simple linear transformation (scaling)

Let's scale vectors by 2 in the x-direction and by 0.5 in the y-direction.

```
A = np.array([
    [2, 0],
    [0, 0.5]
])

v = np.array([1, 2])
Tv = A @ v
print("Original v:", v)
print("Transformed Tv:", Tv)
```

```
Original v: [1 2]
Transformed Tv: [2. 1.]
```

2. Visualizing multiple vectors

```
vectors = [np.array([1,1]), np.array([2,0]), np.array([-1,2])]

for v in vectors:
    Tv = A @ v
    plt.arrow(0,0,v[0],v[1],head_width=0.1,color='blue',length_includes_head=True)
    plt.arrow(0,0,Tv[0],Tv[1],head_width=0.1,color='red',length_includes_head=True)

plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.xlim(-3,5)
plt.ylim(-1,5)
plt.grid()
plt.title("Blue = original, Red = transformed")
plt.show()
```

Blue arrows are the original vectors; red arrows are the transformed ones. Notice how the transformation stretches and compresses consistently.

3. Rotation as a linear transformation

Rotating vectors by $\theta = 90°$:

```
theta = np.pi/2
R = np.array([
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta),  np.cos(theta)]
])

v = np.array([1,0])
print("Rotate [1,0] by 90°:", R @ v)
```

```
Rotate [1,0] by 90°: [6.123234e-17 1.000000e+00]
```

The result is $[0, 1]$, a perfect rotation.

4. Checking linearity

```
u = np.array([1,2])
v = np.array([3,4])
c = 5

lhs = A @ (u+v)
rhs = A@u + A@v
print("Additivity holds?", np.allclose(lhs,rhs))

lhs = A @ (c*u)
rhs = c*(A@u)
print("Homogeneity holds?", np.allclose(lhs,rhs))
```

```
Additivity holds? True
Homogeneity holds? True
```

Both checks return `True`, proving T is linear.

5. Non-linear example (for contrast)

A transformation like $T(x, y) = (x^2, y)$ is not linear.

```
def nonlinear(v):
    return np.array([v[0]**2, v[1]])

print("T([2,3]) =", nonlinear(np.array([2,3])))
print("Check additivity:", nonlinear(np.array([1,2])+np.array([3,4])) == (nonlinear([1,2])+no
```

```
T([2,3]) = [4 3]
Check additivity: [False  True]
```

This fails the additivity test, so it's not linear.

Try It Yourself

1. Define a shear matrix

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

   Apply it to vectors and plot before/after.

2. Verify linearity for rotation by 45°.
3. Test whether $T(x, y) = (x + y, y)$ is linear.

The Takeaway

- A linear transformation preserves vector addition and scalar multiplication.
- Every linear transformation can be represented by a matrix.
- Visualizing with arrows helps build geometric intuition: stretching, rotating, and shearing are all linear.

42. Matrix Representation of a Linear Map (Choosing a Basis)

Every linear transformation can be written as a matrix, but the exact matrix depends on the basis you choose. This lab shows how to build and interpret matrix representations.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. From transformation to matrix

Suppose $T : \mathbb{R}^2 \to \mathbb{R}^2$ is defined by:

$$T(x, y) = (2x + y,\ x - y)$$

To find its matrix in the standard basis, apply T to each basis vector:

```
e1 = Matrix([1,0])
e2 = Matrix([0,1])

def T(v):
    x, y = v
    return Matrix([2*x + y, x - y])

print("T(e1):", T(e1))
print("T(e2):", T(e2))
```

```
T(e1): Matrix([[2], [1]])
T(e2): Matrix([[1], [-1]])
```

Stacking results as columns gives the matrix:

```
A = Matrix.hstack(T(e1), T(e2))
print("Matrix representation in standard basis:\n", A)
```

```
Matrix representation in standard basis:
 Matrix([[2, 1], [1, -1]])
```

2. Using the matrix for computations

```
v = Matrix([3,4])
print("T(v) via definition:", T(v))
print("T(v) via matrix:", A*v)
```

```
T(v) via definition: Matrix([[10], [-1]])
T(v) via matrix: Matrix([[10], [-1]])
```

Both methods match.

3. Matrix in a different basis

Now suppose we use basis

$$B = \{[1, 1], [1, -1]\}$$

To represent T in this basis:

1. Build the change-of-basis matrix P.
2. Compute $A_B = P^{-1}AP$.

```
B = Matrix.hstack(Matrix([1,1]), Matrix([1,-1]))
P = B
A_B = P.inv() * A * P
print("Matrix representation in new basis:\n", A_B)
```

```
Matrix representation in new basis:
 Matrix([[3/2, 3/2], [3/2, -1/2]])
```

4. Interpretation

- In standard basis, A tells us how T acts on unit vectors.
- In basis B, A_B shows how T looks when described using different coordinates.

5. Random linear map in $\mathbb{R}^3$

```
np.random.seed(1)
A3 = Matrix(np.random.randint(-3,4,(3,3)))
print("Random transformation matrix:\n", A3)

B3 = Matrix.hstack(Matrix([1,0,1]), Matrix([0,1,1]), Matrix([1,1,0]))
A3_B = B3.inv() * A3 * B3
print("Representation in new basis:\n", A3_B)
```

```
Random transformation matrix:
 Matrix([[2, 0, 1], [-3, -2, 0], [2, -3, -3]])
Representation in new basis:
 Matrix([[5/2, -3/2, 3], [-7/2, -9/2, -4], [1/2, 5/2, -1]])
```

Try It Yourself

1. Define $T(x, y) = (x + 2y, 3x + y)$. Find its matrix in the standard basis.
2. Use a new basis $\{[2, 0], [0, 3]\}$. Compute the representation A_B.
3. Verify that applying T directly to a vector matches computing via A_B and change-of-basis.

The Takeaway

- A linear transformation becomes a matrix representation once a basis is chosen.
- Columns of the matrix = images of basis vectors.
- Changing the basis changes the matrix, but the transformation itself stays the same.

43. Kernel and Image (Inputs That Vanish; Outputs We Can Reach)

Two fundamental subspaces describe any linear transformation $T(x) = Ax$:

- Kernel (null space): all vectors x such that $Ax = 0$.
- Image (column space): all possible outputs Ax.

The kernel tells us what inputs collapse to zero, while the image tells us what outputs are achievable.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Kernel of a matrix

Consider

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

```
A = Matrix([
    [1,2,3],
    [2,4,6]
])

print("Null space (kernel):", A.nullspace())
```

```
Null space (kernel): [Matrix([
[-2],
[ 1],
[ 0]]), Matrix([
[-3],
[ 0],
[ 1]])]
```

The null space basis shows dependencies among columns. Here, the kernel is 2-dimensional because columns are dependent.

2. Image (column space)

```
print("Column space (image):", A.columnspace())
print("Rank (dimension of image):", A.rank())
```

```
Column space (image): [Matrix([
[1],
[2]])]
Rank (dimension of image): 1
```

The image is spanned by $[1, 2]^T$. So all outputs of A are multiples of this vector.

3. Interpretation

- Kernel vectors $\rightarrow$ directions that map to zero.
- Image vectors $\rightarrow$ directions we can actually reach in the output space.

If $x \in \ker(A)$, then $Ax = 0$. If b is not in the image, the system $Ax = b$ has no solution.

4. Example with full rank

```
B = Matrix([
    [1,0,0],
    [0,1,0],
    [0,0,1]
])

print("Kernel of B:", B.nullspace())
print("Image of B:", B.columnspace())
```

```
Kernel of B: []
Image of B: [Matrix([
[1],
[0],
[0]]), Matrix([
[0],
[1],
[0]]), Matrix([
[0],
[0],
[1]])]
```

- Kernel = only zero vector.
- Image = all of $\mathbb{R}^3$.

5. NumPy version (image via column space)

```
A = np.array([[1,2,3],[2,4,6]], dtype=float)
rank = np.linalg.matrix_rank(A)
print("Rank with NumPy:", rank)
```

```
Rank with NumPy: 1
```

NumPy doesn't compute null spaces directly, but we can use SVD for that if needed.

Try It Yourself

1. Compute kernel and image for

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

   What do they look like?

2. Take a random 3×4 matrix and find its kernel and image dimensions.
3. Solve $Ax = b$ for a matrix A. Try two different b: one inside the image, one outside. Observe the difference.

The Takeaway

- Kernel = inputs that vanish under A.
- Image = outputs that can be reached by A.
- Together, they fully describe what a linear map does: what it "kills" and what it "produces."

44. Invertibility and Isomorphisms (Perfectly Reversible Maps)

A matrix (or linear map) is invertible if it has an inverse A^{-1} such that

$$A^{-1}A = I \quad \text{and} \quad AA^{-1} = I$$

An invertible map is also called an isomorphism, because it preserves all information - every input has exactly one output, and every output comes from exactly one input.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Checking invertibility

```
A = Matrix([
    [2,1],
    [5,3]
])

print("Determinant:", A.det())
print("Is invertible?", A.det() != 0)
```

```
Determinant: 1
Is invertible? True
```

If determinant $\neq 0 \rightarrow$ invertible.

2. Computing the inverse

```
A_inv = A.inv()
print("Inverse matrix:\n", A_inv)

print("Check A*A_inv = I:\n", A * A_inv)
```

```
Inverse matrix:
 Matrix([[3, -1], [-5, 2]])
Check A*A_inv = I:
 Matrix([[1, 0], [0, 1]])
```

3. Solving systems with inverses

For $Ax = b$, if A is invertible:

```
b = Matrix([1,2])
x = A_inv * b
print("Solution x:", x)
```

```
Solution x: Matrix([[1], [-1]])
```

This is equivalent to `A.solve(b)` in SymPy or `np.linalg.solve` in NumPy.

4. Non-invertible (singular) example

```
B = Matrix([
    [1,2],
    [2,4]
])

print("Determinant:", B.det())
print("Is invertible?", B.det() != 0)
```

```
Determinant: 0
Is invertible? False
```

Determinant = 0 → no inverse. The matrix collapses space onto a line, losing information.

5. NumPy version

```
A = np.array([[2,1],[5,3]], dtype=float)
print("Determinant:", np.linalg.det(A))
print("Inverse:\n", np.linalg.inv(A))
```

```
Determinant: 1.0000000000000009
Inverse:
 [[ 3. -1.]
 [-5.  2.]]
```

6. Geometric intuition

- Invertible transformation = reversible (like rotating, scaling by nonzero).
- Non-invertible transformation = squashing space into a lower dimension (like flattening a plane onto a line).

Try It Yourself

1. Test whether

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

   is invertible and find its inverse.

2. Compute the determinant of a 3×3 random integer matrix. If it's nonzero, find its inverse.
3. Create a singular 3×3 matrix (make one row a multiple of another). Confirm it has no inverse.

The Takeaway

- Invertible matrix isomorphism: perfectly reversible, no information lost.
- Determinant $0 \rightarrow$ invertible; determinant $= 0 \rightarrow$ singular.
- Inverses are useful conceptually, but in computation we usually solve systems directly instead of calculating A^{-1}.

45. Composition, Powers, and Iteration (Doing It Again and Again)

Linear transformations can be chained together. Applying one after another is called composition, and in matrix form this becomes multiplication. Repeated application of the same transformation leads to powers of a matrix.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Composition of transformations

Suppose we have two linear maps:

- T_1: rotate by 90°
- T_2: scale x by 2

```
theta = np.pi/2
R = np.array([
    [np.cos(theta), -np.sin(theta)],
    [np.sin(theta),  np.cos(theta)]
])
S = np.array([
    [2,0],
    [0,1]
])

# Compose: apply R then S
C = S @ R
print("Composite matrix:\n", C)
```

```
Composite matrix:
 [[ 1.2246468e-16 -2.0000000e+00]
 [ 1.0000000e+00  6.1232340e-17]]
```

Applying the composite matrix is equivalent to applying both maps in sequence.

2. Verifying with a vector

```
v = np.array([1,1])
step1 = R @ v
step2 = S @ step1
composite = C @ v

print("Step-by-step:", step2)
print("Composite:", composite)
```

```
Step-by-step: [-2.  1.]
Composite: [-2.  1.]
```

Both results are the same → composition = matrix multiplication.

3. Powers of a matrix

Repeatedly applying a transformation corresponds to matrix powers.

Example: scaling by 2.

```
A = np.array([[2,0],[0,2]])
v = np.array([1,1])

print("A @ v =", A @ v)
print("A^2 @ v =", np.linalg.matrix_power(A,2) @ v)
print("A^5 @ v =", np.linalg.matrix_power(A,5) @ v)
```

```
A @ v = [2 2]
A^2 @ v = [4 4]
A^5 @ v = [32 32]
```

Each step doubles the scaling effect.

4. Iteration dynamics

Let's iterate a transformation many times and see what happens.

Example:

$$A = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

```
A = np.array([[0.5,0],[0,0.5]])
v = np.array([4,4])

for i in range(5):
    v = A @ v
    print(f"Step {i+1}:", v)
```

```
Step 1: [2. 2.]
Step 2: [1. 1.]
Step 3: [0.5 0.5]
Step 4: [0.25 0.25]
Step 5: [0.125 0.125]
```

Each step shrinks the vector → iteration can reveal stability.

5. Random example

```
np.random.seed(0)
M = np.random.randint(-2,3,(2,2))
print("Random matrix:\n", M)

print("M^2:\n", np.linalg.matrix_power(M,2))
print("M^3:\n", np.linalg.matrix_power(M,3))
```

```
Random matrix:
 [[ 2 -2]
 [ 1  1]]
M^2:
 [[ 2 -6]
 [ 3 -1]]
M^3:
 [[ -2 -10]
 [  5  -7]]
```

Try It Yourself

1. Create two transformations: reflection across x-axis and scaling by 3. Compose them.
2. Take a shear matrix and compute A^5. What happens to a vector after repeated application?
3. Experiment with a rotation matrix raised to higher powers. What cycle do you see?

The Takeaway

- Composition of linear maps = matrix multiplication.
- Powers of a matrix represent repeated application.
- Iteration reveals long-term dynamics: shrinking, growing, or oscillating behavior.

46. Similarity and Conjugation (Same Action, Different Basis)

Two matrices A and B are called similar if there exists an invertible matrix P such that

$$B = P^{-1}AP$$

This means A and B represent the same linear transformation, but in different bases. This lab explores similarity and why it matters.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Example with a change of basis

```
A = Matrix([
    [2,1],
    [0,2]
])

P = Matrix([
    [1,1],
    [0,1]
])

B = P.inv() * A * P
print("Original A:\n", A)
print("Similar matrix B:\n", B)
```

```
Original A:
 Matrix([[2, 1], [0, 2]])
Similar matrix B:
 Matrix([[2, 1], [0, 2]])
```

Here, A and B are similar: they describe the same transformation in different coordinates.

2. Eigenvalues stay the same

Similarity preserves eigenvalues.

```
print("Eigenvalues of A:", A.eigenvals())
print("Eigenvalues of B:", B.eigenvals())
```

```
Eigenvalues of A: {2: 2}
Eigenvalues of B: {2: 2}
```

Both matrices have the same eigenvalues, even though their entries differ.

3. Similarity and diagonalization

If a matrix is diagonalizable, there exists P such that

$$D = P^{-1}AP$$

where D is diagonal.

```
C = Matrix([
    [4,1],
    [0,2]
])

P, D = C.diagonalize()
print("Diagonal form D:\n", D)
print("Check similarity (P^-1 C P = D):\n", P.inv()*C*P)
```

```
Diagonal form D:
 Matrix([[2, 0], [0, 4]])
Check similarity (P^-1 C P = D):
 Matrix([[2, 0], [0, 4]])
```

Diagonalization is a special case of similarity, where the new matrix is as simple as possible.

4. NumPy version

```
A = np.array([[2,1],[0,2]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A)
print("Eigenvalues:", eigvals)
print("Eigenvectors (basis P):\n", eigvecs)
```

```
Eigenvalues: [2. 2.]
Eigenvectors (basis P):
 [[ 1.0000000e+00 -1.0000000e+00]
 [ 0.0000000e+00  4.4408921e-16]]
```

Here, eigenvectors form the change-of-basis matrix P.

5. Geometric interpretation

- Similar matrices = same transformation, different "ruler" (basis).
- Diagonalization = finding a ruler that makes the transformation look like pure stretching along axes.

Try It Yourself

1. Take

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

   and find a matrix P that gives a similar B.

2. Show that two similar matrices have the same determinant and trace.
3. For a random 3×3 matrix, check if it is diagonalizable using SymPy's `.diagonalize()` method.

The Takeaway

- Similarity = same linear map, different basis.
- Similar matrices share eigenvalues, determinant, and trace.
- Diagonalization is the simplest similarity form, making repeated computations (like powers) much easier.

47. Projections and Reflections (Idempotent and Involutive Maps)

Two very common geometric linear maps are projections and reflections. They show up in graphics, physics, and optimization.

- A projection squashes vectors onto a subspace (like dropping a shadow).
- A reflection flips vectors across a line or plane (like a mirror).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Projection onto a line

If we want to project onto the line spanned by u, the projection matrix is:

$$P = \frac{uu^T}{u^Tu}$$

```
u = np.array([2,1], dtype=float)
u = u / np.linalg.norm(u)   # normalize
P = np.outer(u,u)

print("Projection matrix:\n", P)
```

```
Projection matrix:
 [[0.8 0.4]
 [0.4 0.2]]
```

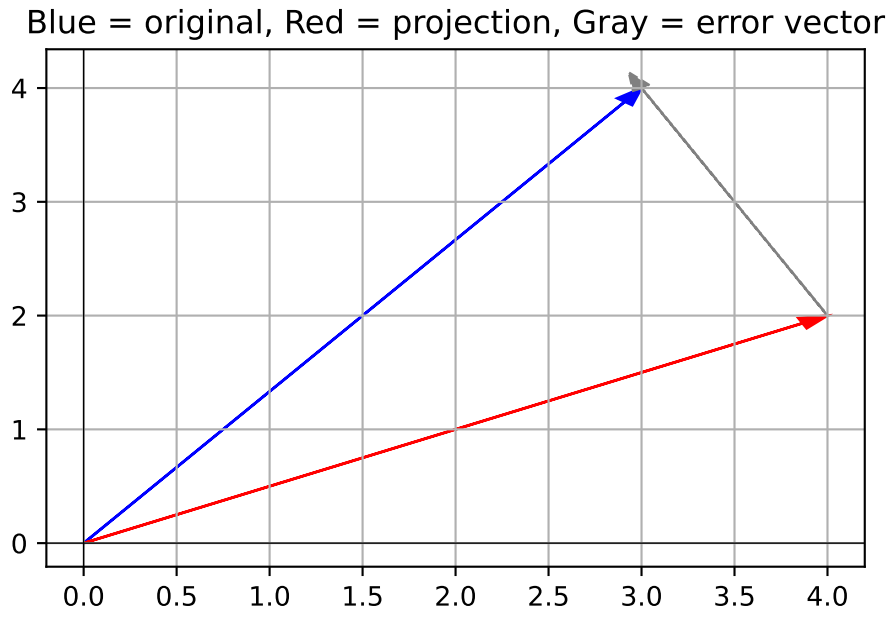
Apply projection:

```
v = np.array([3,4], dtype=float)
proj_v = P @ v
print("Original v:", v)
print("Projection of v onto u:", proj_v)
```

```
Original v: [3. 4.]
Projection of v onto u: [4. 2.]
```

2. Visualization of projection

```
plt.arrow(0,0,v[0],v[1],head_width=0.1,color="blue",length_includes_head=True)
plt.arrow(0,0,proj_v[0],proj_v[1],head_width=0.1,color="red",length_includes_head=True)
plt.arrow(proj_v[0],proj_v[1],v[0]-proj_v[0],v[1]-proj_v[1],head_width=0.1,color="gray",lines

plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.title("Blue = original, Red = projection, Gray = error vector")
plt.show()
```



The projection is the closest point on the line to the original vector.

3. Reflection across a line

The reflection matrix across the line spanned by u is:

$$R = 2P - I$$

```
I = np.eye(2)
R = 2*P - I

reflect_v = R @ v
print("Reflection of v across line u:", reflect_v)
```

```
Reflection of v across line u: [ 5.00000000e+00 -5.55111512e-16]
```

4. Checking algebraic properties

- Projection: $P^2 = P$ (idempotent).
- Reflection: $R^2 = I$ (involutive).

```
print("P^2 =\n", P @ P)
print("R^2 =\n", R @ R)
```

```
P^2 =
 [[0.8 0.4]
 [0.4 0.2]]
R^2 =
 [[ 1.00000000e+00 -1.59872116e-16]
 [-1.59872116e-16  1.00000000e+00]]
```

5. Projection in higher dimensions

Project onto the plane spanned by two vectors in $\mathbb{R}^3$.

```
u1 = np.array([1,0,0], dtype=float)
u2 = np.array([0,1,0], dtype=float)

U = np.column_stack((u1,u2))   # basis for plane
P_plane = U @ np.linalg.inv(U.T @ U) @ U.T

v = np.array([1,2,3], dtype=float)
proj_plane = P_plane @ v
print("Projection onto xy-plane:", proj_plane)
```

```
Projection onto xy-plane: [1. 2. 0.]
```

Try It Yourself

1. Project $[4, 5]$ onto the x-axis and verify the result.
2. Reflect $[1, 2]$ across the line $y = x$.
3. Create a random 3D vector and project it onto the plane spanned by $[1, 1, 0]$ and $[0, 1, 1]$.

The Takeaway

- Projection: idempotent ($P^2 = P$), finds the closest vector in a subspace.
- Reflection: involutive ($R^2 = I$), flips across a line/plane but preserves lengths.
- Both are simple but powerful examples of linear transformations with clear geometry.

48. Rotations and Shear (Geometric Intuition)

Two transformations often used in geometry, graphics, and physics are rotations and shears. Both are linear maps, but they behave differently:

- Rotation preserves lengths and angles.
- Shear preserves area (in 2D) but distorts shapes, turning squares into parallelograms.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Rotation in 2D

The rotation matrix by angle θ is:

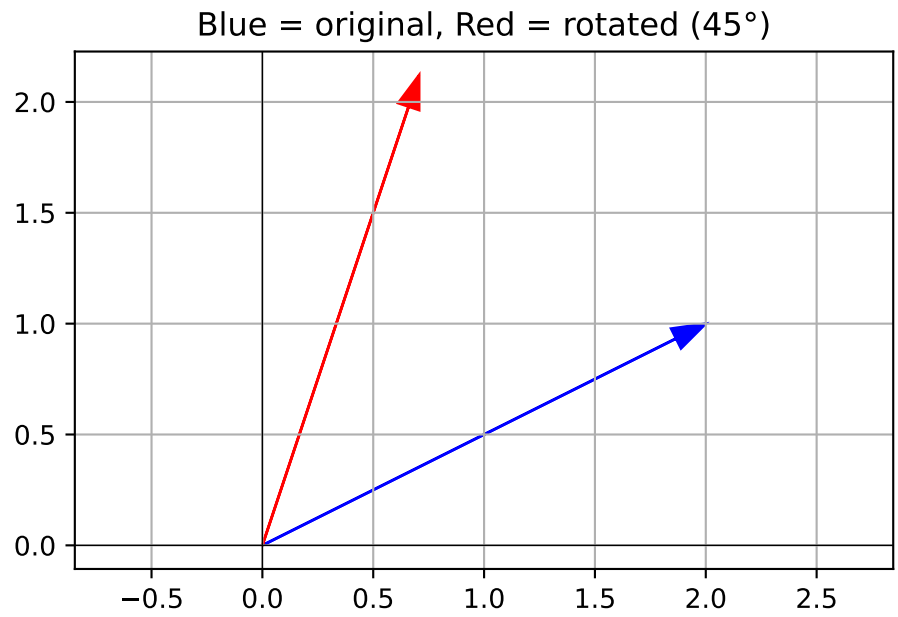
$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

```python
def rotation_matrix(theta):
    return np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta),  np.cos(theta)]
    ])

theta = np.pi/4   # 45 degrees
R = rotation_matrix(theta)

v = np.array([2,1])
rotated_v = R @ v
print("Original v:", v)
print("Rotated v (45°):", rotated_v)
```

```
Original v: [2 1]
Rotated v (45°): [0.70710678 2.12132034]
```

2. Visualizing rotation

```python
plt.arrow(0,0,v[0],v[1],head_width=0.1,color="blue",length_includes_head=True)
plt.arrow(0,0,rotated_v[0],rotated_v[1],head_width=0.1,color="red",length_includes_head=True)

plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.title("Blue = original, Red = rotated (45°)")
plt.axis("equal")
plt.show()
```



The vector rotates counterclockwise by 45°.

3. Shear in 2D

A shear along the x-axis by factor k:

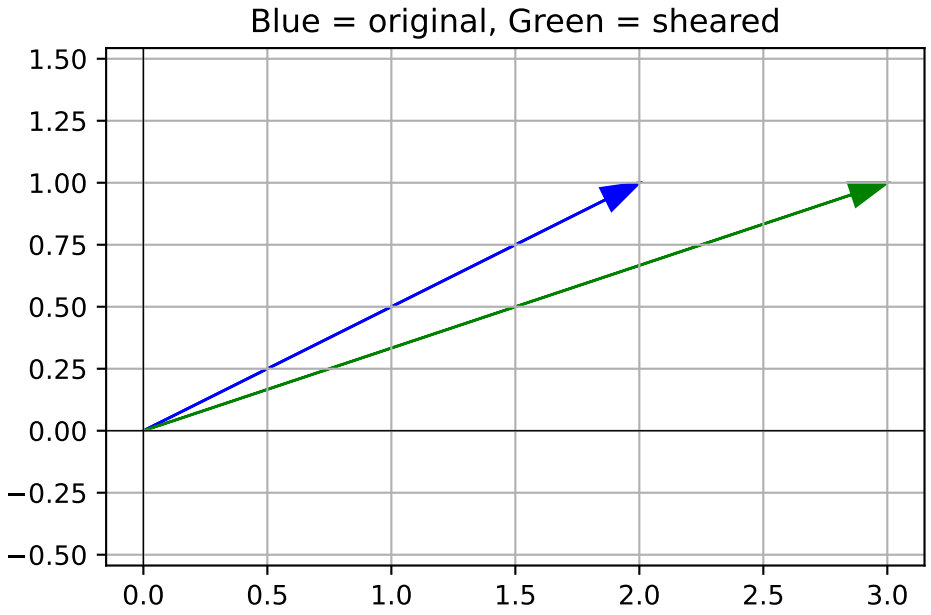
$$S = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$$

```
k = 1.0
S = np.array([
    [1,k],
    [0,1]
])

sheared_v = S @ v
print("Sheared v:", sheared_v)
```

```
Sheared v: [3. 1.]
```

4. Visualizing shear

```
plt.arrow(0,0,v[0],v[1],head_width=0.1,color="blue",length_includes_head=True)
plt.arrow(0,0,sheared_v[0],sheared_v[1],head_width=0.1,color="green",length_includes_head=Tru

plt.axhline(0,color='black',linewidth=0.5)
plt.axvline(0,color='black',linewidth=0.5)
plt.grid()
plt.title("Blue = original, Green = sheared")
plt.axis("equal")
plt.show()
```



The shear moves the vector sideways, distorting its angle.

5. Properties check

- Rotation preserves length:

```
print("||v|| =", np.linalg.norm(v))
print("||R v|| =", np.linalg.norm(rotated_v))
```

```
||v|| = 2.23606797749979
||R v|| = 2.2360679774997894
```

- Shear preserves area (determinant = 1):

```
print("det(S) =", np.linalg.det(S))
```

```
det(S) = 1.0
```

Try It Yourself

1. Rotate $[1, 0]$ by $90°$ and check it becomes $[0, 1]$.
2. Apply shear with $k = 2$ to a square (points $(0, 0), (1, 0), (1, 1), (0, 1)$) and plot before/after.
3. Combine rotation and shear: apply shear first, then rotation. What happens?

The Takeaway

- Rotation: length- and angle-preserving, determinant $= 1$.
- Shear: shape-distorting but area-preserving, determinant $= 1$.
- Both are linear maps that provide geometric intuition and real-world modeling tools.

49. Rank and Operator Viewpoint (Rank Beyond Elimination)

The rank of a matrix tells us how much "information" a linear map carries. Algebraically, it is the dimension of the image (column space). Geometrically, it measures how many independent directions survive the transformation.

From the operator viewpoint:

- A matrix A is not just a table of numbers - it is a linear operator that maps vectors to other vectors.
- The rank is the dimension of the output space that A actually reaches.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Rank via elimination (SymPy)

```
A = Matrix([
    [1,2,3],
    [2,4,6],
    [1,1,1]
])

print("Matrix A:\n", A)
print("Rank of A:", A.rank())
```

```
Matrix A:
 Matrix([[1, 2, 3], [2, 4, 6], [1, 1, 1]])
Rank of A: 2
```

Here, the second row is a multiple of the first → less independence → rank < 3.

2. Rank via NumPy

```
A_np = np.array([[1,2,3],[2,4,6],[1,1,1]], dtype=float)
print("Rank (NumPy):", np.linalg.matrix_rank(A_np))
```

```
Rank (NumPy): 2
```

3. Operator viewpoint

Let's apply A to random vectors:

```
for v in [np.array([1,0,0]), np.array([0,1,0]), np.array([0,0,1])]:
    print("A @", v, "=", A_np @ v)
```

```
A @ [1 0 0] = [1. 2. 1.]
A @ [0 1 0] = [2. 4. 1.]
A @ [0 0 1] = [3. 6. 1.]
```

Even though we started in 3D, all outputs lie in a plane in $\mathbb{R}^3$. That's why rank $= 2$.

4. Full rank vs reduced rank

- Full rank: the transformation preserves dimension (no collapse).
- Reduced rank: the transformation collapses onto a lower-dimensional subspace.

Example full-rank:

```
B = Matrix([
    [1,0,0],
    [0,1,0],
    [0,0,1]
])

print("Rank of B:", B.rank())
```

```
Rank of B: 3
```

5. Connection to nullity

The rank-nullity theorem:

$$\text{rank}(A) + \text{nullity}(A) = \text{number of columns of } A$$

Check with SymPy:

```
print("Null space (basis):", A.nullspace())
print("Nullity:", len(A.nullspace()))
print("Rank + Nullity =", A.rank() + len(A.nullspace()))
```

```
Null space (basis): [Matrix([
[ 1],
[-2],
[ 1]])]
Nullity: 1
Rank + Nullity = 3
```

Try It Yourself

1. Take

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

   and compute its rank. Why is it 1?

2. For a random 4×4 matrix, use `np.linalg.matrix_rank` to check if it's invertible.
3. Verify rank-nullity theorem for a 3×5 random integer matrix.

The Takeaway

- Rank = dimension of the image (how many independent outputs a transformation has).
- Operator viewpoint: rank shows how much of the input space survives after transformation.
- Rank-nullity links the image and kernel - together they fully describe a linear operator.

50. Block Matrices and Block Maps (Divide and Conquer Structure)

Sometimes matrices can be arranged in blocks (submatrices). Treating a big matrix as smaller pieces helps simplify calculations, especially in systems with structure (networks, coupled equations, or partitioned variables).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Constructing block matrices

We can build a block matrix from smaller pieces:

```
A11 = Matrix([[1,2],[3,4]])
A12 = Matrix([[5,6],[7,8]])
A21 = Matrix([[9,10]])
A22 = Matrix([[11,12]])

# Combine into a block matrix
A = Matrix.vstack(
    Matrix.hstack(A11, A12),
    Matrix.hstack(A21, A22)
)
print("Block matrix A:\n", A)
```

```
Block matrix A:
 Matrix([[1, 2, 5, 6], [3, 4, 7, 8], [9, 10, 11, 12]])
```

2. Block multiplication

If a matrix is partitioned into blocks, multiplication follows block rules:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Ax + By \\ Cx + Dy \end{bmatrix}$$

Example:

```
A = Matrix([
    [1,2,5,6],
    [3,4,7,8],
    [9,10,11,12]
])

x = Matrix([1,1,2,2])
print("A * x =", A*x)
```

```
A * x = Matrix([[25], [37], [65]])
```

Here the vector is split into blocks $[x, y]$.

3. Block diagonal matrices

Block diagonal = independent subproblems:

```
B1 = Matrix([[2,0],[0,2]])
B2 = Matrix([[3,1],[0,3]])

BlockDiag = Matrix([
    [2,0,0,0],
    [0,2,0,0],
    [0,0,3,1],
    [0,0,0,3]
])

print("Block diagonal matrix:\n", BlockDiag)
```

```
Block diagonal matrix:
 Matrix([[2, 0, 0, 0], [0, 2, 0, 0], [0, 0, 3, 1], [0, 0, 0, 3]])
```

Applying this matrix acts separately on each block - like running two smaller transformations in parallel.

4. Inverse of block diagonal

The inverse of a block diagonal is just the block diagonal of inverses:

```
B1_inv = B1.inv()
B2_inv = B2.inv()
BlockDiagInv = Matrix([
    [B1_inv[0,0],0,0,0],
    [0,B1_inv[1,1],0,0],
    [0,0,B2_inv[0,0],B2_inv[0,1]],
    [0,0,B2_inv[1,0],B2_inv[1,1]]
])
print("Inverse block diag:\n", BlockDiagInv)
```

```
Inverse block diag:
 Matrix([[1/2, 0, 0, 0], [0, 1/2, 0, 0], [0, 0, 1/3, -1/9], [0, 0, 0, 1/3]])
```

5. Practical example - coupled equations

Suppose we have two independent systems:

- System 1: $Ax = b$
- System 2: $Cy = d$

We can represent both together:

$$\begin{bmatrix} A & 0 \\ 0 & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ d \end{bmatrix}$$

This shows how block matrices organize multiple systems in one big equation.

Try It Yourself

1. Build a block diagonal matrix with three 2×2 blocks. Apply it to a vector.
2. Verify block multiplication rule by manually computing $Ax + By$ and $Cx + Dy$.
3. Write two small systems of equations and combine them into one block system.

The Takeaway

- Block matrices let us break down big systems into smaller parts.
- Block diagonal matrices = independent subsystems.
- Thinking in blocks simplifies algebra, programming, and numerical computation.

Chapter 6. Determinants and volume

51. Areas, Volumes, and Signed Scale Factors (Geometric Entry Point)

The determinant of a matrix has a deep geometric meaning: it tells us how a linear transformation scales area (in 2D), volume (in 3D), or higher-dimensional content. It can also flip orientation (sign).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Determinant in 2D (area scaling)

Let's take a matrix that stretches and shears:

```
A = Matrix([
    [2,1],
    [1,1]
])

print("Determinant:", A.det())
```

```
Determinant: 1
```

The determinant = 1 → areas are preserved, even though the shape is distorted.

2. Unit square under transformation

Transform the square with corners $(0,0), (1,0), (1,1), (0,1)$:

```
square = Matrix([
    [0,0],
    [1,0],
    [1,1],
    [0,1]
])

transformed = (A * square.T).T
print("Original square:\n", square)
print("Transformed square:\n", transformed)
```

```
Original square:
 Matrix([[0, 0], [1, 0], [1, 1], [0, 1]])
Transformed square:
 Matrix([[0, 0], [2, 1], [3, 2], [1, 1]])
```

The area of the transformed shape equals $|\det(A)|$.

3. Determinant in 3D (volume scaling)

```
B = Matrix([
    [1,2,0],
    [0,1,0],
    [0,0,3]
])

print("Determinant:", B.det())
```

```
Determinant: 3
```

$\det(B) = 3$ means that volumes are scaled by 3.

4. Negative determinant = orientation flip

```
C = Matrix([
    [0,1],
    [1,0]
])

print("Determinant:", C.det())
```

```
Determinant: -1
```

The determinant = -1 → area preserved but orientation flipped (like a mirror reflection).

5. NumPy version

```
A = np.array([[2,1],[1,1]], dtype=float)
print("Det (NumPy):", np.linalg.det(A))
```

```
Det (NumPy): 1.0
```

Try It Yourself

1. Take

$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$$

   and compute the determinant. Verify it scales areas by 6.

2. Build a 3×3 shear matrix and check how it affects volume.
3. Test a reflection matrix and confirm that the determinant is negative.

The Takeaway

- Determinant measures how a linear map scales area, volume, or hypervolume.
- Positive determinant = preserves orientation; negative = flips it.
- Magnitude of determinant = scaling factor of geometric content.

52. Determinant via Linear Rules (Multilinearity, Sign, Normalization)

The determinant isn't just a formula; it's defined by three elegant rules that make it unique. These rules capture its geometric meaning as a volume-scaling factor.

1. Multilinearity: Linear in each row (or column).
2. Sign Change: Swapping two rows flips the sign.
3. Normalization: The determinant of the identity matrix is 1.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Multilinearity

If one row is scaled, the determinant scales the same way.

```
A = Matrix([[1,2],[3,4]])
print("det(A):", A.det())

B = Matrix([[2,4],[3,4]])  # first row doubled
print("det(B):", B.det())
```

```
det(A): -2
det(B): -4
```

You'll see `det(B) = 2 * det(A)`.

2. Sign change by row swap

```
C = Matrix([[1,2],[3,4]])
C_swapped = Matrix([[3,4],[1,2]])

print("det(C):", C.det())
print("det(C_swapped):", C_swapped.det())
```

```
det(C): -2
det(C_swapped): 2
```

Swapping rows flips the sign of the determinant.

3. Normalization rule

```
I = Matrix.eye(3)
print("det(I):", I.det())
```

```
det(I): 1
```

The determinant of the identity is always 1 - this fixes the scaling baseline.

4. Combining rules (example in 3×3)

```
M = Matrix([[1,2,3],[4,5,6],[7,8,9]])
print("det(M):", M.det())
```

```
det(M): 0
```

Here, rows are linearly dependent, so the determinant is 0 - consistent with multilinearity (since one row can be written as a combo of others).

5. NumPy check

```
A = np.array([[1,2],[3,4]], dtype=float)
print("det(A) NumPy:", np.linalg.det(A))
```

```
det(A) NumPy: -2.0000000000000004
```

Both SymPy and NumPy confirm the same result.

Try It Yourself

1. Scale a row of a 3×3 matrix by 3. Confirm the determinant scales by 3.
2. Swap two rows twice in a row - does the determinant return to its original value?
3. Compute determinant of a triangular matrix. What pattern do you see?

The Takeaway

- Determinant is defined by multilinearity, sign change, and normalization.
- These rules uniquely pin down the determinant's behavior.
- Every formula (cofactor expansion, row-reduction method, etc.) comes from these core principles.

53. Determinant and Row Operations (How Each Move Changes det)

Row operations are at the heart of Gaussian elimination, and the determinant has simple, predictable reactions to them. Understanding these reactions gives both computational shortcuts and geometric intuition.

The Three Key Rules

1. Row swap: Swapping two rows flips the sign of the determinant.
2. Row scaling: Multiplying a row by a scalar c multiplies the determinant by c.
3. Row replacement: Adding a multiple of one row to another leaves the determinant unchanged.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Row swap

```
A = Matrix([[1,2],[3,4]])
print("det(A):", A.det())

A_swapped = Matrix([[3,4],[1,2]])
print("det(after swap):", A_swapped.det())
```

```
det(A): -2
det(after swap): 2
```

The result flips sign.

2. Row scaling

```
B = Matrix([[1,2],[3,4]])
B_scaled = Matrix([[2,4],[3,4]])  # first row × 2

print("det(B):", B.det())
print("det(after scaling row 1 by 2):", B_scaled.det())
```

```
det(B): -2
det(after scaling row 1 by 2): -4
```

Determinant is multiplied by 2.

3. Row replacement (no change)

```
C = Matrix([[1,2],[3,4]])
C_replaced = Matrix([[1,2],[3-2*1, 4-2*2]])  # row2 → row2 - 2*row1

print("det(C):", C.det())
print("det(after row replacement):", C_replaced.det())
```

```
det(C): -2
det(after row replacement): -2
```

Determinant stays the same.

4. Triangular form shortcut

Since elimination only uses row replacement (which doesn't change the determinant) and row swaps/scales (which we can track), the determinant of a triangular matrix is just the product of its diagonal entries.

```
D = Matrix([[2,1,3],[0,4,5],[0,0,6]])
print("det(D):", D.det())
print("Product of diagonals:", 2*4*6)
```

```
det(D): 48
Product of diagonals: 48
```

5. NumPy confirmation

```
A = np.array([[1,2,3],[0,4,5],[1,0,6]], dtype=float)
print("det(A):", np.linalg.det(A))
```

```
det(A): 22.000000000000004
```

Try It Yourself

1. Take

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

   and scale the second row by $\frac{1}{2}$. Compare determinants before and after.

2. Do Gaussian elimination on a 3×3 matrix, and track how each row operation changes the determinant.

3. Compute determinant by reducing to triangular form and compare with SymPy's `.det()`.

The Takeaway

- Determinant reacts predictably to row operations.
- Row replacement is "safe" (no change), scaling multiplies by the factor, and swapping flips the sign.
- This makes elimination not just a solving tool, but also a method to compute determinants efficiently.

54. Triangular Matrices and Product of Diagonals (Fast Wins)

For triangular matrices (upper or lower), the determinant is simply the product of the diagonal entries. This rule is one of the biggest shortcuts in linear algebra - no expansion or elimination needed.

Why It Works

- Triangular matrices already look like the end result of Gaussian elimination.
- Since row replacement operations don't change the determinant, what's left is just the product of the diagonal.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Upper triangular example

```
A = Matrix([
    [2,1,3],
    [0,4,5],
    [0,0,6]
])

print("det(A):", A.det())
print("Product of diagonals:", 2*4*6)
```

```
det(A): 48
Product of diagonals: 48
```

Both values match exactly.

2. Lower triangular example

```
B = Matrix([
    [7,0,0],
    [2,5,0],
    [3,4,9]
])

print("det(B):", B.det())
print("Product of diagonals:", 7*5*9)
```

```
det(B): 315
Product of diagonals: 315
```

3. Diagonal matrix (special case)

For diagonal matrices, determinant = product of diagonal entries directly.

```
C = Matrix.diag(3,5,7)
print("det(C):", C.det())
print("Product of diagonals:", 3*5*7)
```

```
det(C): 105
Product of diagonals: 105
```

4. NumPy version

```
A = np.array([[2,1,3],[0,4,5],[0,0,6]], dtype=float)
print("det(A):", np.linalg.det(A))
print("Product of diagonals:", np.prod(np.diag(A)))
```

```
det(A): 47.999999999999986
Product of diagonals: 48.0
```

5. Quick elimination to triangular form

Even for non-triangular matrices, elimination reduces them to triangular form, where this rule applies.

```
D = Matrix([[1,2,3],[4,5,6],[7,8,10]])
print("det(D) via SymPy:", D.det())
print("det(D) via LU decomposition:", D.LUdecomposition()[0].det() * D.LUdecomposition()[1].
```

```
det(D) via SymPy: -3
det(D) via LU decomposition: -3
```

Try It Yourself

1. Compute the determinant of a 4×4 diagonal matrix quickly.
2. Verify that triangular matrices with a zero on the diagonal always have determinant 0.
3. Use SymPy to check that elimination to triangular form preserves determinant (except for swaps/scales).

The Takeaway

- For triangular (and diagonal) matrices:

$$\det(A) = \prod_i a_{ii}$$

- This shortcut makes determinant computation trivial.
- Gaussian elimination leverages this fact: once reduced to triangular form, the determinant is just the product of pivots (with sign adjustments for swaps).

55. det(AB) = det(A)det(B) (Multiplicative Magic)

One of the most elegant properties of determinants is multiplicativity:

$$\det(AB) = \det(A)\ \det(B)$$

This rule is powerful because it connects algebra (matrix multiplication) with geometry (volume scaling).

Geometric Intuition

- If A scales volumes by factor $\det(A)$, and B scales them by $\det(B)$, then applying B followed by A scales volumes by $\det(A)\det(B)$.
- This property works in all dimensions.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. 2×2 example

```
A = Matrix([[2,1],[0,3]])
B = Matrix([[1,4],[2,5]])

detA = A.det()
detB = B.det()
detAB = (A*B).det()

print("det(A):", detA)
print("det(B):", detB)
print("det(AB):", detAB)
print("det(A)*det(B):", detA*detB)
```

```
det(A): 6
det(B): -3
det(AB): -18
det(A)*det(B): -18
```

The two results match.

2. 3×3 random matrix check

```
np.random.seed(1)
A = Matrix(np.random.randint(-3,4,(3,3)))
B = Matrix(np.random.randint(-3,4,(3,3)))

print("det(A):", A.det())
print("det(B):", B.det())
print("det(AB):", (A*B).det())
print("det(A)*det(B):", A.det()*B.det())
```

```
det(A): 25
det(B): -15
det(AB): -375
det(A)*det(B): -375
```

3. Special cases

- If $\det(A) = 0$, then $\det(AB) = 0$.
- If $\det(A) = \pm 1$, it acts like a "volume-preserving" transformation (rotation/reflection).

```
A = Matrix([[1,0],[0,0]])  # singular
B = Matrix([[2,3],[4,5]])

print("det(A):", A.det())
print("det(AB):", (A*B).det())
```

```
det(A): 0
det(AB): 0
```

Both are 0.

4. NumPy version

```
A = np.array([[2,1],[0,3]], dtype=float)
B = np.array([[1,4],[2,5]], dtype=float)

lhs = np.linalg.det(A @ B)
rhs = np.linalg.det(A) * np.linalg.det(B)

print("det(AB) =", lhs)
print("det(A)*det(B) =", rhs)
```

```
det(AB) = -17.999999999999996
det(A)*det(B) = -17.999999999999996
```

Try It Yourself

1. Construct two triangular matrices and verify multiplicativity (diagonal products multiply too).
2. Test the property with an orthogonal matrix Q ($\det(Q) = \pm 1$). What happens?
3. Try with one matrix singular - confirm the product is always singular.

The Takeaway

- Determinant is multiplicative, not additive.
- $\det(AB) = \det(A)\det(B)$ is a cornerstone identity in linear algebra.
- This property connects geometry (volume scaling) with algebra (matrix multiplication).

56. Invertibility and Zero Determinant (Flat vs. Full Volume)

The determinant gives a quick test for invertibility:

- If $\det(A) \neq 0$, the matrix is invertible.
- If $\det(A) = 0$, the matrix is singular (non-invertible).

Geometrically:

- Nonzero determinant → transformation keeps full dimension (no collapse).
- Zero determinant → transformation flattens space into a lower dimension (volume = 0).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
from sympy.matrices.common import NonInvertibleMatrixError
```

Step-by-Step Code Walkthrough

1. Invertible example

```
A = Matrix([[2,1],[5,3]])
print("det(A):", A.det())
print("Inverse exists?", A.det() != 0)
print("A inverse:\n", A.inv())
```

```
det(A): 1
Inverse exists? True
A inverse:
 Matrix([[3, -1], [-5, 2]])
```

The determinant is nonzero → invertible.

2. Singular example (zero determinant)

```
B = Matrix([[1,2],[2,4]])
print("det(B):", B.det())
print("Inverse exists?", B.det() != 0)
```

```
det(B): 0
Inverse exists? False
```

Since the second row is a multiple of the first, determinant $= 0 \rightarrow$ no inverse.

3. Solving systems with determinant check

If $\det(A) = 0$, the system $Ax = b$ may have no solutions or infinitely many.

```
# 3. Solving systems with determinant check
b = Matrix([1,2])
try:
    print("Solve Ax=b with singular B:", B.solve(b))
except NonInvertibleMatrixError as e:
    print("Error when solving Ax=b:", e)
```

```
Error when solving Ax=b: Matrix det == 0; not invertible.
```

SymPy indicates inconsistency or multiple solutions.

4. Higher-dimensional example

```
C = Matrix([
    [1,0,0],
    [0,2,0],
    [0,0,3]
])
print("det(C):", C.det())
print("Invertible?", C.det() != 0)
```

```
det(C): 6
Invertible? True
```

Diagonal entries all nonzero $\rightarrow$ invertible.

5. NumPy version

```python
A = np.array([[2,1],[5,3]], dtype=float)
print("det(A):", np.linalg.det(A))
print("Inverse:\n", np.linalg.inv(A))

B = np.array([[1,2],[2,4]], dtype=float)
print("det(B):", np.linalg.det(B))
# np.linalg.inv(B) would fail because det=0
```

```
det(A): 1.0000000000000009
Inverse:
 [[ 3. -1.]
 [-5.  2.]]
det(B): 0.0
```

Try It Yourself

1. Build a 3×3 matrix with determinant 0 by making one row a multiple of another. Confirm singularity.
2. Generate a random 4×4 matrix and check whether it's invertible using `.det()`.
3. Test if two different 2×2 matrices are invertible, then multiply them together - is the product invertible too?

The Takeaway

- $\det(A) \neq 0 \implies$ invertible (full volume).
- $\det(A) = 0 \implies$ singular (space collapsed).
- Determinant gives both algebraic and geometric insight into when a matrix is reversible.

57. Cofactor Expansion (Laplace's Method)

The cofactor expansion is a systematic way to compute determinants using minors. It's not efficient for large matrices, but it reveals the recursive structure of determinants.

Definition

For an $n \times n$ matrix A,

$$\det(A) = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det(M_{ij})$$

where:

- i = chosen row (or column),
- M_{ij} = minor matrix after removing row i, column j.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix, symbols
```

Step-by-Step Code Walkthrough

1. 2×2 case (base rule)

```
# declare symbols
a, b, c, d = symbols('a b c d')

# build the matrix
A = Matrix([[a, b],[c, d]])

# compute determinant
detA = A.det()
print("Determinant 2x2:", detA)
```

```
Determinant 2x2: a*d - b*c
```

Formula: $\det(A) = ad - bc$.

2. 3×3 example using cofactor expansion

```
A = Matrix([
    [1,2,3],
    [4,5,6],
    [7,8,9]
])

detA = A.det()
print("Determinant via SymPy:", detA)
```

```
Determinant via SymPy: 0
```

Let's compute manually along the first row:

```
cofactor_expansion = (
    1 * Matrix([[5,6],[8,9]]).det()
    - 2 * Matrix([[4,6],[7,9]]).det()
    + 3 * Matrix([[4,5],[7,8]]).det()
)
print("Cofactor expansion result:", cofactor_expansion)
```

```
Cofactor expansion result: 0
```

Both match (here = 0 because rows are dependent).

3. Expansion along different rows/columns

The result is the same no matter which row/column you expand along.

```
cofactor_col1 = (
    1 * Matrix([[2,3],[8,9]]).det()
    - 4 * Matrix([[2,3],[5,6]]).det()
    + 7 * Matrix([[2,3],[5,6]]).det()
)
print("Expansion along col1:", cofactor_col1)
```

```
Expansion along col1: -15
```

4. Larger example (4×4)

```
B = Matrix([
    [2,0,1,3],
    [1,2,0,4],
    [0,1,1,0],
    [3,0,2,1]
])

print("Determinant 4x4:", B.det())
```

```
Determinant 4x4: -15
```

SymPy handles it directly, but conceptually it's still the same recursive expansion.

5. NumPy vs SymPy

```
B_np = np.array([[2,0,1,3],[1,2,0,4],[0,1,1,0],[3,0,2,1]], dtype=float)
print("NumPy determinant:", np.linalg.det(B_np))
```

```
NumPy determinant: -15.0
```

Try It Yourself

1. Compute a 3×3 determinant manually using cofactor expansion and confirm with `.det()`.
2. Expand along a different row and check that the result is unchanged.
3. Build a 4×4 diagonal matrix and expand it - what simplification do you notice?

The Takeaway

- Cofactor expansion defines determinant recursively.
- Works on any row or column, with consistent results.
- Important for proofs and theory, though not practical for computation on large matrices.

58. Permutations and Sign (The Combinatorial Core)

The determinant can also be defined using permutations of indices. This looks abstract, but it's the most fundamental definition:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)}$$

- S_n = set of all permutations of $\{1, \ldots, n\}$
- $\operatorname{sgn}(\sigma) = +1$ if the permutation is even, -1 if odd
- Each term = one product of entries, one from each row and column

This formula explains why determinants mix signs, why row swaps flip the determinant, and why dependence kills it.

Set Up Your Lab

```
import itertools
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Determinant by permutation expansion (3×3)

```
def determinant_permutation(A):
    n = A.shape[0]
    total = 0
    for perm in itertools.permutations(range(n)):
        sign = (-1)**(sum(1 for i in range(n) for j in range(i) if perm[j] > perm[i]))
        product = 1
        for i in range(n):
            product *= A[i, perm[i]]
        total += sign * product
    return total

A = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])

print("Permutation formula det:", determinant_permutation(A))
print("NumPy det:", np.linalg.det(A))
```

```
Permutation formula det: 0
NumPy det: 0.0
```

Both results 0, since rows are dependent.

2. Count permutations

For $n = 3$, there are $3! = 6$ terms:

$$\det(A) = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

You can see the alternating signs explicitly.

3. Verification with SymPy

```
M = Matrix([[2,1,0],
            [1,3,4],
            [0,2,5]])
print("SymPy det:", M.det())
```

```
SymPy det: 9
```

Matches the permutation expansion.

4. Growth of terms

- $2\times2 \to 2$ terms
- $3\times3 \to 6$ terms
- $4\times4 \to 24$ terms
- $n \to n!$ terms (factorial growth!)

This is why cofactor or LU is preferred computationally.

Try It Yourself

1. Write out the 2×2 permutation formula explicitly and check it equals $ad - bc$.
2. Expand a 3×3 determinant by hand using the six terms.
3. Modify the code to count how many multiplications are required for a 5×5 matrix using the permutation definition.

The Takeaway

- Determinant = signed sum over all permutations.
- Signs come from permutation parity (even/odd swaps).
- This definition is the combinatorial foundation that unifies all determinant properties.

59. Cramer's Rule (Solving with Determinants, and When Not to Use It)

Cramer's Rule gives an explicit formula for solving a system of linear equations $Ax = b$ using determinants. It is elegant but inefficient for large systems.

For $A \in \mathbb{R}^{n\times n}$ with $\det(A) \neq 0$:

$$x_i = \frac{\det(A_i)}{\det(A)}$$

where A_i is A with its i-th column replaced by b.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Simple 2×2 example

Solve:

$$\begin{cases} 2x + y = 5 \\ x - y = 1 \end{cases}$$

```
A = Matrix([[2,1],[1,-1]])
b = Matrix([5,1])

detA = A.det()
print("det(A):", detA)

# Replace columns
A1 = A.copy()
A1[:,0] = b
A2 = A.copy()
A2[:,1] = b

x1 = A1.det() / detA
x2 = A2.det() / detA
print("Solution via Cramer's Rule:", [x1, x2])

# Check with built-in solver
print("SymPy solve:", A.LUsolve(b))
```

```
det(A): -3
Solution via Cramer's Rule: [2, 1]
SymPy solve: Matrix([[2], [1]])
```

Both give the same solution.

2. 3×3 example

```
A = Matrix([
    [1,2,3],
    [0,1,4],
    [5,6,0]
])
b = Matrix([7,8,9])

detA = A.det()
print("det(A):", detA)

solutions = []
for i in range(A.shape[1]):
    Ai = A.copy()
    Ai[:,i] = b
    solutions.append(Ai.det()/detA)

print("Solution via Cramer's Rule:", solutions)
print("SymPy solve:", A.LUsolve(b))
```

```
det(A): 1
Solution via Cramer's Rule: [21, -16, 6]
SymPy solve: Matrix([[21], [-16], [6]])
```

3. NumPy version (inefficient but illustrative)

```
A = np.array([[2,1],[1,-1]], dtype=float)
b = np.array([5,1], dtype=float)

detA = np.linalg.det(A)

solutions = []
for i in range(A.shape[1]):
    Ai = A.copy()
    Ai[:,i] = b
    solutions.append(np.linalg.det(Ai)/detA)

print("Solution:", solutions)
```

```
Solution: [np.float64(2.0000000000000004), np.float64(1.0)]
```

4. Why not use it in practice?

- Requires computing $n + 1$ determinants.
- Determinant computation via cofactor expansion is factorial-time.
- Gaussian elimination or LU is far more efficient.

Try It Yourself

1. Solve a 3×3 system using Cramer's Rule and confirm with `A.solve(b)`.
2. Try Cramer's Rule when $\det(A) = 0$. What happens?
3. Compare runtime of Cramer's Rule vs LU for a random 5×5 matrix.

The Takeaway

- Cramer's Rule gives explicit formulas for solutions using determinants.
- Beautiful for theory, useful for small cases, but not computationally practical.
- It highlights the deep connection between determinants and solving linear systems.

60. Computing Determinants in Practice (Use LU, Mind Stability)

While definitions like cofactor expansion and permutations are beautiful, they are too slow for large matrices. In practice, determinants are computed using row reduction or LU decomposition, with careful attention to numerical stability.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Cofactor expansion is too slow

```
A = Matrix([
    [1,2,3],
    [4,5,6],
    [7,8,10]
])
print("det via cofactor expansion:", A.det())
```

```
det via cofactor expansion: -3
```

This works for 3×3, but complexity grows factorially.

2. Determinant via triangular form (LU decomposition)

LU decomposition factorizes $A = LU$, where L is lower triangular and U is upper triangular. Determinant = product of diagonals of U, up to sign corrections for row swaps.

```
L, U, perm = A.LUdecomposition()
detA = A.det()
print("L:\n", L)
print("U:\n", U)
print("Permutation matrix:\n", perm)
print("det via LU product:", detA)
```

```
L:
 Matrix([[1, 0, 0], [4, 1, 0], [7, 2, 1]])
U:
 Matrix([[1, 2, 3], [0, -3, -6], [0, 0, 1]])
Permutation matrix:
 []
det via LU product: -3
```

3. NumPy efficient method

```
A_np = np.array([[1,2,3],[4,5,6],[7,8,10]], dtype=float)
print("NumPy det:", np.linalg.det(A_np))
```

```
NumPy det: -2.9999999999999996
```

NumPy uses optimized routines (LAPACK under the hood).

4. Large random matrix

```
np.random.seed(0)
B = np.random.rand(5,5)
print("NumPy det (5x5):", np.linalg.det(B))
```

```
NumPy det (5x5): 0.00965822550588513
```

Computes quickly even for larger matrices.

5. Stability issues

Determinants of large or ill-conditioned matrices can suffer from floating-point errors. For example, if rows are nearly dependent:

```
C = np.array([[1,2,3],[2,4.0000001,6],[3,6,9]], dtype=float)
print("det(C):", np.linalg.det(C))
```

```
det(C): 0.0
```

The result may not be exactly 0 due to floating-point approximations.

Try It Yourself

1. Compute the determinant of a random 10×10 matrix with `np.linalg.det`.
2. Compare results between SymPy (exact rational arithmetic) and NumPy (floating-point).
3. Test determinant of a nearly singular matrix - notice numerical instability.

The Takeaway

- Determinants in practice are computed with LU decomposition or equivalent.
- Always be mindful of numerical stability - small errors matter when determinant 0.
- For exact answers (small cases), use symbolic tools like SymPy; for speed, use NumPy.

Chapter 7. Eigenvalues, Eigenvectors, and Dynamics

61. Eigenvalues and Eigenvectors (Directions That Stay Put)

An eigenvector of a matrix A is a special vector that doesn't change direction when multiplied by A. Instead, it only gets stretched or shrunk by a scalar called the eigenvalue.

Formally:

$$Av = \lambda v$$

where v is an eigenvector and λ is the eigenvalue.

Geometrically: eigenvectors are "preferred directions" of a linear transformation.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. A simple 2×2 example

```
A = Matrix([
    [2,1],
    [1,2]
])

eigs = A.eigenvects()
print("Eigenvalues and eigenvectors:", eigs)
```

```
Eigenvalues and eigenvectors: [(1, 1, [Matrix([
[-1],
[ 1]])]), (3, 1, [Matrix([
[1],
[1]])])]
```

This outputs eigenvalues and their associated eigenvectors.

2. Verify the eigen equation

Pick one eigenpair (λ, v):

```
lam = eigs[0][0]
v = eigs[0][2][0]
print("Check Av =  v:", A*v, lam*v)
```

```
Check Av =  v: Matrix([[-1], [1]]) Matrix([[-1], [1]])
```

Both sides match → confirming the eigenpair.

3. NumPy version

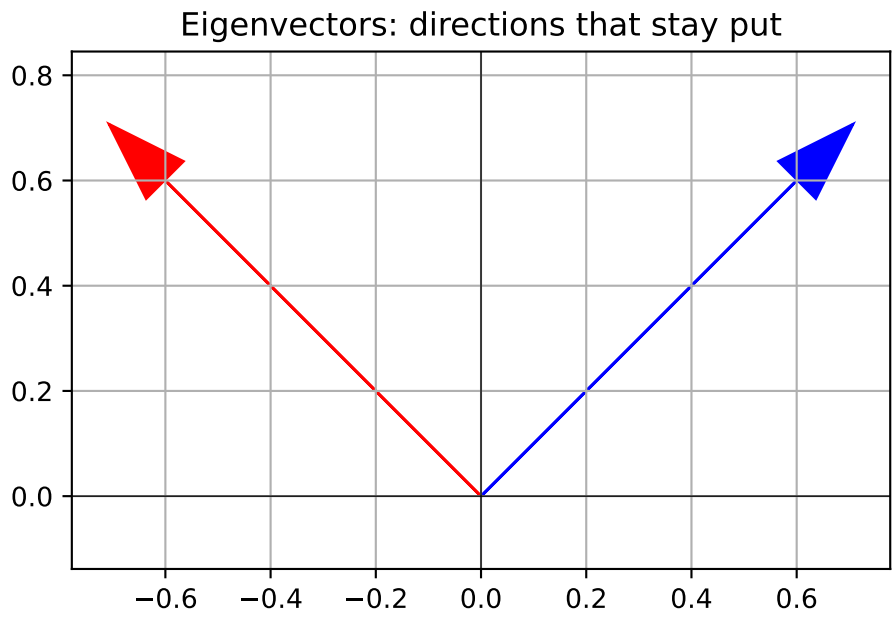
```
A_np = np.array([[2,1],[1,2]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A_np)

print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

```
Eigenvalues: [3. 1.]
Eigenvectors:
 [[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]]
```

Columns of `eigvecs` are eigenvectors.

4. Geometric interpretation (plot)

```
import matplotlib.pyplot as plt

v1 = np.array(eigvecs[:,0])
v2 = np.array(eigvecs[:,1])

plt.arrow(0,0,v1[0],v1[1],head_width=0.1,color="blue",length_includes_head=True)
plt.arrow(0,0,v2[0],v2[1],head_width=0.1,color="red",length_includes_head=True)

plt.axhline(0,color="black",linewidth=0.5)
plt.axvline(0,color="black",linewidth=0.5)
plt.axis("equal")
plt.grid()
plt.title("Eigenvectors: directions that stay put")
plt.show()
```



Both eigenvectors define directions where the transformation acts by scaling only.

5. Random 3×3 matrix example

```
np.random.seed(1)
B = Matrix(np.random.randint(-2,3,(3,3)))
print("Matrix B:\n", B)
print("Eigenvalues/vectors:", B.eigenvects())
```

```
Matrix B:
 Matrix([[1, 2, -2], [-1, 1, -2], [-2, -1, 2]])
Eigenvalues/vectors: [(4/3 + (-1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3) + 13/(9*(-
1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)), 1, [Matrix([
[ -16/27 - 91/(81*(-1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)) + (4/3 + (-
1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3) + 13/(9*(-1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3
1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)/9],
[50/27 + 5*(-1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)/9 - 2*(4/3 + (-
1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3) + 13/(9*(-1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3
1/2 - sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3))],
[
1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)) + (-1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127
[ -16/27 - 7*(-1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)/9 + (4/3 + 13/(9*(-
1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)) + (-1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127
1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3))],
```

```
[50/27 + 65/(81*(-1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)) - 2*(4/3 + 13/(9*(-
1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)) + (-1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 12
1/2 + sqrt(3)*I/2)*(2*sqrt(43)/3 + 127/27)**(1/3)/9],
[
[  -7*(2*sqrt(43)/3 + 127/27)**(1/3)/9 - 16/27 - 91/(81*(2*sqrt(43)/3 + 127/27)**(1/3)) + (1
[-2*(13/(9*(2*sqrt(43)/3 + 127/27)**(1/3)) + 4/3 + (2*sqrt(43)/3 + 127/27)**(1/3))**2/9 + 65/
[
```

Try It Yourself

1. Compute eigenvalues and eigenvectors of

$$\begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$$

   and verify that they match the diagonal entries.

2. Use NumPy to find eigenvectors of a rotation matrix by 90°. What do you notice?

3. For a singular matrix, check if 0 is an eigenvalue.

The Takeaway

- Eigenvalues = scale factors; eigenvectors = directions that stay put.
- The eigen equation $Av = \lambda v$ captures the essence of a matrix's action.
- They form the foundation for deeper topics like diagonalization, stability, and dynamics.

62. Characteristic Polynomial (Where Eigenvalues Come From)

Eigenvalues don't appear out of thin air - they come from the characteristic polynomial of a matrix. For a square matrix A,

$$p(\lambda) = \det(A - \lambda I)$$

The roots of this polynomial are the eigenvalues of A.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix, symbols
```

Step-by-Step Code Walkthrough

1. 2×2 example

$$A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

```
  = symbols(' ')
A = Matrix([[2,1],[1,2]])
char_poly = A.charpoly( )
print("Characteristic polynomial:", char_poly.as_expr())
print("Eigenvalues (roots):", char_poly.all_roots())
```

```
Characteristic polynomial:  **2 - 4*  + 3
Eigenvalues (roots): [1, 3]
```

Polynomial: $\lambda^2 - 4\lambda + 3$. Roots: $\lambda = 3, 1$.

2. Verify with eigen computation

```
print("Eigenvalues directly:", A.eigenvals())
```

```
Eigenvalues directly: {3: 1, 1: 1}
```

Matches the roots of the polynomial.

3. 3×3 example

```
B = Matrix([
    [1,2,3],
    [0,1,4],
    [5,6,0]
])

char_poly_B = B.charpoly( )
print("Characteristic polynomial of B:", char_poly_B.as_expr())
print("Eigenvalues of B:", char_poly_B.all_roots())
```

```
Characteristic polynomial of B:  **3 - 2* **2 - 38*  - 1
Eigenvalues of B: [CRootOf(x**3 - 2*x**2 - 38*x - 1, 0), CRootOf(x**3 - 2*x**2 - 38*x - 1, 1)
```

4. NumPy version

NumPy doesn't give the polynomial directly, but eigenvalues can be checked:

```
B_np = np.array([[1,2,3],[0,1,4],[5,6,0]], dtype=float)
eigvals = np.linalg.eigvals(B_np)
print("NumPy eigenvalues:", eigvals)
```

```
NumPy eigenvalues: [-5.2296696  -0.02635282  7.25602242]
```

5. Relation to trace and determinant

For a 2×2 matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the characteristic polynomial is

$$\lambda^2 - (a+d)\lambda + (ad - bc).$$

- Coefficient of λ: $-\text{trace}(A)$.
- Constant term: $\det(A)$.

```
print("Trace:", A.trace())
print("Determinant:", A.det())
```

```
Trace: 4
Determinant: 3
```

Try It Yourself

1. Compute the characteristic polynomial of

$$\begin{bmatrix} 4 & 0 \\ 0 & 5 \end{bmatrix}$$

   and confirm eigenvalues are 4 and 5.

2. Check the relationship between polynomial coefficients, trace, and determinant for a 3×3 case.

3. Verify with NumPy that the roots of the polynomial equal the eigenvalues.

The Takeaway

- The characteristic polynomial encodes eigenvalues as its roots.
- Coefficients are tied to invariants: trace and determinant.
- This polynomial viewpoint is the bridge from algebraic formulas to geometric eigenbehavior.

63. Algebraic vs. Geometric Multiplicity (How Many and How Independent)

Eigenvalues can repeat, and when they do, two notions of multiplicity arise:

- Algebraic multiplicity: how many times the eigenvalue appears as a root of the characteristic polynomial.
- Geometric multiplicity: the dimension of the eigenspace (number of independent eigenvectors).

Always:

$$1 \leq \text{geometric multiplicity} \leq \text{algebraic multiplicity}$$

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Matrix with repeated eigenvalue

```
A = Matrix([
    [2,1],
    [0,2]
])

print("Eigenvalues and algebraic multiplicity:", A.eigenvals())
print("Eigenvectors:", A.eigenvects())
```

```
Eigenvalues and algebraic multiplicity: {2: 2}
Eigenvectors: [(2, 2, [Matrix([
[1],
[0]])])]
```

- Eigenvalue 2 has algebraic multiplicity = 2.
- But only 1 independent eigenvector → geometric multiplicity = 1.

2. Diagonal matrix with repetition

```
B = Matrix([
    [3,0,0],
    [0,3,0],
    [0,0,3]
])

print("Eigenvalues:", B.eigenvals())
print("Eigenvectors:", B.eigenvects())
```

```
Eigenvalues: {3: 3}
Eigenvectors: [(3, 3, [Matrix([
[1],
[0],
[0]]), Matrix([
[0],
[1],
[0]]), Matrix([
[0],
[0],
[1]])])]
```

Here, eigenvalue 3 has algebraic multiplicity = 3, and geometric multiplicity = 3.

3. NumPy check

```
A_np = np.array([[2,1],[0,2]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A_np)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

```
Eigenvalues: [2. 2.]
Eigenvectors:
 [[ 1.0000000e+00 -1.0000000e+00]
 [ 0.0000000e+00  4.4408921e-16]]
```

NumPy won't show multiplicities directly, but you can see repeated eigenvalues.

4. Comparing two cases

- Defective matrix: Algebraic > geometric (like the upper triangular A).
- Diagonalizable matrix: Algebraic = geometric (like B).

This distinction determines whether a matrix can be diagonalized.

Try It Yourself

1. Compute algebraic and geometric multiplicities of

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

   (hint: only one eigenvector).

2. Take a diagonal matrix with repeated entries - what happens to multiplicities?
3. Test a random 3×3 singular matrix. Does 0 have algebraic multiplicity > 1?

The Takeaway

- Algebraic multiplicity = count of root in characteristic polynomial.
- Geometric multiplicity = dimension of eigenspace.
- If they match for all eigenvalues → matrix is diagonalizable.

64. Diagonalization (When a Matrix Becomes Simple)

A matrix A is diagonalizable if it can be written as

$$A = PDP^{-1}$$

- D is diagonal (containing eigenvalues).
- Columns of P are the eigenvectors.

This means A acts like simple scaling in a "better" coordinate system.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. A diagonalizable 2×2 matrix

```
A = Matrix([
    [4,1],
    [2,3]
])

P, D = A.diagonalize()
print("P (eigenvectors):")
print(P)
print("D (eigenvalues on diagonal):")
print(D)

# Verify A = P D P^-1
print("Check:", P*D*P.inv())
```

```
P (eigenvectors):
Matrix([[-1, 1], [2, 1]])
D (eigenvalues on diagonal):
Matrix([[2, 0], [0, 5]])
Check: Matrix([[4, 1], [2, 3]])
```

2. A non-diagonalizable matrix

```
B = Matrix([
    [2,1],
    [0,2]
])

try:
    P, D = B.diagonalize()
    print("Diagonalization successful")
except Exception as e:
    print("Not diagonalizable:", e)
```

```
Not diagonalizable: Matrix is not diagonalizable
```

This fails because eigenvalue 2 has algebraic multiplicity 2 but geometric multiplicity 1.

3. Diagonalization with NumPy

NumPy doesn't diagonalize explicitly, but we can build P and D ourselves:

```
A_np = np.array([[4,1],[2,3]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A_np)

P = eigvecs
D = np.diag(eigvals)
Pinv = np.linalg.inv(P)

print("Check A = PDP^-1:\n", P @ D @ Pinv)
```

```
Check A = PDP^-1:
 [[4. 1.]
 [2. 3.]]
```

4. Powers of a diagonalizable matrix

One reason diagonalization is powerful:

$$A^k = PD^kP^{-1}$$

Since D^k is trivial (just raise each diagonal entry to power k).

```python
k = 5

A_power = np.linalg.matrix_power(A, k)
D_power = np.linalg.matrix_power(D, k)
A_via_diag = P @ D_power @ np.linalg.inv(P)

print("A^5 via diagonalization:\n", A_via_diag)
print("Direct A^5:\n", A_power)
```

```
A^5 via diagonalization:
 [[2094. 1031.]
 [2062. 1063.]]
Direct A^5:
 [[2094 1031]
 [2062 1063]]
```

Both match.

Try It Yourself

1. Check whether

$$\begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$$

   is diagonalizable.

2. Try diagonalizing a rotation matrix by 90°. Do you get complex eigenvalues?
3. Verify the formula $A^k = PD^kP^{-1}$ for a 3×3 diagonalizable matrix.

The Takeaway

- Diagonalization rewrites a matrix in its simplest form.
- Works if there are enough independent eigenvectors.
- It makes powers of A easy, and is the gateway to analyzing dynamics.

65. Powers of a Matrix (Long-Term Behavior via Eigenvalues)

One of the most useful applications of eigenvalues and diagonalization is computing powers of a matrix:

$$A^k = PD^kP^{-1}$$

where D is diagonal with eigenvalues of A. Each eigenvalue λ raised to k dictates how its eigenvector direction grows, decays, or oscillates over time.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Simple diagonal matrix

If $D = \text{diag}(2, 3)$:

```
D = Matrix([[2,0],[0,3]])
print("D^5 =")
print(D**5)
```

```
D^5 =
Matrix([[32, 0], [0, 243]])
```

Eigenvalues are 2 and 3. Raising to the 5th power just raises each eigenvalue to the 5th: $2^5, 3^5$.

2. Non-diagonal matrix

```
A = Matrix([
    [4,1],
    [2,3]
])

P, D = A.diagonalize()
print("D (eigenvalues):")
print(D)

# Compute A^10 via diagonalization
A10 = P * (D**10) * P.inv()
print("A^10 =")
print(A10)
```

```
D (eigenvalues):
Matrix([[2, 0], [0, 5]])
A^10 =
Matrix([[6510758, 3254867], [6509734, 3255891]])
```

Much easier than multiplying A ten times!

3. NumPy version

```
A_np = np.array([[4,1],[2,3]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A_np)

k = 10
D_power = np.diag(eigvals**k)
A10_np = eigvecs @ D_power @ np.linalg.inv(eigvecs)

print("A^10 via eigen-decomposition:\n", A10_np)
```

```
A^10 via eigen-decomposition:
 [[6510758. 3254867.]
 [6509734. 3255891.]]
```

4. Long-term behavior

Eigenvalues tell us what happens as $k \to \infty$:

- If $|\lambda| < 1 \to$ decay to 0.
- If $|\lambda| > 1 \to$ grows unbounded.

- If $|\lambda| = 1 \rightarrow$ oscillates or stabilizes.

```
B = Matrix([
    [0.5,0],
    [0,1.2]
])

P, D = B.diagonalize()
print("Eigenvalues:", D)
print("B^20:", P*(D**20)*P.inv())
```

```
Eigenvalues: Matrix([[0.500000000000000, 0], [0, 1.20000000000000]])
B^20: Matrix([[9.53674316406250e-7, 0], [0, 38.3375999244747]])
```

Here, the component along eigenvalue 0.5 decays, while eigenvalue 1.2 grows.

Try It Yourself

1. Compute A^{50} for a diagonal matrix with eigenvalues 0.9 and 1.1. Which component dominates?
2. Take a stochastic (Markov) matrix and compute powers. Do the rows stabilize?
3. Experiment with complex eigenvalues (like a rotation) and check if the powers oscillate.

The Takeaway

- Matrix powers are simple when using eigenvalues.
- Long-term dynamics are controlled by eigenvalue magnitudes.
- This insight is critical in Markov chains, stability analysis, and dynamical systems.

66. Real vs. Complex Spectra (Rotations and Oscillations)

Not all eigenvalues are real. Some matrices, especially those involving rotations, have complex eigenvalues. Complex eigenvalues often describe oscillations or rotations in systems.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Rotation matrix in 2D

A 90° rotation matrix:

$$R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

```
R = Matrix([[0, -1],
            [1,  0]])

print("Characteristic polynomial:", R.charpoly())
print("Eigenvalues:", R.eigenvals())
```

```
Characteristic polynomial: PurePoly(lambda**2 + 1, lambda, domain='ZZ')
Eigenvalues: {-I: 1, I: 1}
```

Result: eigenvalues are i and $-i$ (purely imaginary).

2. Verify eigen-equation with complex numbers

```
eigs = R.eigenvects()
for eig in eigs:
    lam = eig[0]
    v = eig[2][0]
    print(f"  = {lam}, Av = {R*v},  v = {lam*v}")
```

```
  = -I, Av = Matrix([[-1], [-I]]),  v = Matrix([[-1], [-I]])
  = I, Av = Matrix([[-1], [I]]),  v = Matrix([[-1], [I]])
```

3. NumPy version

```
R_np = np.array([[0,-1],[1,0]], dtype=float)
eigvals, eigvecs = np.linalg.eig(R_np)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

```
Eigenvalues: [0.+1.j 0.-1.j]
Eigenvectors:
 [[0.70710678+0.j         0.70710678-0.j        ]
 [0.         -0.70710678j 0.         +0.70710678j]]
```

NumPy shows complex eigenvalues with `j` (Python's imaginary unit).

4. Rotation by arbitrary angle

General 2D rotation:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Eigenvalues:

$$\lambda = e^{\pm i\theta} = \cos\theta \pm i\sin\theta$$

```
theta = np.pi/4  # 45 degrees
R_theta = np.array([[np.cos(theta), -np.sin(theta)],
                    [np.sin(theta),  np.cos(theta)]])

eigvals, eigvecs = np.linalg.eig(R_theta)
print("Eigenvalues (rotation 45°):", eigvals)
```

```
Eigenvalues (rotation 45°): [0.70710678+0.70710678j 0.70710678-0.70710678j]
```

5. Oscillation insight

- Complex eigenvalues with $|\lambda| = 1$ → pure oscillation (no growth).
- If $|\lambda| < 1$ → decaying spiral.
- If $|\lambda| > 1$ → growing spiral.

Example:

```
A = np.array([[0.8, -0.6],
              [0.6,  0.8]])

eigvals, _ = np.linalg.eig(A)
print("Eigenvalues:", eigvals)
```

```
Eigenvalues: [0.8+0.6j 0.8-0.6j]
```

These eigenvalues lie inside the unit circle → spiral decay.

Try It Yourself

1. Compute eigenvalues of a 180° rotation. What happens?
2. Modify the rotation matrix to include scaling (e.g., multiply by 1.1). Do the eigenvalues lie outside the unit circle?
3. Plot the trajectory of repeatedly applying a rotation matrix to a vector.

The Takeaway

- Complex eigenvalues naturally appear in rotations and oscillatory systems.
- Their magnitude controls growth or decay; their angle controls oscillation.
- This is a key link between linear algebra and dynamics in physics and engineering.

67. Defective Matrices and a Peek at Jordan Form (When Diagonalization Fails)

Not every matrix has enough independent eigenvectors to be diagonalized. Such matrices are called defective. To handle them, mathematicians use the Jordan normal form, which extends diagonalization with extra structure.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. A defective example

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

```
A = Matrix([[2,1],
            [0,2]])

print("Eigenvalues:", A.eigenvals())
print("Eigenvectors:", A.eigenvects())
```

```
Eigenvalues: {2: 2}
Eigenvectors: [(2, 2, [Matrix([
[1],
[0]])])]
```

- Eigenvalue 2 has algebraic multiplicity = 2.
- Only 1 eigenvector exists → geometric multiplicity = 1.

Thus A is defective, not diagonalizable.

2. Attempt diagonalization

```
try:
    P, D = A.diagonalize()
    print("Diagonal form:", D)
except Exception as e:
    print("Diagonalization failed:", e)
```

```
Diagonalization failed: Matrix is not diagonalizable
```

You'll see an error - confirming A is not diagonalizable.

3. Jordan form in SymPy

```
J, P = A.jordan_form()
print("Jordan form J:")
print(J)
print("P (generalized eigenvectors):")
print(P)
```

```
Jordan form J:
Matrix([[1, 0], [0, 1]])
P (generalized eigenvectors):
Matrix([[2, 1], [0, 2]])
```

The Jordan form shows a Jordan block:

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

This block structure represents the failure of diagonalization.

4. NumPy perspective

NumPy doesn't compute Jordan form, but you can see repeated eigenvalues and lack of eigenvectors:

```
A_np = np.array([[2,1],[0,2]], dtype=float)
eigvals, eigvecs = np.linalg.eig(A_np)
print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

```
Eigenvalues: [2. 2.]
Eigenvectors:
 [[ 1.0000000e+00 -1.0000000e+00]
 [ 0.0000000e+00  4.4408921e-16]]
```

The eigenvectors matrix has fewer independent columns than expected.

5. Generalized eigenvectors

Jordan form introduces generalized eigenvectors, which satisfy:

$$(A - \lambda I)^k v = 0 \quad \text{for some } k > 1$$

They "fill the gap" when ordinary eigenvectors are insufficient.

Try It Yourself

1. Test diagonalizability of

$$\begin{bmatrix} 3 & 1 \\ 0 & 3 \end{bmatrix}$$

   and compare with its Jordan form.

2. Try a 3×3 defective matrix with one Jordan block of size 3.
3. Verify that Jordan blocks still capture the correct eigenvalues.

The Takeaway

- Defective matrices lack enough eigenvectors for diagonalization.
- Jordan form replaces diagonalization with blocks, keeping eigenvalues on the diagonal.
- Understanding Jordan blocks is essential for advanced linear algebra and differential equations.

68. Stability and Spectral Radius (Grow, Decay, or Oscillate)

The spectral radius of a matrix A is defined as

$$\rho(A) = \max_i |\lambda_i|$$

where λ_i are the eigenvalues. It tells us the long-term behavior of repeated applications of A:

- If $\rho(A) < 1 \rightarrow$ powers of A tend to 0 (stable/decay).
- If $\rho(A) = 1 \rightarrow$ powers neither blow up nor vanish (neutral, may oscillate).
- If $\rho(A) > 1 \rightarrow$ powers diverge (unstable/growth).

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Stable matrix ($\rho < 1$)

```
A = np.array([[0.5, 0],
              [0, 0.3]])

eigvals = np.linalg.eigvals(A)
spectral_radius = max(abs(eigvals))

print("Eigenvalues:", eigvals)
print("Spectral radius:", spectral_radius)

print("A^10:\n", np.linalg.matrix_power(A, 10))
```

```
Eigenvalues: [0.5 0.3]
Spectral radius: 0.5
A^10:
 [[9.765625e-04 0.000000e+00]
 [0.000000e+00 5.904900e-06]]
```

All entries shrink toward zero.

2. Unstable matrix ($\rho > 1$)

```
B = np.array([[1.2, 0],
              [0, 0.9]])

eigvals = np.linalg.eigvals(B)
print("Eigenvalues:", eigvals, "Spectral radius:", max(abs(eigvals)))
print("B^10:\n", np.linalg.matrix_power(B, 10))
```

```
Eigenvalues: [1.2 0.9] Spectral radius: 1.2
B^10:
 [[6.19173642 0.        ]
 [0.         0.34867844]]
```

The component along eigenvalue 1.2 grows quickly.

3. Neutral/oscillatory case ($\rho = 1$)

90° rotation matrix:

```
R = np.array([[0, -1],
              [1,  0]])

eigvals = np.linalg.eigvals(R)
print("Eigenvalues:", eigvals)
print("Spectral radius:", max(abs(eigvals)))
print("R^4:\n", np.linalg.matrix_power(R, 4))
```

```
Eigenvalues: [0.+1.j 0.-1.j]
Spectral radius: 1.0
R^4:
 [[1 0]
 [0 1]]
```

Eigenvalues are ±i, with modulus 1 → pure oscillation.

4. Spectral radius with SymPy

```
M = Matrix([[2,1],[1,2]])
eigs = M.eigenvals()
print("Eigenvalues:", eigs)
print("Spectral radius:", max(abs(ev) for ev in eigs))
```

```
Eigenvalues: {3: 1, 1: 1}
Spectral radius: 3
```

Try It Yourself

1. Build a diagonal matrix with entries 0.8, 1.0, and 1.1. Predict which direction dominates as powers grow.
2. Apply a random matrix repeatedly to a vector. Does it shrink, grow, or oscillate?
3. Check if a Markov chain transition matrix always has spectral radius 1.

The Takeaway

- The spectral radius is the key number that predicts growth, decay, or oscillation.
- Long-term stability in dynamical systems is governed entirely by eigenvalue magnitudes.
- This connects linear algebra directly to control theory, Markov chains, and differential equations.

69. Markov Chains and Steady States (Probabilities as Linear Algebra)

A Markov chain is a process that moves between states according to probabilities. The transitions are encoded in a stochastic matrix P:

- Each entry $p_{ij} \geq 0$
- Each row sums to 1

If we start with a probability vector v_0, then after k steps:

$$v_k = v_0 P^k$$

A steady state is a probability vector v such that $vP = v$. It corresponds to eigenvalue $\lambda = 1$.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix
```

Step-by-Step Code Walkthrough

1. Simple two-state chain

```
P = np.array([
    [0.9, 0.1],
    [0.5, 0.5]
])

v0 = np.array([1.0, 0.0])  # start in state 1
for k in [1, 2, 5, 10, 50]:
    vk = v0 @ np.linalg.matrix_power(P, k)
    print(f"Step {k}: {vk}")
```

```
Step 1: [0.9 0.1]
Step 2: [0.86 0.14]
Step 5: [0.83504 0.16496]
Step 10: [0.83335081 0.16664919]
Step 50: [0.83333333 0.16666667]
```

The distribution stabilizes as k increases.

2. Steady state via eigenvector

Find eigenvector for eigenvalue 1:

```
eigvals, eigvecs = np.linalg.eig(P.T)
steady_state = eigvecs[:, np.isclose(eigvals, 1)]
steady_state = steady_state / steady_state.sum()
print("Steady state:", steady_state.real.flatten())
```

```
Steady state: [0.83333333 0.16666667]
```

3. SymPy exact check

```
P_sym = Matrix([[0.9,0.1],[0.5,0.5]])
steady = P_sym.eigenvects()
print("Eigen info:", steady)
```

```
Eigen info: [(1.00000000000000, 1, [Matrix([
[0.707106781186548],
[0.707106781186547]])]), (0.400000000000000, 1, [Matrix([
[-0.235702260395516],
[  1.17851130197758]])])]
```

4. A 3-state example

```
Q = np.array([
    [0.3, 0.7, 0.0],
    [0.2, 0.5, 0.3],
    [0.1, 0.2, 0.7]
])

eigvals, eigvecs = np.linalg.eig(Q.T)
steady = eigvecs[:, np.isclose(eigvals, 1)]
steady = steady / steady.sum()
print("Steady state for Q:", steady.real.flatten())
```

```
Steady state for Q: [0.17647059 0.41176471 0.41176471]
```

Try It Yourself

1. Create a transition matrix where one state is absorbing (e.g., row = [0,0,1]). What happens to the steady state?
2. Simulate a random walk on 3 states. Does the steady state distribute evenly?
3. Compare long-run simulation with eigenvector computation.

The Takeaway

- Markov chains evolve by repeated multiplication with a stochastic matrix.
- Steady states are eigenvectors with eigenvalue 1.
- This framework powers real applications like PageRank, weather models, and queuing systems.

70. Linear Differential Systems (Solutions via Eigen-Decomposition)

Linear differential equations often reduce to systems of the form:

$$\frac{d}{dt}x(t) = Ax(t)$$

where A is a matrix and $x(t)$ is a vector of functions. The solution is given by the matrix exponential:

$$x(t) = e^{At}x(0)$$

If A is diagonalizable, this becomes simple using eigenvalues and eigenvectors.

Set Up Your Lab

```
import numpy as np
from sympy import Matrix, exp, symbols
from scipy.linalg import expm
```

Step-by-Step Code Walkthrough

1. Simple system with diagonal matrix

$$A = \begin{bmatrix} -1 & 0 \\ 0 & 2 \end{bmatrix}$$

```
A = Matrix([[-1,0],
            [0, 2]])
t = symbols('t')
expAt = (A*t).exp()
print("e^{At} =")
print(expAt)
```

```
e^{At} =
Matrix([[exp(-t), 0], [0, exp(2*t)]])
```

Solution:

$$x(t) = \begin{bmatrix} e^{-t} & 0 \\ 0 & e^{2t} \end{bmatrix} x(0)$$

One component decays, the other grows.

2. Non-diagonal example

```
B = Matrix([[0,1],
            [-2,-3]])
expBt = (B*t).exp()
print("e^{Bt} =")
print(expBt)
```

```
e^{Bt} =
Matrix([[2*exp(-t) - exp(-2*t), exp(-t) - exp(-2*t)], [-2*exp(-t) + 2*exp(-
2*t), -exp(-t) + 2*exp(-2*t)]])
```

Here the solution involves exponentials and possibly sines/cosines (oscillatory behavior).

3. Numeric computation with SciPy

```
import numpy as np
from scipy.linalg import expm

A = np.array([[-1,0],[0,2]], dtype=float)
t = 1.0
print("Matrix exponential e^{At} at t=1:\n", expm(A*t))
```

```
Matrix exponential e^{At} at t=1:
 [[0.36787944 0.        ]
 [0.         7.3890561 ]]
```

This computes e^{At} numerically.

4. Simulation of a trajectory

```
x0 = np.array([1.0, 1.0])
for t in [0, 0.5, 1, 2]:
    xt = expm(A*t) @ x0
    print(f"x({t}) = {xt}")
```

```
x(0) = [1. 1.]
x(0.5) = [0.60653066 2.71828183]
x(1) = [0.36787944 7.3890561 ]
x(2) = [ 0.13533528 54.59815003]
```

One coordinate decays, the other explodes with time.

Try It Yourself

1. Solve the system $\dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} x$. What kind of motion do you see?
2. Use SciPy to simulate a system with eigenvalues less than 0. Does it always decay?
3. Try an unstable system with eigenvalues > 0 and watch how trajectories diverge.

The Takeaway

- Linear systems $\dot{x} = Ax$ are solved via the matrix exponential.
- Eigenvalues determine stability: negative real parts = stable, positive = unstable, imaginary = oscillations.
- This ties linear algebra directly to differential equations and dynamical systems.

Chapter 8. Orthogonality, least squars, and QR

71. Inner Products Beyond Dot Product (Custom Notions of Angle)

The dot product is the standard inner product in $\mathbb{R}^n$, but linear algebra allows us to define more general inner products that measure length and angle in different ways.

An inner product on a vector space is a function $\langle u, v \rangle$ that satisfies:

1. Linearity in the first argument.
2. Symmetry: $\langle u, v \rangle = \langle v, u \rangle$.
3. Positive definiteness: $\langle v, v \rangle \geq 0$ and equals 0 only if $v = 0$.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Standard dot product

```
u = np.array([1,2,3])
v = np.array([4,5,6])

print("Dot product:", np.dot(u,v))
```

```
Dot product: 32
```

This is the familiar formula: $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$.

2. Weighted inner product

We can define:

$$\langle u, v \rangle_W = u^T W v$$

where W is a positive definite matrix.

```
W = np.array([[2,0,0],
              [0,1,0],
              [0,0,3]])

def weighted_inner(u,v,W):
    return u.T @ W @ v

print("Weighted inner product:", weighted_inner(u,v,W))
```

```
Weighted inner product: 72
```

Here, some coordinates "count more" than others.

3. Check symmetry and positivity

```
print(" u,v  ==  v,u  ?", weighted_inner(u,v,W) == weighted_inner(v,u,W))
print(" u,u  (should be >0):", weighted_inner(u,u,W))
```

```
 u,v  ==  v,u  ? True
 u,u  (should be >0): 33
```

4. Angle with weighted inner product

$$\cos\theta = \frac{\langle u, v\rangle_W}{\|u\|_W \|v\|_W}$$

```
def weighted_norm(u,W):
    return np.sqrt(weighted_inner(u,u,W))

cos_theta = weighted_inner(u,v,W) / (weighted_norm(u,W) * weighted_norm(v,W))
print("Cosine of angle (weighted):", cos_theta)
```

```
Cosine of angle (weighted): 0.97573875381809
```

5. Custom example: correlation inner product

For statistics, an inner product can be defined as covariance or correlation. Example with mean-centered vectors:

```
x = np.array([2,4,6])
y = np.array([1,3,5])

x_centered = x - x.mean()
y_centered = y - y.mean()

corr_inner = np.dot(x_centered,y_centered)
print("Correlation-style inner product:", corr_inner)
```

```
Correlation-style inner product: 8.0
```

Try It Yourself

1. Define a custom inner product with $W = \text{diag}(1, 10, 100)$. How does it change angles between vectors?
2. Verify positivity: compute $\langle v, v\rangle_W$ for a random vector v.
3. Compare dot product vs weighted inner product on the same pair of vectors.

The Takeaway

- Inner products generalize the dot product to new "geometries."
- By changing the weight matrix W, you change how lengths and angles are measured.
- This flexibility is essential in statistics, optimization, and machine learning.

72. Orthogonality and Orthonormal Bases (Perpendicular Power)

Two vectors are orthogonal if their inner product is zero:

$$\langle u, v \rangle = 0$$

If, in addition, each vector has length 1, the set is orthonormal. Orthonormal bases are extremely useful because they simplify computations: projections, decompositions, and coordinate changes all become clean.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Check orthogonality

```
u = np.array([1, -1])
v = np.array([1, 1])

print("Dot product:", np.dot(u,v))
```

```
Dot product: 0
```

Since the dot product is 0, they're orthogonal.

2. Normalizing vectors

$$\hat{u} = \frac{u}{\|u\|}$$

```
def normalize(vec):
    return vec / np.linalg.norm(vec)

u_norm = normalize(u)
v_norm = normalize(v)

print("Normalized u:", u_norm)
print("Normalized v:", v_norm)
```

```
Normalized u: [ 0.70710678 -0.70710678]
Normalized v: [0.70710678 0.70710678]
```

Now both have length 1.

3. Form an orthonormal basis

```
basis = np.column_stack((u_norm, v_norm))
print("Orthonormal basis:\n", basis)

print("Check inner products:\n", basis.T @ basis)
```

```
Orthonormal basis:
 [[ 0.70710678  0.70710678]
 [-0.70710678  0.70710678]]
Check inner products:
 [[ 1.00000000e+00 -2.23711432e-17]
 [-2.23711432e-17  1.00000000e+00]]
```

The result is the identity matrix → perfectly orthonormal.

4. Apply to coordinates

If $x = [2, 3]$, coordinates in the orthonormal basis are:

```
x = np.array([2,3])
coords = basis.T @ x
print("Coordinates in new basis:", coords)
print("Reconstruction:", basis @ coords)
```

```
Coordinates in new basis: [-0.70710678  3.53553391]
Reconstruction: [2. 3.]
```

It reconstructs exactly.

5. Random example with QR

Any set of linearly independent vectors can be orthonormalized (Gram–Schmidt, or QR decomposition):

```
M = np.random.rand(3,3)
Q, R = np.linalg.qr(M)
print("Q (orthonormal basis):\n", Q)
print("Check Q^T Q = I:\n", Q.T @ Q)
```

```
Q (orthonormal basis):
 [[-0.37617518  0.91975919 -0.111961  ]
 [-0.82070726 -0.38684608 -0.42046368]
 [-0.430037   -0.06628079  0.90037494]]
Check Q^T Q = I:
 [[ 1.00000000e+00 -1.29639194e-16 -1.91943696e-17]
 [-1.29639194e-16  1.00000000e+00  5.38253498e-17]
 [-1.91943696e-17  5.38253498e-17  1.00000000e+00]]
```

Try It Yourself

1. Create two 3D vectors and check if they're orthogonal.
2. Normalize them to form an orthonormal set.
3. Use `np.linalg.qr` on a 4×3 random matrix and verify that the columns of Q are orthonormal.

The Takeaway

- Orthogonality means perpendicularity; orthonormality adds unit length.
- Orthonormal bases simplify coordinate systems, making inner products and projections easy.
- QR decomposition is the practical tool to generate orthonormal bases in higher dimensions.

73. Gram–Schmidt Process (Constructing Orthonormal Bases)

The Gram–Schmidt process takes a set of linearly independent vectors and turns them into an orthonormal basis. This is crucial for working with subspaces, projections, and numerical stability.

Given vectors $v_1, v_2, \ldots, v_n$:

1. Set $u_1 = v_1$.

2. Subtract projections to make each new vector orthogonal to the earlier ones:

$$u_k = v_k - \sum_{j=1}^{k-1} \frac{\langle v_k, u_j \rangle}{\langle u_j, u_j \rangle} u_j$$

3. Normalize:

$$e_k = \frac{u_k}{\|u_k\|}$$

The set $\{e_1, e_2, \ldots, e_n\}$ is orthonormal.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Define vectors

```
v1 = np.array([1.0, 1.0, 0.0])
v2 = np.array([1.0, 0.0, 1.0])
v3 = np.array([0.0, 1.0, 1.0])
V = [v1, v2, v3]
```

2. Implement Gram–Schmidt

```
def gram_schmidt(V):
    U = []
    for v in V:
        u = v.copy()
        for uj in U:
            u -= np.dot(v, uj) / np.dot(uj, uj) * uj
        U.append(u)
    # Normalize
    E = [u/np.linalg.norm(u) for u in U]
    return np.array(E)

E = gram_schmidt(V)
```

```
print("Orthonormal basis:\n", E)
print("Check orthonormality:\n", np.round(E @ E.T, 6))
```

```
Orthonormal basis:
 [[ 0.70710678  0.70710678  0.        ]
 [ 0.40824829 -0.40824829  0.81649658]
 [-0.57735027  0.57735027  0.57735027]]
Check orthonormality:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

3. Compare with NumPy QR

```
Q, R = np.linalg.qr(np.column_stack(V))
print("QR-based orthonormal basis:\n", Q)
print("Check Q^T Q = I:\n", np.round(Q.T @ Q, 6))
```

```
QR-based orthonormal basis:
 [[-0.70710678  0.40824829 -0.57735027]
 [-0.70710678 -0.40824829  0.57735027]
 [-0.          0.81649658  0.57735027]]
Check Q^T Q = I:
 [[ 1.  0. -0.]
 [ 0.  1. -0.]
 [-0. -0.  1.]]
```

Both methods give orthonormal bases.

4. Application: projection

To project a vector x onto the span of V:

```
x = np.array([2.0, 2.0, 2.0])
proj = sum((x @ e) * e for e in E)
print("Projection of x onto span(V):", proj)
```

```
Projection of x onto span(V): [2. 2. 2.]
```

Try It Yourself

1. Run Gram–Schmidt on two vectors in 2D. Compare with just normalizing and checking orthogonality.
2. Replace one vector with a linear combination of others. What happens?
3. Use QR decomposition on a 4×3 random matrix and compare with Gram–Schmidt.

The Takeaway

- Gram–Schmidt converts arbitrary independent vectors into an orthonormal basis.
- Orthonormal bases simplify projections, decompositions, and computations.
- In practice, QR decomposition is often used as a numerically stable implementation.

74. Orthogonal Projections onto Subspaces (Closest Point Principle)

Given a subspace spanned by vectors, the orthogonal projection of a vector x onto the subspace is the point in the subspace that is closest to x. This is a cornerstone idea in least squares, data fitting, and signal processing.

Formula Recap

If Q is a matrix with orthonormal columns spanning the subspace, the projection of x is:

$$\text{proj}(x) = QQ^T x$$

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Projection onto a line (1D subspace)

Suppose the subspace is spanned by $u = [1, 2]$.

```
u = np.array([1.0,2.0])
x = np.array([3.0,1.0])

u_norm = u / np.linalg.norm(u)
proj = np.dot(x, u_norm) * u_norm
print("Projection of x onto span(u):", proj)
```

```
Projection of x onto span(u): [1. 2.]
```

This gives the closest point to x along the line spanned by u.

2. Projection onto a plane (2D subspace in 3D)

```
u1 = np.array([1.0,0.0,0.0])
u2 = np.array([0.0,1.0,0.0])
Q = np.column_stack([u1,u2])   # Orthonormal basis for xy-plane

x = np.array([2.0,3.0,5.0])
proj = Q @ Q.T @ x
print("Projection of x onto xy-plane:", proj)
```

```
Projection of x onto xy-plane: [2. 3. 0.]
```

Result drops the z-component → projection onto the plane.

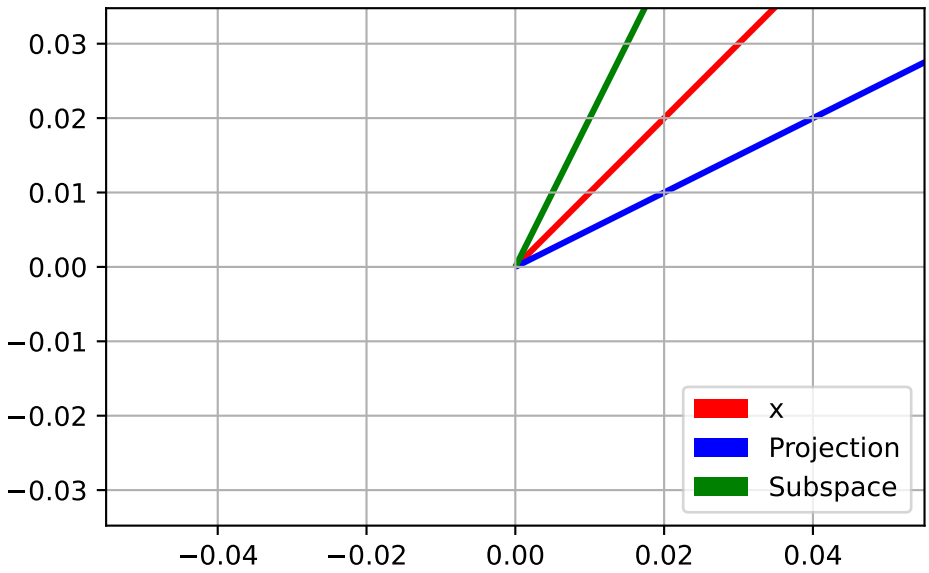
3. General projection using QR

```
A = np.array([[1,1,0],
              [0,1,1],
              [1,0,1]], dtype=float)

Q, R = np.linalg.qr(A)
Q = Q[:, :2]   # take first 2 independent columns
x = np.array([2,2,2], dtype=float)

proj = Q @ Q.T @ x
print("Projection of x onto span(A):", proj)
```

```
Projection of x onto span(A): [2.66666667 1.33333333 1.33333333]
```

4. Visualization (2D case)

```
import matplotlib.pyplot as plt

plt.quiver(0,0,x[0],x[1],angles='xy',scale_units='xy',scale=1,color='red',label="x")
plt.quiver(0,0,proj[0],proj[1],angles='xy',scale_units='xy',scale=1,color='blue',label="Proj
plt.quiver(0,0,u[0],u[1],angles='xy',scale_units='xy',scale=1,color='green',label="Subspace"
plt.axis('equal'); plt.grid(); plt.legend(); plt.show()
```



Try It Yourself

1. Project a vector onto the line spanned by $[2, 1]$.
2. Project $[1, 2, 3]$ onto the plane spanned by $[1, 0, 1]$ and $[0, 1, 1]$.
3. Compare projection via formula $QQ^T x$ with manually solving least squares.

The Takeaway

- Orthogonal projection finds the closest point in a subspace.
- Formula $QQ^T x$ works perfectly when Q has orthonormal columns.
- Projections are the foundation of least squares, PCA, and many geometric algorithms.

75. Least-Squares Problems (Fit When Exact Solve Is Impossible)

Sometimes a system of equations $Ax = b$ has no exact solution - usually because it's overdetermined (more equations than unknowns). In this case, we look for an approximate solution x^* that minimizes the error:

$$x^* = \arg\min_x \|Ax - b\|^2$$

This is the least-squares solution, which geometrically is the projection of b onto the column space of A.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Overdetermined system

3 equations, 2 unknowns:

```
A = np.array([[1,1],
              [1,2],
              [1,3]], dtype=float)
b = np.array([6, 0, 0], dtype=float)
```

2. Solve least squares with NumPy

```
x_star, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
print("Least squares solution:", x_star)
print("Residual norm squared:", residuals)
```

```
Least squares solution: [ 8. -3.]
Residual norm squared: [6.]
```

3. Compare with normal equations

$$A^T Ax = A^T b$$

```
x_normal = np.linalg.inv(A.T @ A) @ (A.T @ b)
print("Solution via normal equations:", x_normal)
```

```
Solution via normal equations: [ 8. -3.]
```

4. Geometric picture

The least-squares solution projects b onto the column space of A:

```
proj = A @ x_star
print("Projection of b onto Col(A):", proj)
print("Original b:", b)
print("Error vector (b - proj):", b - proj)
```

```
Projection of b onto Col(A): [ 5.  2. -1.]
Original b: [6. 0. 0.]
Error vector (b - proj): [ 1. -2.  1.]
```

The error vector is orthogonal to the column space.

5. Verify orthogonality condition

$$A^T(b - Ax^*) = 0$$

```
print("Check orthogonality:", A.T @ (b - A @ x_star))
```

```
Check orthogonality: [0. 0.]
```

The result should be (close to) zero.

Try It Yourself

1. Create a taller A (say 5×2) with random numbers and solve least squares for a random b.
2. Compare the residual from `np.linalg.lstsq` with geometric intuition (projection).
3. Modify b so that the system has an exact solution. Check if least squares gives it exactly.

The Takeaway

- Least-squares finds the best-fit solution when no exact solution exists.
- It works by projecting b onto the column space of A.
- This principle underlies regression, curve fitting, and countless applications in data science.

76. Normal Equations and Geometry of Residuals (Why It Works)

The least-squares solution can be found by solving the normal equations:

$$A^T Ax = A^T b$$

This comes from the condition that the residual vector

$$r = b - Ax$$

is orthogonal to the column space of A.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Build an overdetermined system

```
A = np.array([[1,1],
              [1,2],
              [1,3]], dtype=float)
b = np.array([6, 0, 0], dtype=float)
```

2. Solve least squares via normal equations

```
ATA = A.T @ A
ATb = A.T @ b
x_star = np.linalg.solve(ATA, ATb)

print("Least-squares solution x*:", x_star)
```

```
Least-squares solution x*: [ 8. -3.]
```

3. Compute residual and check orthogonality

```
residual = b - A @ x_star
print("Residual vector:", residual)
print("Check A^T r  0:", A.T @ residual)
```

```
Residual vector: [ 1. -2.  1.]
Check A^T r  0: [0. 0.]
```

This verifies the residual is perpendicular to the column space of A.

4. Compare with NumPy's least squares solver

```
x_lstsq, *_ = np.linalg.lstsq(A, b, rcond=None)
print("NumPy lstsq solution:", x_lstsq)
```

```
NumPy lstsq solution: [ 8. -3.]
```

The solutions should match (within numerical precision).

5. Geometric picture

- b is a point in $\mathbb{R}^3$.
- Ax is restricted to lie in the 2D column space of A.
- The least-squares solution picks the Ax closest to b.
- The error vector $r = b - Ax^*$ is orthogonal to the subspace.

```
proj = A @ x_star
print("Projection of b onto Col(A):", proj)
```

```
Projection of b onto Col(A): [ 5.  2. -1.]
```

Try It Yourself

1. Change b to $[1, 1, 1]$. Solve again and check the residual.
2. Use a random tall A (say 6×2) and verify that the residual is always orthogonal.
3. Compute $\|r\|$ and see how it changes when you change b.

The Takeaway

- Least squares works by making the residual orthogonal to the column space.
- Normal equations are the algebraic way to encode this condition.
- This orthogonality principle is the geometric heart of least-squares fitting.

77. QR Factorization (Stable Least Squares via Orthogonality)

While normal equations solve least squares, they can be numerically unstable if A^TA is ill-conditioned. A more stable method uses QR factorization:

$$A = QR$$

- Q: matrix with orthonormal columns
- R: upper triangular matrix

Then the least-squares problem reduces to solving:

$$Rx = Q^Tb$$

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Overdetermined system

```
A = np.array([[1,1],
              [1,2],
              [1,3]], dtype=float)
b = np.array([6, 0, 0], dtype=float)
```

2. QR factorization

```
Q, R = np.linalg.qr(A)
print("Q (orthonormal basis):\n", Q)
print("R (upper triangular):\n", R)
```

```
Q (orthonormal basis):
 [[-5.77350269e-01  7.07106781e-01]
 [-5.77350269e-01  5.55111512e-17]
 [-5.77350269e-01 -7.07106781e-01]]
R (upper triangular):
 [[-1.73205081 -3.46410162]
 [ 0.         -1.41421356]]
```

3. Solve least squares using QR

```
y = Q.T @ b
x_star = np.linalg.solve(R[:2,:], y[:2])  # only top rows matter
print("Least squares solution via QR:", x_star)
```

```
Least squares solution via QR: [ 8. -3.]
```

4. Compare with NumPy's `lstsq`

```
x_lstsq, *_ = np.linalg.lstsq(A, b, rcond=None)
print("NumPy lstsq:", x_lstsq)
```

```
NumPy lstsq: [ 8. -3.]
```

The answers should match closely.

5. Residual check

```
residual = b - A @ x_star
print("Residual vector:", residual)
print("Check orthogonality (Q^T r):", Q.T @ residual)
```

```
Residual vector: [ 1. -2.  1.]
Check orthogonality (Q^T r): [-1.44328993e-15 -1.22124533e-15]
```

Residual is orthogonal to the column space, confirming correctness.

Try It Yourself

1. Solve least squares for a 5×2 random matrix using both normal equations and QR. Compare results.
2. Check stability by making columns of A nearly dependent - see if QR behaves better than normal equations.
3. Compute projection of b using QQ^Tb and confirm it equals Ax^*.

The Takeaway

- QR factorization provides a numerically stable way to solve least squares.
- It avoids the instability of normal equations.
- In practice, modern solvers (like NumPy's `lstsq`) rely on QR or SVD under the hood.

78. Orthogonal Matrices (Length-Preserving Transforms)

An orthogonal matrix Q is a square matrix whose columns (and rows) are orthonormal vectors. Formally:

$$Q^TQ = QQ^T = I$$

Key properties:

- Preserves lengths: $\|Qx\| = \|x\|$
- Preserves dot products: $\langle Qx, Qy\rangle = \langle x, y\rangle$
- Determinant is either $+1$ (rotation) or -1 (reflection)

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Construct a simple orthogonal matrix

90° rotation in 2D:

```
Q = np.array([[0, -1],
              [1,  0]])

print("Q^T Q =\n", Q.T @ Q)
```

```
Q^T Q =
 [[1 0]
 [0 1]]
```

Result = identity → confirms orthogonality.

2. Check length preservation

```
x = np.array([3,4])
print("Original length:", np.linalg.norm(x))
print("Transformed length:", np.linalg.norm(Q @ x))
```

```
Original length: 5.0
Transformed length: 5.0
```

Both lengths match.

3. Check dot product preservation

```
u = np.array([1,0])
v = np.array([0,1])

print("Dot(u,v):", np.dot(u,v))
print("Dot(Q u, Q v):", np.dot(Q @ u, Q @ v))
```

```
Dot(u,v): 0
Dot(Q u, Q v): 0
```

Dot product is preserved.

4. Reflection matrix

Reflection about the x-axis:

```
R = np.array([[1,0],
              [0,-1]])

print("R^T R =\n", R.T @ R)
print("Determinant of R:", np.linalg.det(R))
```

```
R^T R =
 [[1 0]
 [0 1]]
Determinant of R: -1.0
```

Determinant = -1 → reflection.

5. Random orthogonal matrix via QR

```
M = np.random.rand(3,3)
Q, _ = np.linalg.qr(M)
print("Q (random orthogonal):\n", Q)
print("Check Q^T Q  I:\n", np.round(Q.T @ Q, 6))
```

```
Q (random orthogonal):
 [[-0.59472353  0.03725157 -0.80306677]
 [-0.61109913 -0.67000966  0.42147943]
 [-0.52236172  0.74141714  0.42123492]]
Check Q^T Q  I:
 [[ 1.  0. -0.]
 [ 0.  1.  0.]
 [-0.  0.  1.]]
```

Try It Yourself

1. Build a 2D rotation matrix for 45°. Verify it's orthogonal.
2. Check whether scaling matrices (e.g., diag(2, 1)) are orthogonal. Why or why not?
3. Generate a random orthogonal matrix with `np.linalg.qr` and test its determinant.

The Takeaway

- Orthogonal matrices are rigid motions: they rotate or reflect without distorting lengths or angles.
- They play a key role in numerical stability, geometry, and physics.
- Every orthonormal basis corresponds to an orthogonal matrix.

79. Fourier Viewpoint (Expanding in Orthogonal Waves)

The Fourier viewpoint treats functions or signals as combinations of orthogonal waves (sines and cosines). This is just linear algebra: sine and cosine functions form an orthogonal basis, and any signal can be expressed as a linear combination of them.

Formula Recap

For a discrete signal x, the Discrete Fourier Transform (DFT) is:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n/N}, \quad k = 0, \ldots, N-1$$

The inverse DFT reconstructs the signal. Orthogonality of complex exponentials makes this work.

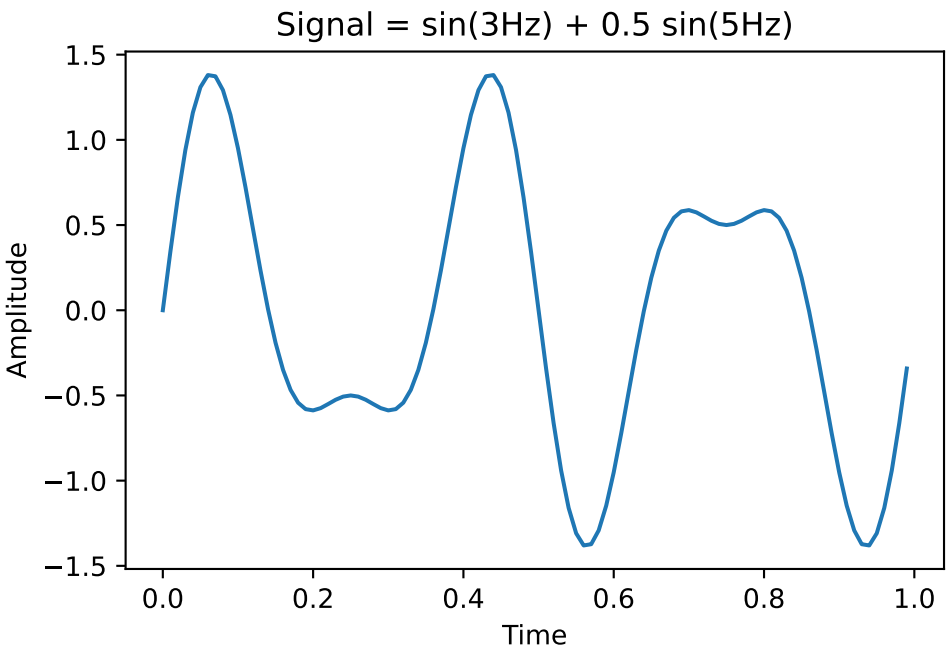
Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

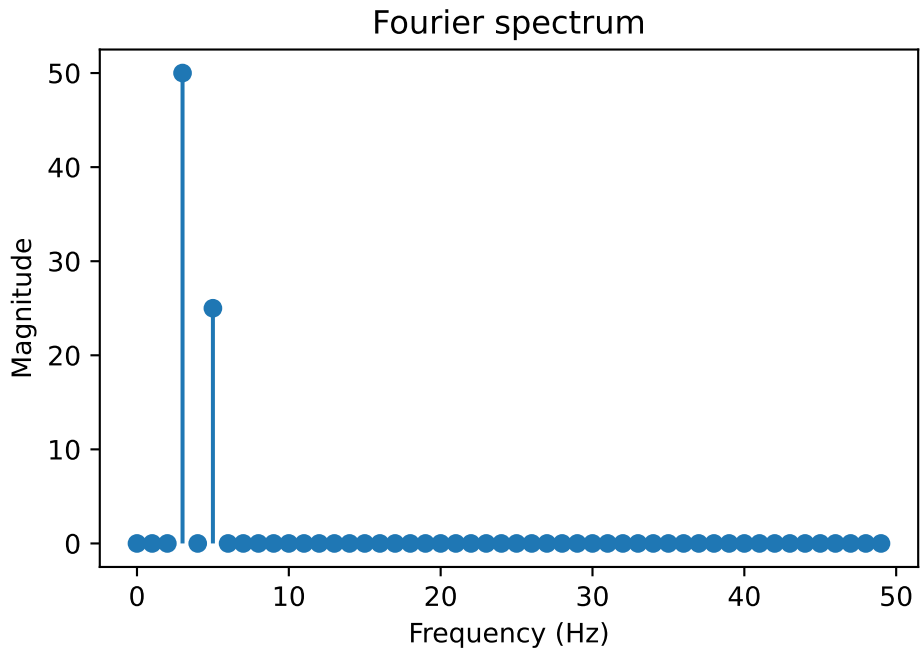
1. Build a simple signal

```
t = np.linspace(0, 1, 100, endpoint=False)
signal = np.sin(2*np.pi*3*t) + 0.5*np.sin(2*np.pi*5*t)
plt.plot(t, signal)
plt.title("Signal = sin(3Hz) + 0.5 sin(5Hz)")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.show()
```



2. Compute Fourier transform (DFT)

```
X = np.fft.fft(signal)
freqs = np.fft.fftfreq(len(t), d=1/100)  # sampling rate = 100Hz

plt.stem(freqs[:50], np.abs(X[:50]), basefmt=" ")
plt.title("Fourier spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.show()
```



Peaks appear at 3Hz and 5Hz → the frequencies of the original signal.

3. Reconstruct signal using inverse FFT

```
signal_reconstructed = np.fft.ifft(X).real
print("Reconstruction error:", np.linalg.norm(signal - signal_reconstructed))
```

```
Reconstruction error: 1.786526604658442e-15
```

Error is near zero → perfect reconstruction.

4. Orthogonality check of sinusoids

```
u = np.sin(2*np.pi*3*t)
v = np.sin(2*np.pi*5*t)

inner = np.dot(u, v)
print("Inner product of 3Hz and 5Hz sinusoids:", inner)
```

```
Inner product of 3Hz and 5Hz sinusoids: 1.2982670494210424e-14
```

The result is 0 → confirms orthogonality.

Try It Yourself

1. Change the frequencies to 7Hz and 9Hz. Do the Fourier peaks move accordingly?
2. Mix in some noise and check how the spectrum looks.
3. Try cosine signals instead of sine. Do you still see orthogonality?

The Takeaway

- Fourier analysis = linear algebra with orthogonal sinusoidal basis functions.
- Any signal can be decomposed into orthogonal waves.
- This orthogonal viewpoint powers audio, image compression, and signal processing.

80. Polynomial and Multifeature Least Squares (Fitting More Flexibly)

Least squares isn't limited to straight lines. By adding polynomial or multiple features, we can fit curves and capture more complex relationships. This is the foundation of regression models in data science.

Formula Recap

Given data (x_i, y_i), we build a design matrix A:

- For polynomial fit of degree d:

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^d \\ 1 & x_2 & x_2^2 & \dots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^d \end{bmatrix}$$

Then solve least squares:

$$\hat{c} = \arg\min_c \|Ac - y\|^2$$

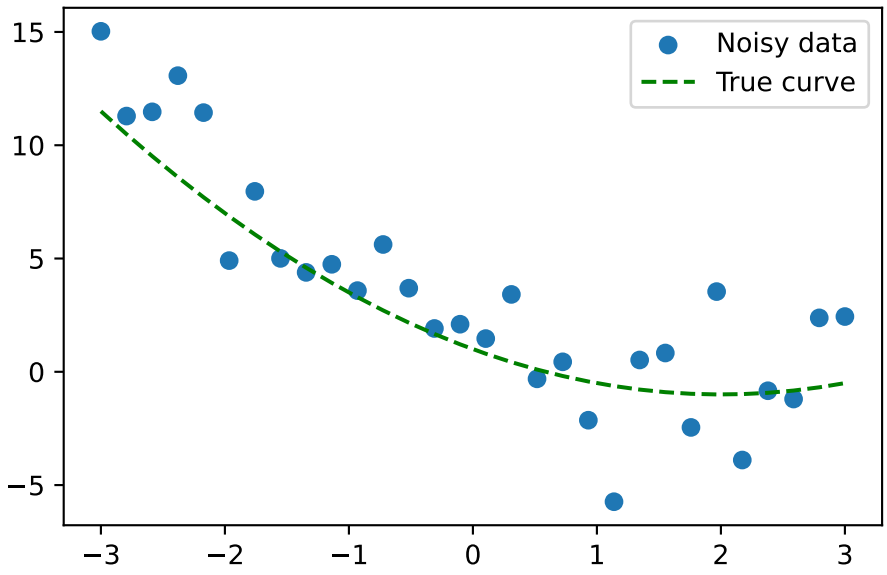
Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Generate noisy quadratic data

```
np.random.seed(0)
x = np.linspace(-3, 3, 30)
y_true = 1 - 2*x + 0.5*x**2
y_noisy = y_true + np.random.normal(scale=2.0, size=x.shape)

plt.scatter(x, y_noisy, label="Noisy data")
plt.plot(x, y_true, "g--", label="True curve")
plt.legend()
plt.show()
```



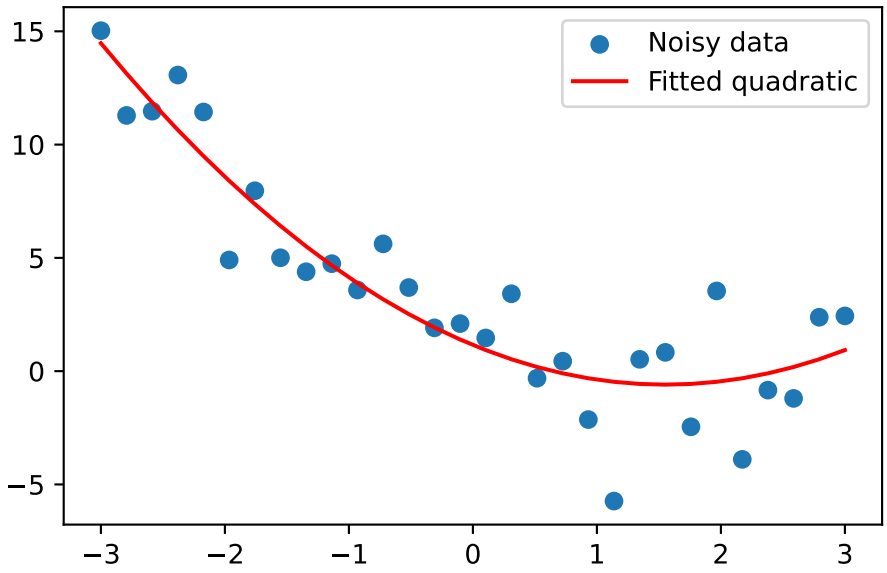
2. Build polynomial design matrix (degree 2)

```
A = np.column_stack([np.ones_like(x), x, x**2])
coeffs, *_ = np.linalg.lstsq(A, y_noisy, rcond=None)
print("Fitted coefficients:", coeffs)
```

```
Fitted coefficients: [ 1.15666306 -2.25753954  0.72733812]
```

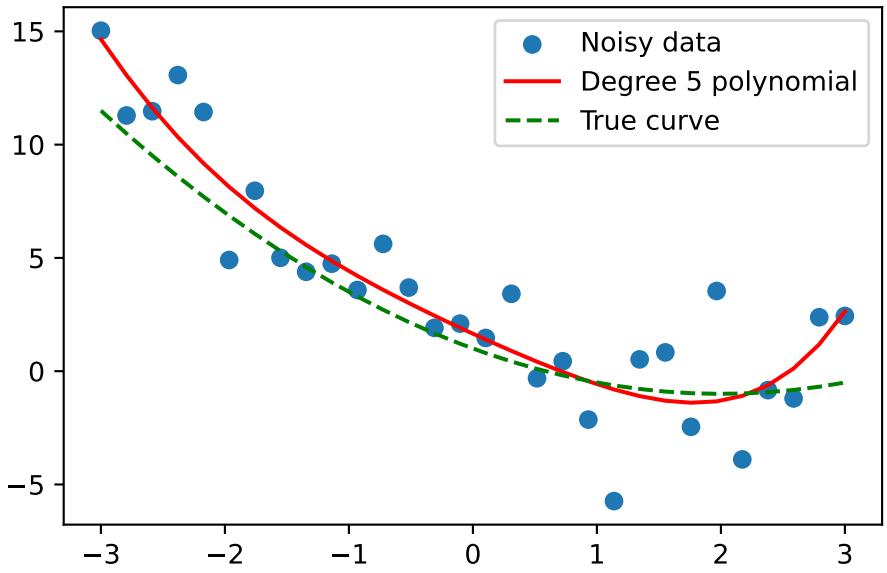
3. Plot fitted polynomial

```
y_fit = A @ coeffs
plt.scatter(x, y_noisy, label="Noisy data")
plt.plot(x, y_fit, "r-", label="Fitted quadratic")
plt.legend()
plt.show()
```



4. Higher-degree fit (overfitting demonstration)

```
A_high = np.column_stack([x**i for i in range(6)])  # degree 5
coeffs_high, *_ = np.linalg.lstsq(A_high, y_noisy, rcond=None)

y_fit_high = A_high @ coeffs_high
plt.scatter(x, y_noisy, label="Noisy data")
plt.plot(x, y_fit_high, "r-", label="Degree 5 polynomial")
plt.plot(x, y_true, "g--", label="True curve")
plt.legend()
plt.show()
```



5. Multifeature regression example

Suppose we predict y from features $[x, x^2, \sin(x)]$:

```
A_multi = np.column_stack([np.ones_like(x), x, x**2, np.sin(x)])
coeffs_multi, *_ = np.linalg.lstsq(A_multi, y_noisy, rcond=None)
print("Multi-feature coefficients:", coeffs_multi)
```

```
Multi-feature coefficients: [ 1.15666306 -2.0492999   0.72733812 -0.65902274]
```

Try It Yourself

1. Fit degree 3, 4, 5 polynomials to the same data. Watch how the curve changes.
2. Add features like $\cos(x)$ or $\exp(x)$ - does the fit improve?
3. Compare training error (fit to noisy data) vs error on new test points.

The Takeaway

- Least squares can fit polynomials and arbitrary feature combinations.
- The design matrix encodes how input variables transform into features.
- This is the basis of regression, curve fitting, and many machine learning models.

Chapter 9. SVD, PCA, and Conditioning

81. Singular Values and SVD (Universal Factorization)

The Singular Value Decomposition (SVD) is one of the most powerful results in linear algebra. It says any $m \times n$ matrix A can be factored as:

$$A = U\Sigma V^T$$

- U: orthogonal $m \times m$ matrix (left singular vectors)
- Σ: diagonal $m \times n$ matrix with nonnegative numbers (singular values)
- V: orthogonal $n \times n$ matrix (right singular vectors)

Singular values are always nonnegative and sorted $\sigma_1 \geq \sigma_2 \geq ....$

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Compute SVD of a matrix

```
A = np.array([[3,1,1],
              [-1,3,1]])

U, S, Vt = np.linalg.svd(A, full_matrices=True)

print("U:\n", U)
print("Singular values:", S)
print("V^T:\n", Vt)
```

```
U:
 [[-0.70710678 -0.70710678]
 [-0.70710678  0.70710678]]
Singular values: [3.46410162 3.16227766]
V^T:
 [[-4.08248290e-01 -8.16496581e-01 -4.08248290e-01]
 [-8.94427191e-01  4.47213595e-01  5.26260748e-16]
 [-1.82574186e-01 -3.65148372e-01  9.12870929e-01]]
```

- `U`: orthogonal basis in input space.
- `S`: singular values (as a 1D array).
- `V^T`: orthogonal basis in output space.

2. Reconstruct A from decomposition

```
Sigma = np.zeros((U.shape[1], Vt.shape[0]))
Sigma[:len(S), :len(S)] = np.diag(S)

A_reconstructed = U @ Sigma @ Vt
print("Reconstruction error:", np.linalg.norm(A - A_reconstructed))
```

```
Reconstruction error: 1.709166621382058e-15
```

The error should be near zero.

3. Rank from SVD

Number of nonzero singular values = rank of A.

```
rank = np.sum(S > 1e-10)
print("Rank of A:", rank)
```

```
Rank of A: 2
```

4. Geometry: effect of A

SVD says:

1. V rotates input space.
2. Σ scales along orthogonal directions (by singular values).
3. U rotates to output space.

This explains why SVD works for any matrix (not just square ones).

5. Low-rank approximation preview

Keep only the top singular value(s) $\rightarrow$ best approximation of A.

```
k = 1
A_approx = np.outer(U[:,0], Vt[0]) * S[0]
print("Rank-1 approximation:\n", A_approx)
```

```
Rank-1 approximation:
 [[1. 2. 1.]
 [1. 2. 1.]]
```

Try It Yourself

1. Compute SVD for a random 5×3 matrix. Check if U and V are orthogonal.
2. Compare singular values of a diagonal matrix vs a rotation matrix.
3. Zero out small singular values and see how much of A is preserved.

The Takeaway

- SVD factorizes any matrix into rotations and scalings.
- Singular values reveal rank and strength of directions.
- It's the universal tool of numerical linear algebra: the backbone of PCA, compression, and stability analysis.

82. Geometry of SVD (Rotations + Stretching)

The Singular Value Decomposition (SVD) has a beautiful geometric interpretation: every matrix is just a combination of two rotations (or reflections) and a stretching.

For $A = U\Sigma V^T$:

1. V^T: rotates (or reflects) the input space.
2. Σ: stretches space along orthogonal axes by singular values σ_i.
3. U: rotates (or reflects) the result into the output space.

This turns any linear transformation into a rotation → stretching → rotation pipeline.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Make a 2D matrix

```
A = np.array([[2, 1],
              [1, 3]])
```

2. Apply SVD

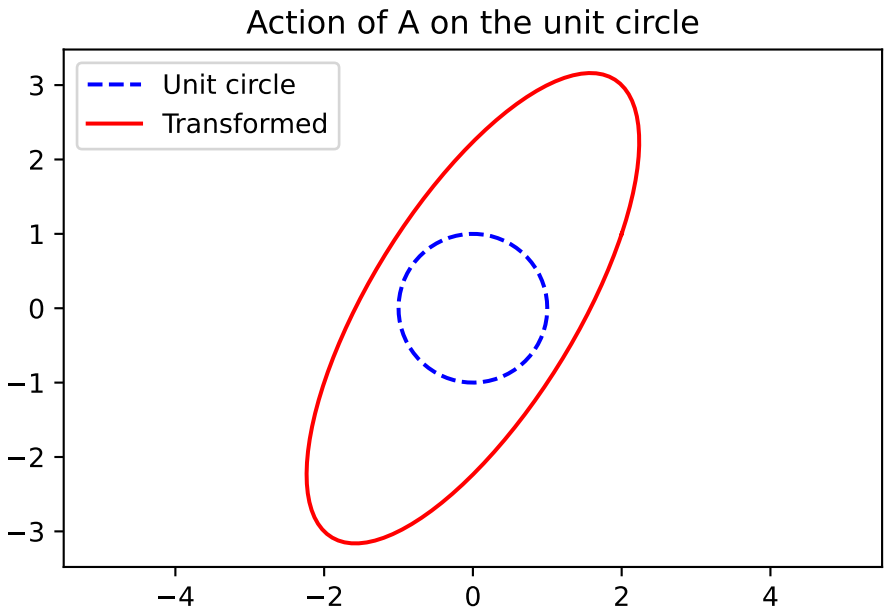
```
U, S, Vt = np.linalg.svd(A)

print("U:\n", U)
print("Singular values:", S)
print("V^T:\n", Vt)
```

```
U:
 [[-0.52573111 -0.85065081]
 [-0.85065081  0.52573111]]
Singular values: [3.61803399 1.38196601]
V^T:
 [[-0.52573111 -0.85065081]
 [-0.85065081  0.52573111]]
```

3. Visualize effect on the unit circle

The unit circle is often used to visualize linear transformations.

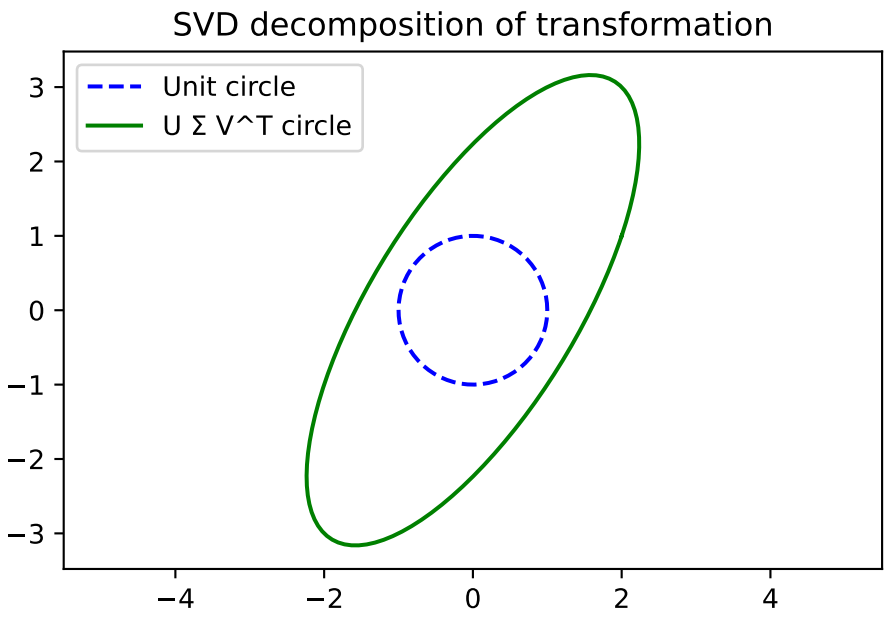
```
theta = np.linspace(0, 2*np.pi, 200)
circle = np.vstack((np.cos(theta), np.sin(theta)))

transformed = A @ circle

plt.plot(circle[0], circle[1], 'b--', label="Unit circle")
plt.plot(transformed[0], transformed[1], 'r-', label="Transformed")
plt.axis("equal")
plt.legend()
plt.title("Action of A on the unit circle")
plt.show()
```



The circle becomes an ellipse. Its axes align with the singular vectors, and its radii are the singular values.

4. Compare with decomposition steps

```
# Apply V^T
step1 = Vt @ circle
# Apply Σ
Sigma = np.diag(S)
step2 = Sigma @ step1
# Apply U
step3 = U @ step2

plt.plot(circle[0], circle[1], 'b--', label="Unit circle")
plt.plot(step3[0], step3[1], 'g-', label="U Σ V^T circle")
plt.axis("equal")
plt.legend()
plt.title("SVD decomposition of transformation")
plt.show()
```



Both transformed shapes match → confirms SVD's geometric picture.

Try It Yourself

1. Change A to a pure shear, like `[[1,2],[0,1]]`. How does the ellipse look?
2. Try a diagonal matrix, like `[[3,0],[0,1]]`. Do the singular vectors match the coordinate axes?
3. Scale the input circle to a square and see if geometry still works.

The Takeaway

- SVD = rotate → stretch → rotate.
- The unit circle becomes an ellipse: axes = singular vectors, radii = singular values.
- This geometric lens makes SVD intuitive and explains why it's so widely used in data, graphics, and signal processing.

83. Relation to Eigen-Decompositions (ATA and AAT)

Singular values and eigenvalues are closely connected. While eigen-decomposition applies only to square matrices, SVD works for any rectangular matrix. The bridge between them is:

$$A^T A v = \sigma^2 v \quad \text{and} \quad A A^T u = \sigma^2 u$$

- v: right singular vector (from eigenvectors of A^TA)
- u: left singular vector (from eigenvectors of AA^T)
- σ: singular values (square roots of eigenvalues of A^TA or AA^T)

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Define a rectangular matrix

```
A = np.array([[2, 0],
              [1, 1],
              [0, 1]])  # shape 3x2
```

2. Compute SVD directly

```
U, S, Vt = np.linalg.svd(A)
print("Singular values:", S)
```

```
Singular values: [2.30277564 1.30277564]
```

3. Compare with eigenvalues of A^TA

```
ATA = A.T @ A
eigvals, eigvecs = np.linalg.eig(ATA)

print("Eigenvalues of A^T A:", eigvals)
print("Square roots (sorted):", np.sqrt(np.sort(eigvals)[::-1]))
```

```
Eigenvalues of A^T A: [5.30277564 1.69722436]
Square roots (sorted): [2.30277564 1.30277564]
```

Notice: singular values from SVD = square roots of eigenvalues of A^TA.

4. Compare with eigenvalues of AA^T

```
AAT = A @ A.T
eigvals2, eigvecs2 = np.linalg.eig(AAT)

print("Eigenvalues of A A^T:", eigvals2)
print("Square roots:", np.sqrt(np.sort(eigvals2)[::-1]))
```

```
Eigenvalues of A A^T: [ 5.30277564e+00  1.69722436e+00 -2.15148422e-17]
Square roots: [2.30277564 1.30277564        nan]
```

```
/tmp/ipykernel_2715/436251338.py:5: RuntimeWarning: invalid value encountered in sqrt
  print("Square roots:", np.sqrt(np.sort(eigvals2)[::-1]))
```

They match too → confirming the relationship.

5. Verify singular vectors

- Right singular vectors (V) = eigenvectors of A^TA.
- Left singular vectors (U) = eigenvectors of AA^T.

```
print("Right singular vectors (V):\n", Vt.T)
print("Eigenvectors of A^T A:\n", eigvecs)

print("Left singular vectors (U):\n", U)
print("Eigenvectors of A A^T:\n", eigvecs2)
```

```
Right singular vectors (V):
 [[-0.95709203  0.28978415]
 [-0.28978415 -0.95709203]]
Eigenvectors of A^T A:
 [[ 0.95709203 -0.28978415]
 [ 0.28978415  0.95709203]]
Left singular vectors (U):
 [[-0.83125078  0.44487192  0.33333333]
 [-0.54146663 -0.51222011 -0.66666667]
 [-0.12584124 -0.73465607  0.66666667]]
Eigenvectors of A A^T:
 [[-0.83125078  0.44487192  0.33333333]
 [-0.54146663 -0.51222011 -0.66666667]
 [-0.12584124 -0.73465607  0.66666667]]
```

Try It Yourself

1. Try a square symmetric matrix and compare SVD with eigen-decomposition. Do they match?
2. For a tall vs wide rectangular matrix, check whether U and V differ.
3. Compute eigenvalues manually with `np.linalg.eig` for a random A and confirm singular values.

The Takeaway

- Singular values = square roots of eigenvalues of A^TA (or AA^T).
- Right singular vectors = eigenvectors of A^TA.
- Left singular vectors = eigenvectors of AA^T.
- SVD generalizes eigen-decomposition to all matrices, rectangular or square.

84. Low-Rank Approximation (Best Small Models)

One of the most useful applications of SVD is low-rank approximation: compressing a large matrix into a smaller one while keeping most of the important information.

The Eckart–Young theorem says: If $A = U\Sigma V^T$, then the best rank-k approximation (in least-squares sense) is:

$$A_k = U_k \Sigma_k V_k^T$$

where we keep only the top k singular values (and corresponding vectors).

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Create a matrix with hidden low-rank structure

```
np.random.seed(0)
U = np.random.randn(50, 5)   # 50 x 5
V = np.random.randn(5, 30)   # 5 x 30
A = U @ V  # true rank  5
```

2. Full SVD

```
U, S, Vt = np.linalg.svd(A, full_matrices=False)
print("Singular values:", S[:10])
```

```
Singular values: [4.90672194e+01 4.05935057e+01 3.39228766e+01 3.07883338e+01
 2.29261740e+01 3.97150036e-15 3.97150036e-15 3.97150036e-15
 3.97150036e-15 3.97150036e-15]
```

Only the first ~5 should be large; the rest close to zero.

3. Build rank-1 approximation

```
k = 1
A1 = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]
error1 = np.linalg.norm(A - A1)
print("Rank-1 approximation error:", error1)
```

```
Rank-1 approximation error: 65.36149641872868
```

4. Rank-5 approximation (should be almost exact)

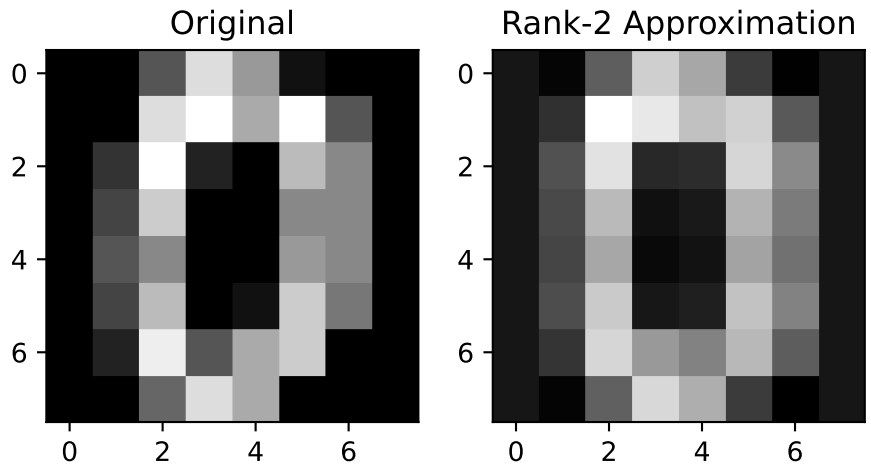
```
k = 5
A5 = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]
error5 = np.linalg.norm(A - A5)
print("Rank-5 approximation error:", error5)
```

```
Rank-5 approximation error: 6.37596738696176e-14
```

5. Visual comparison (image compression demo)

Let's see it on an image.

```
from sklearn.datasets import load_digits
digits = load_digits()
img = digits.images[0]  # 8x8 grayscale digit

U, S, Vt = np.linalg.svd(img, full_matrices=False)

# Keep only top 2 singular values
k = 2
img2 = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]

plt.subplot(1,2,1)
plt.imshow(img, cmap="gray")
plt.title("Original")

plt.subplot(1,2,2)
plt.imshow(img2, cmap="gray")
plt.title("Rank-2 Approximation")
plt.show()
```



Even with just 2 singular values, the digit shape is recognizable.

Try It Yourself

1. Vary k in the image example (1, 2, 5, 10). How much detail do you keep?
2. Compare the approximation error $\|A - A_k\|$ as k increases.
3. Apply low-rank approximation to random noisy data. Does it denoise?

The Takeaway

- SVD gives the best possible low-rank approximation in terms of error.
- By truncating singular values, you compress data while keeping its essential structure.
- This is the backbone of image compression, recommender systems, and dimensionality reduction.

85. Principal Component Analysis (Variance and Directions)

Principal Component Analysis (PCA) is one of the most important applications of SVD. It finds the directions (principal components) where data varies the most, and projects the data onto them to reduce dimensionality while preserving as much information as possible.

Mathematically:

1. Center the data (subtract the mean).
2. Compute covariance matrix $C = \frac{1}{n} X^T X$.
3. Eigenvectors of C = principal directions.
4. Eigenvalues = variance explained.
5. Equivalently: PCA = SVD of centered data matrix.

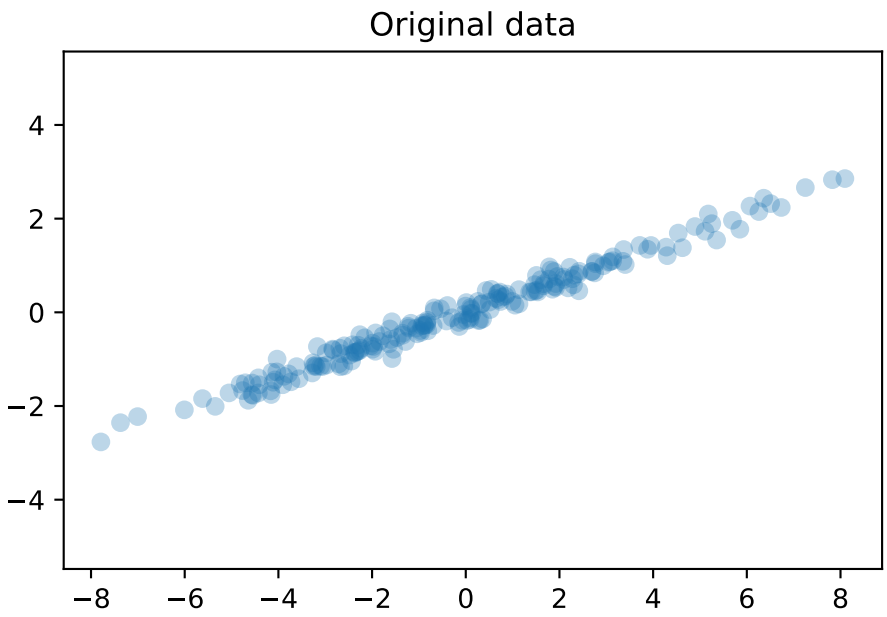
Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
```

Step-by-Step Code Walkthrough

1. Generate synthetic 2D data

```
np.random.seed(0)
X = np.random.randn(200, 2) @ np.array([[3,1],[1,0.5]])  # stretched cloud

plt.scatter(X[:,0], X[:,1], alpha=0.3)
plt.title("Original data")
plt.axis("equal")
plt.show()
```



2. Center the data

```
X_centered = X - X.mean(axis=0)
```

3. Compute SVD

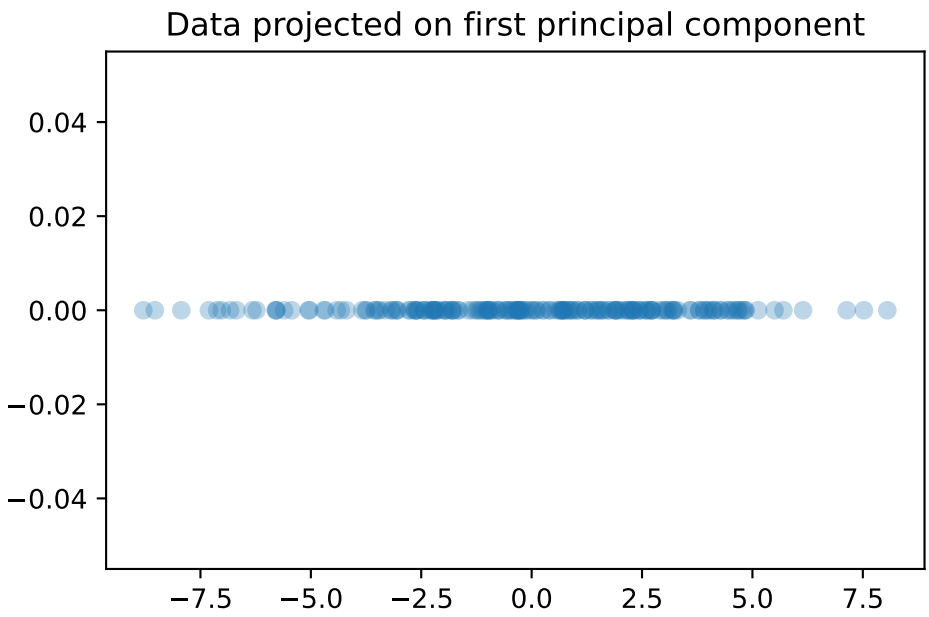
```
U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)
print("Principal directions (V):\n", Vt)
```

```
Principal directions (V):
 [[-0.94430098 -0.32908307]
 [ 0.32908307 -0.94430098]]
```

Rows of `Vt` are the principal components.

4. Project data onto first component

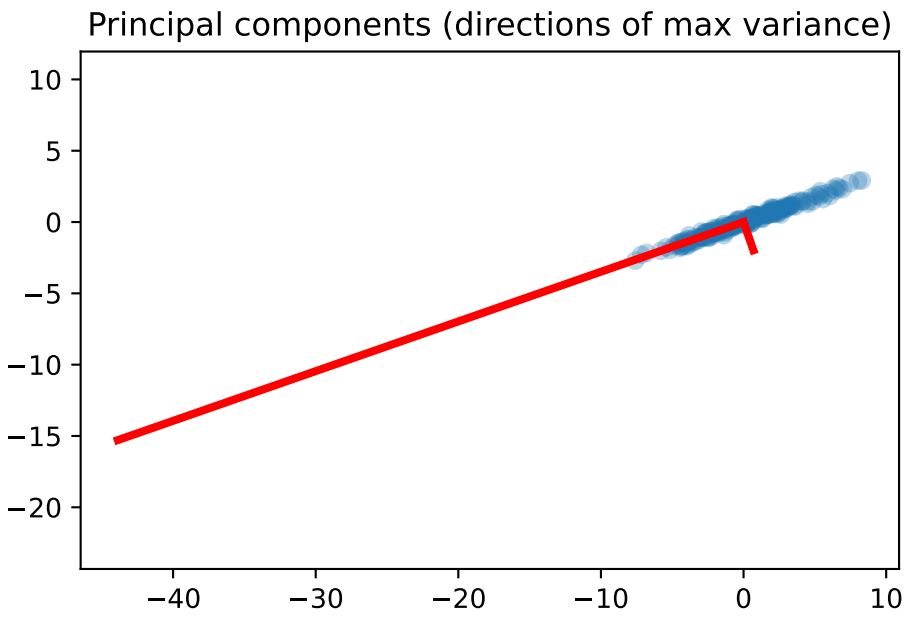
```
X_pca1 = X_centered @ Vt.T[:,0]

plt.scatter(X_pca1, np.zeros_like(X_pca1), alpha=0.3)
plt.title("Data projected on first principal component")
plt.show()
```



This collapses data into 1D, keeping the most variance.

5. Visualize principal axes

```
plt.scatter(X_centered[:,0], X_centered[:,1], alpha=0.3)
for length, vector in zip(S, Vt):
    plt.plot([0, vector[0]*length], [0, vector[1]*length], 'r-', linewidth=3)
plt.title("Principal components (directions of max variance)")
plt.axis("equal")
plt.show()
```



The red arrows show where the data spreads most.

6. PCA on real data (digits)

```
digits = load_digits()
X = digits.data  # 1797 samples, 64 features
X_centered = X - X.mean(axis=0)

U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)

explained_variance = (S**2) / np.sum(S**2)
print("Explained variance ratio (first 5):", explained_variance[:5])
```

```
Explained variance ratio (first 5): [0.14890594 0.13618771 0.11794594 0.08409979 0.05782415]
```

Try It Yourself

1. Reduce digits dataset to 2D using the top 2 components and plot. Do digit clusters separate?
2. Compare explained variance ratio for top 10 components.
3. Add noise to data and check if PCA filters it out when projecting to fewer dimensions.

The Takeaway

- PCA finds directions of maximum variance using SVD.
- By projecting onto top components, you compress data with minimal information loss.
- PCA is the backbone of dimensionality reduction, visualization, and preprocessing in machine learning.

86. Pseudoinverse (Moore–Penrose) and Solving Ill-Posed Systems

The Moore–Penrose pseudoinverse A^+ generalizes the inverse of a matrix. It allows solving systems $Ax = b$ even when:

- A is not square, or
- A is singular (non-invertible).

The solution given by the pseudoinverse is the least-squares solution with minimum norm:

$$x = A^+ b$$

If $A = U\Sigma V^T$, then:

$$A^+ = V\Sigma^+ U^T$$

where Σ^+ is obtained by taking reciprocals of nonzero singular values.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Solve an overdetermined system (more equations than unknowns)

```
A = np.array([[1,1],
              [1,2],
              [1,3]])  # 3x2 system
b = np.array([1,2,2])

x_ls, *_ = np.linalg.lstsq(A, b, rcond=None)
print("Least-squares solution:", x_ls)
```

```
Least-squares solution: [0.66666667 0.5       ]
```

2. Compute with pseudoinverse

```
A_pinv = np.linalg.pinv(A)
x_pinv = A_pinv @ b
print("Pseudoinverse solution:", x_pinv)
```

```
Pseudoinverse solution: [0.66666667 0.5       ]
```

Both match → pseudoinverse gives least-squares solution.

3. Solve an underdetermined system (fewer equations than unknowns)

```
A = np.array([[1,2,3]])  # 1x3
b = np.array([1])

x_pinv = np.linalg.pinv(A) @ b
print("Minimum norm solution:", x_pinv)
```

```
Minimum norm solution: [0.07142857 0.14285714 0.21428571]
```

Here, infinitely many solutions exist. The pseudoinverse picks the one with smallest norm.

4. Compare with singular matrix

```
A = np.array([[1,2],
              [2,4]])  # rank deficient
b = np.array([1,2])

x_pinv = np.linalg.pinv(A) @ b
print("Solution with pseudoinverse:", x_pinv)
```

```
Solution with pseudoinverse: [0.2 0.4]
```

Even when A is singular, pseudoinverse provides a solution.

5. Manual pseudoinverse via SVD

```
A = np.array([[1,2],
              [3,4]])
U, S, Vt = np.linalg.svd(A)
S_inv = np.zeros((Vt.shape[0], U.shape[0]))
for i in range(len(S)):
    if S[i] > 1e-10:
        S_inv[i,i] = 1/S[i]

A_pinv_manual = Vt.T @ S_inv @ U.T
print("Manual pseudoinverse:\n", A_pinv_manual)
print("NumPy pseudoinverse:\n", np.linalg.pinv(A))
```

```
Manual pseudoinverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
NumPy pseudoinverse:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

They match.

Try It Yourself

1. Create an overdetermined system with noise and see how pseudoinverse smooths the solution.
2. Compare pseudoinverse with direct inverse (`np.linalg.inv`) on a square nonsingular matrix.
3. Zero out small singular values manually and see how solution changes.

The Takeaway

- The pseudoinverse solves any linear system, square or not.
- It provides the least-squares solution in overdetermined cases and the minimum-norm solution in underdetermined cases.
- Built on SVD, it is a cornerstone of regression, optimization, and numerical methods.

87. Conditioning and Sensitivity (How Errors Amplify)

Conditioning tells us how sensitive a system is to small changes. For a linear system $Ax = b$:

- If A is well-conditioned, small changes in b or $A \rightarrow$ small changes in x.
- If A is ill-conditioned, tiny changes can cause huge swings in x.

The condition number is defined as:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|$$

For SVD:

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

where $\sigma_{\max}$ and $\sigma_{\min}$ are the largest and smallest singular values.

- Large $\kappa(A) \rightarrow$ unstable system.
- Small $\kappa(A) \rightarrow$ stable system.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Well-conditioned system

```
A = np.array([[2,0],
              [0,1]])
b = np.array([1,1])

x = np.linalg.solve(A, b)
cond = np.linalg.cond(A)
print("Solution:", x)
print("Condition number:", cond)
```

```
Solution: [0.5 1. ]
Condition number: 2.0
```

Condition number = ratio of singular values → moderate size.

2. Ill-conditioned system

```
A = np.array([[1, 1.0001],
              [1, 1.0000]])
b = np.array([2,2])

x = np.linalg.lstsq(A, b, rcond=None)[0]
cond = np.linalg.cond(A)
print("Solution:", x)
print("Condition number:", cond)
```

```
Solution: [ 2.00000000e+00 -7.20128227e-17]
Condition number: 40002.0000750375
```

Condition number is very large → instability.

3. Perturb the right-hand side

```
b2 = np.array([2, 2.001])  # tiny change
x2 = np.linalg.lstsq(A, b2, rcond=None)[0]
print("Solution after tiny change:", x2)
```

```
Solution after tiny change: [ 12.001 -10.   ]
```

The solution changes drastically → shows sensitivity.

4. Relation to singular values

```
U, S, Vt = np.linalg.svd(A)
print("Singular values:", S)
print("Condition number (SVD):", S[0]/S[-1])
```

```
Singular values: [2.000050e+00 4.999875e-05]
Condition number (SVD): 40002.0000750375
```

5. Scaling experiment

```
for scale in [1,1e-2,1e-4,1e-6]:
    A = np.array([[1,0],[0,scale]])
    print(f"Scale={scale}, condition number={np.linalg.cond(A)}")
```

```
Scale=1, condition number=1.0
Scale=0.01, condition number=100.0
Scale=0.0001, condition number=10000.0
Scale=1e-06, condition number=1000000.0
```

As scale shrinks, condition number explodes.

Try It Yourself

1. Generate random matrices and compute their condition numbers. Which are stable?
2. Compare condition numbers of Hilbert matrices (notoriously ill-conditioned).
3. Explore how rounding errors grow with high condition numbers.

The Takeaway

- Condition number = measure of problem sensitivity.
- $\kappa(A) = \sigma_{\max}/\sigma_{\min}$.
- Ill-conditioned problems amplify errors and are numerically unstable → why scaling, regularization, and good formulations matter.

88. Matrix Norms and Singular Values (Measuring Size Properly)

Matrix norms measure the size or strength of a matrix. They extend the idea of vector length to matrices. Norms are crucial for analyzing stability, error growth, and performance of algorithms.

Some important norms:

- Frobenius norm:

$$\|A\|_F = \sqrt{\sum_{i,j} |a_{ij}|^2}$$

Equivalent to treating the matrix as a big vector.

- Spectral norm (operator 2-norm):

$$\|A\|_2 = \sigma_{\max}$$

The largest singular value - tells how much A can stretch a vector.

- 1-norm: maximum absolute column sum.
- ∞-norm: maximum absolute row sum.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Build a test matrix

```
A = np.array([[1, -2, 3],
              [0,  4, 5],
              [-1, 2, 1]])
```

2. Compute different norms

```
fro = np.linalg.norm(A, 'fro')
spec = np.linalg.norm(A, 2)
one_norm = np.linalg.norm(A, 1)
inf_norm = np.linalg.norm(A, np.inf)

print("Frobenius norm:", fro)
print("Spectral norm:", spec)
print("1-norm:", one_norm)
print("Infinity norm:", inf_norm)
```

```
Frobenius norm: 7.810249675906654
Spectral norm: 6.813953458914003
1-norm: 9.0
Infinity norm: 9.0
```

3. Compare spectral norm with largest singular value

```
U, S, Vt = np.linalg.svd(A)
print("Largest singular value:", S[0])
print("Spectral norm:", spec)
```

```
Largest singular value: 6.8139534589140025
Spectral norm: 6.813953458914003
```

They match → spectral norm = largest singular value.

4. Frobenius norm from singular values

$$\|A\|_F = \sqrt{\sigma_1^2 + \sigma_2^2 + ...}$$

```
fro_from_svd = np.sqrt(np.sum(S**2))
print("Frobenius norm (from SVD):", fro_from_svd)
```

```
Frobenius norm (from SVD): 7.810249675906653
```

5. Stretching effect demonstration

Pick a random vector and see how much it grows:

```
x = np.random.randn(3)
stretch = np.linalg.norm(A @ x) / np.linalg.norm(x)
print("Stretch factor:", stretch)
print("Spectral norm (max possible stretch):", spec)
```

```
Stretch factor: 2.7537463268177698
Spectral norm (max possible stretch): 6.813953458914003
```

The stretch spectral norm, always.

Try It Yourself

1. Compare norms for diagonal matrices - do they match the largest diagonal entry?
2. Generate random matrices and see how norms differ.
3. Compute Frobenius vs spectral norm for a rank-1 matrix.

The Takeaway

- Frobenius norm = overall energy of the matrix.
- Spectral norm = maximum stretching power (largest singular value).
- Other norms (1-norm, ∞-norm) capture row/column dominance.
- Singular values unify all these views of "matrix size."

89. Regularization (Ridge/Tikhonov to Tame Instability)

When solving $Ax = b$, if A is ill-conditioned (large condition number), small errors in data can cause huge errors in the solution. Regularization stabilizes the problem by adding a penalty term that discourages extreme solutions.

The most common form: ridge regression (a.k.a. Tikhonov regularization):

$$x_\lambda = \arg\min_x \|Ax - b\|^2 + \lambda\|x\|^2$$

Closed form:

$$x_\lambda = (A^T A + \lambda I)^{-1} A^T b$$

Here $\lambda > 0$ controls the amount of regularization:

- Small λ: solution close to least-squares.
- Large λ: smaller coefficients, more stability.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Build an ill-conditioned system

```
A = np.array([[1, 1.001],
              [1, 0.999]])
b = np.array([2, 2])
```

2. Solve without regularization

```
x_ls, *_ = np.linalg.lstsq(A, b, rcond=None)
print("Least squares solution:", x_ls)
```

```
Least squares solution: [2.0000000e+00 4.6705917e-17]
```

The result may be unstable.

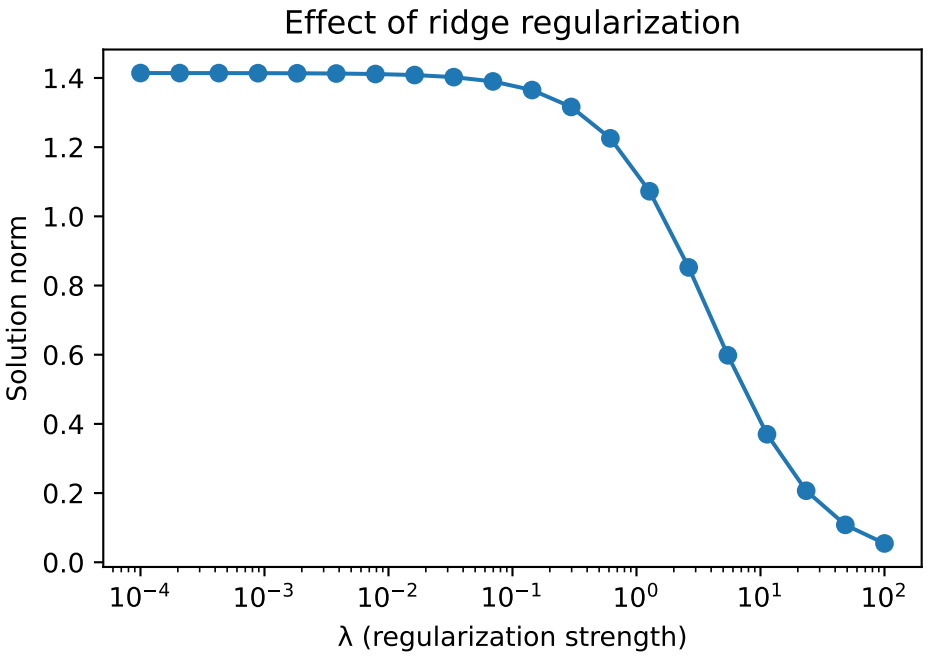
3. Apply ridge regularization

```
lam = 0.1
x_ridge = np.linalg.inv(A.T @ A + lam*np.eye(2)) @ A.T @ b
print("Ridge solution ( =0.1):", x_ridge)
```

```
Ridge solution ( =0.1): [0.97561927 0.97559976]
```

4. Compare effect of different

```
lambdas = np.logspace(-4, 2, 20)
solutions = []
for lam in lambdas:
    x_reg = np.linalg.inv(A.T @ A + lam*np.eye(2)) @ A.T @ b
    solutions.append(np.linalg.norm(x_reg))

plt.semilogx(lambdas, solutions, 'o-')
plt.xlabel("  (regularization strength)")
plt.ylabel("Solution norm")
plt.title("Effect of ridge regularization")
plt.show()
```



As λ increases, the solution becomes smaller and more stable.

5. Connection to SVD

If $A = U\Sigma V^T$:

$$x_\lambda = \sum_i \frac{\sigma_i}{\sigma_i^2 + \lambda}(u_i^T b)v_i$$

Small singular values (causing instability) get damped by $\frac{\sigma_i}{\sigma_i^2+\lambda}$.

Try It Yourself

1. Experiment with larger and smaller λ. What happens to the solution?
2. Add random noise to b. Compare least-squares vs ridge stability.
3. Plot how each coefficient changes with .

The Takeaway

- Regularization controls instability in ill-conditioned problems.
- Ridge regression balances fit vs. stability using .
- In SVD terms, regularization damps small singular values that cause wild solutions.

90. Rank-Revealing QR and Practical Diagnostics (What Rank Really Is)

In practice, we often need to determine the numerical rank of a matrix - not just the theoretical rank, but how many directions carry meaningful information beyond round-off errors or noise. A useful tool for this is the Rank-Revealing QR (RRQR) factorization.

For a matrix A:

$$AP = QR$$

- Q: orthogonal matrix
- R: upper triangular matrix
- P: column permutation matrix

By reordering columns smartly, the diagonal of R reveals which directions are significant.

Set Up Your Lab

```
import numpy as np
from scipy.linalg import qr
```

Step-by-Step Code Walkthrough

1. Build a nearly rank-deficient matrix

```
A = np.array([[1, 2, 3],
              [2, 4.001, 6],
              [3, 6, 9.001]])
print("Rank (theoretical):", np.linalg.matrix_rank(A))
```

```
Rank (theoretical): 3
```

This matrix is almost rank 2 but with small perturbations.

2. QR with column pivoting

```
Q, R, P = qr(A, pivoting=True)
print("R:\n", R)
print("Column permutation:", P)
```

```
R:
 [[-1.12257740e+01 -7.48384925e+00 -3.74165738e+00]
 [ 0.00000000e+00 -1.20185042e-03 -1.84886859e-04]
 [ 0.00000000e+00  0.00000000e+00 -7.41196374e-05]]
Column permutation: [2 1 0]
```

The diagonal entries of R decrease rapidly → numerical rank is determined where they become tiny.

3. Compare with SVD

```
U, S, Vt = np.linalg.svd(A)
print("Singular values:", S)
```

```
Singular values: [1.40009286e+01 1.00000000e-03 7.14238341e-05]
```

The singular values tell the same story: one is very small → effective rank 2.

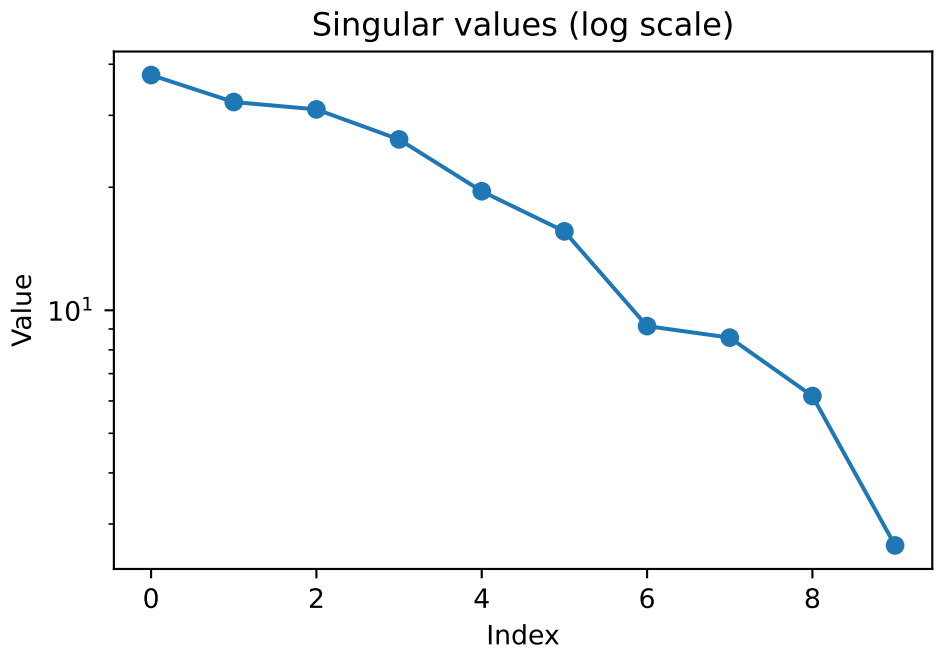
4. Thresholding for rank

```
tol = 1e-3
rank_est = np.sum(S > tol)
print("Estimated rank:", rank_est)
```

```
Estimated rank: 2
```

5. Diagnostics on a noisy matrix

```
np.random.seed(0)
B = np.random.randn(50, 10) @ np.random.randn(10, 10)  # rank  10
B[:, -1] += 1e-6 * np.random.randn(50)  # tiny noise

U, S, Vt = np.linalg.svd(B)
plt.semilogy(S, 'o-')
plt.title("Singular values (log scale)")
plt.xlabel("Index")
plt.ylabel("Value")
plt.show()
```



The drop in singular values shows effective rank.

Try It Yourself

1. Change the perturbation in A from 0.001 to 0.000001. Does the numerical rank change?
2. Test QR with pivoting on random rectangular matrices.
3. Compare rank estimates from QR vs SVD for large noisy matrices.

The Takeaway

- Rank-revealing QR is a practical tool to detect effective rank in real-world data.
- SVD gives the most precise picture (singular values), but QR with pivoting is faster.
- Understanding numerical rank is crucial for diagnostics, stability, and model complexity control.

Chapter 10. Applications and computation

91. 2D/3D Geometry Pipelines (Cameras, Rotations, and Transforms)

Linear algebra powers the geometry pipelines in computer graphics and robotics.

- 2D transforms: rotation, scaling, translation.
- 3D transforms: same ideas, but with an extra dimension.
- Homogeneous coordinates let us unify all transforms (even translations) into matrix multiplications.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Rotation in 2D

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

```
theta = np.pi/4  # 45 degrees
R = np.array([[np.cos(theta), -np.sin(theta)],
              [np.sin(theta),  np.cos(theta)]])

point = np.array([1, 0])
rotated = R @ point

print("Original:", point)
print("Rotated:", rotated)
```

```
Original: [1 0]
Rotated: [0.70710678 0.70710678]
```

2. Translation using homogeneous coordinates

In 2D:

$$T(dx, dy) = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

```
T = np.array([[1,0,2],
              [0,1,1],
              [0,0,1]])

p_h = np.array([1,1,1])  # homogeneous (x=1,y=1)
translated = T @ p_h
print("Translated point:", translated)
```

```
Translated point: [3 2 1]
```

3. Combine rotation + translation

Transformations compose by multiplying matrices.

```
M = T @ np.block([[R, np.zeros((2,1))],
                  [np.zeros((1,2)), 1]])
combined = M @ p_h
print("Combined transform (rotation+translation):", combined)
```

```
Combined transform (rotation+translation): [2.         2.41421356 1.        ]
```

4. 3D rotation (around z-axis)

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
theta = np.pi/3
Rz = np.array([[np.cos(theta), -np.sin(theta), 0],
               [np.sin(theta),  np.cos(theta), 0],
               [0,              0,             1]])

point3d = np.array([1,0,0])
rotated3d = Rz @ point3d
print("3D rotated point:", rotated3d)
```

```
3D rotated point: [0.5       0.8660254 0.       ]
```

5. Camera projection (3D → 2D)

Simple pinhole model:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} f \cdot x/z \\ f \cdot y/z \end{bmatrix}$$

```
f = 1.0  # focal length
P = np.array([[f,0,0],
              [0,f,0],
              [0,0,1]])  # projection matrix

point3d = np.array([2,3,5])
p_proj = P @ point3d
p_proj = p_proj[:2] / p_proj[2]  # divide by z
print("Projected 2D point:", p_proj)
```

```
Projected 2D point: [0.4 0.6]
```

Try It Yourself

1. Rotate a square in 2D, then translate it. Plot before/after.
2. Rotate a 3D point cloud around x, y, and z axes.
3. Project a cube into 2D using the pinhole camera model.

The Takeaway

- Geometry pipelines = sequences of linear transforms.
- Homogeneous coordinates unify rotation, scaling, and translation.
- Camera projection links 3D world to 2D images - a cornerstone of graphics and vision.

92. Computer Graphics and Robotics (Homogeneous Tricks in Action)

Computer graphics and robotics both rely on homogeneous coordinates to unify rotations, translations, scalings, and projections into a single framework. With 4×4 matrices in 3D, entire transformation pipelines can be built as matrix products.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Homogeneous representation of a point

In 3D:

$$(x, y, z) \mapsto (x, y, z, 1)$$

```
p = np.array([1,2,3,1])  # homogeneous point
```

2. Define translation, rotation, and scaling matrices

- Translation by (dx, dy, dz):

```
T = np.array([[1,0,0,2],
              [0,1,0,1],
              [0,0,1,3],
              [0,0,0,1]])
```

- Scaling by factors (sx, sy, sz):

```
S = np.diag([2, 0.5, 1.5, 1])
```

- Rotation about z-axis ($\theta = 90^\circ$):

```
theta = np.pi/2
Rz = np.array([[np.cos(theta), -np.sin(theta), 0, 0],
               [np.sin(theta),  np.cos(theta), 0, 0],
               [0,              0,             1, 0],
               [0,              0,             0, 1]])
```

3. Combine transforms into a pipeline

```
M = T @ Rz @ S  # first scale, then rotate, then translate
p_transformed = M @ p
print("Transformed point:", p_transformed)
```

```
Transformed point: [1.  3.  7.5 1. ]
```

4. Robotics: forward kinematics of a 2-link arm

Each joint is a rotation + translation.

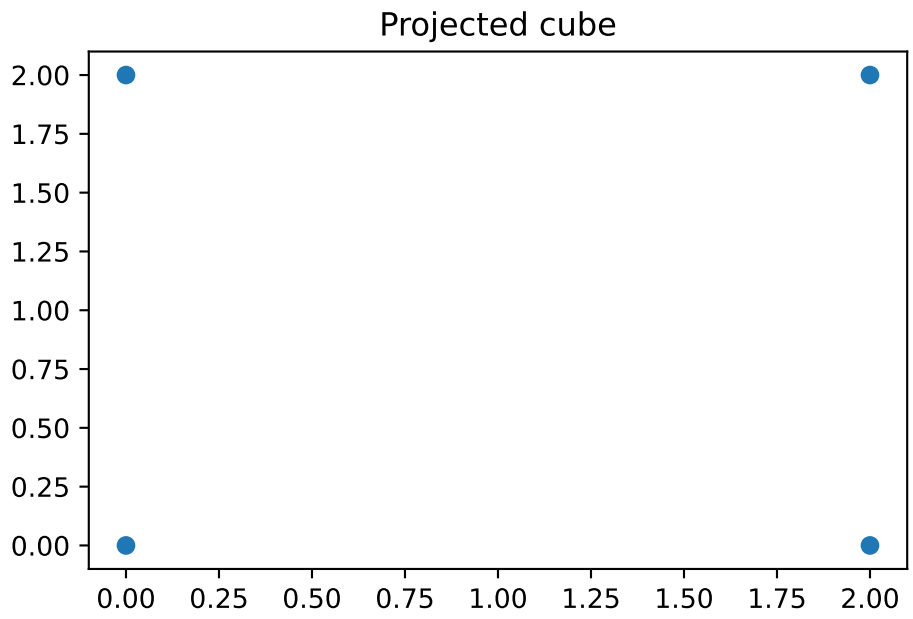
```
def link(theta, length):
    return np.array([[np.cos(theta), -np.sin(theta), 0, length*np.cos(theta)],
                     [np.sin(theta),  np.cos(theta), 0, length*np.sin(theta)],
                     [0,              0,             1, 0],
                     [0,              0,             0, 1]])

theta1, theta2 = np.pi/4, np.pi/6
L1, L2 = 2, 1.5

M1 = link(theta1, L1)
M2 = link(theta2, L2)

end_effector = M1 @ M2 @ np.array([0,0,0,1])
print("End effector position:", end_effector[:3])
```

```
End effector position: [1.80244213 2.8631023  0.        ]
```

5. Graphics: simple 3D camera projection

```
f = 2.0
P = np.array([[f,0,0,0],
              [0,f,0,0],
              [0,0,1,0]])

cube = np.array([[x,y,z,1] for x in [0,1] for y in [0,1] for z in [0,1]])
proj = (P @ cube.T).T
proj2d = proj[:,:2] / proj[:,2:3]

plt.scatter(proj2d[:,0], proj2d[:,1])
plt.title("Projected cube")
plt.show()
```

```
/tmp/ipykernel_2715/2038614107.py:8: RuntimeWarning: divide by zero encountered in divide
  proj2d = proj[:,:2] / proj[:,2:3]
/tmp/ipykernel_2715/2038614107.py:8: RuntimeWarning: invalid value encountered in divide
  proj2d = proj[:,:2] / proj[:,2:3]
```



Try It Yourself

1. Change order of transforms (`Rz @ S @ T`). How does the result differ?
2. Add a third joint to the robotic arm and compute new end-effector position.
3. Project the cube with different focal lengths f.

The Takeaway

- Homogeneous coordinates unify all transformations.
- Robotics uses this framework for forward kinematics.
- Graphics uses it for camera and projection pipelines.
- Both fields rely on the same linear algebra tricks - just applied differently.

93. Graphs, Adjacency, and Laplacians (Networks via Matrices)

Graphs can be studied with linear algebra by encoding them into matrices. Two of the most important:

- Adjacency matrix A:

$$A_{ij} = \begin{cases} 1 & \text{if edge between i and j exists} \\ 0 & \text{otherwise} \end{cases}$$

- Graph Laplacian L:

$$L = D - A$$

  where D is the degree matrix ($D_{ii} = $ number of neighbors of node i).

These matrices let us analyze connectivity, diffusion, and clustering.

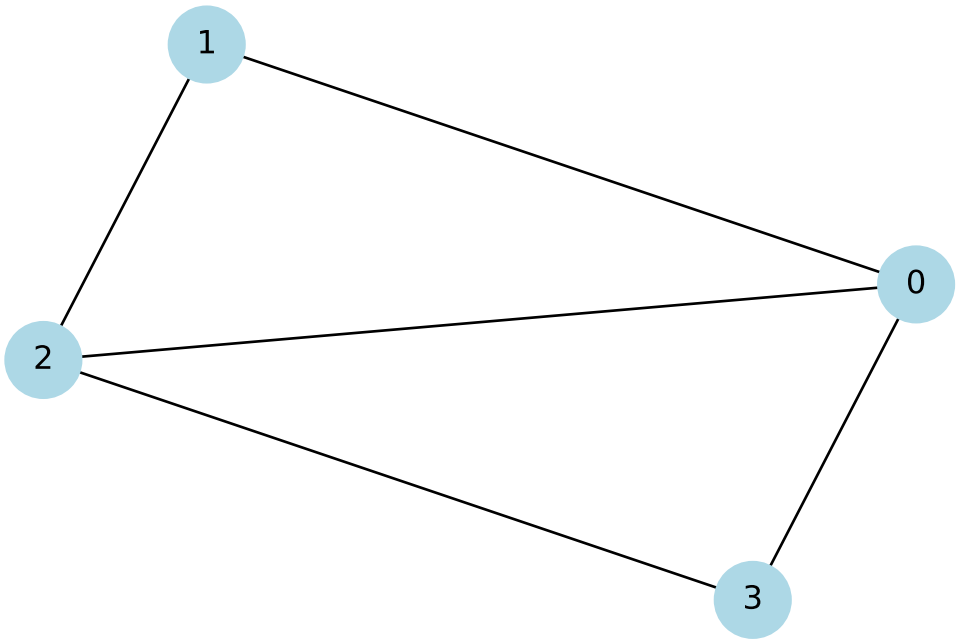
Set Up Your Lab

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Build a simple graph

```
G = nx.Graph()
G.add_edges_from([(0,1), (1,2), (2,3), (3,0), (0,2)])  # square with diagonal

nx.draw(G, with_labels=True, node_color="lightblue", node_size=800)
plt.show()
```



2. Adjacency matrix

```
A = nx.to_numpy_array(G)
print("Adjacency matrix:\n", A)
```

```
Adjacency matrix:
 [[0. 1. 1. 1.]
 [1. 0. 1. 0.]
 [1. 1. 0. 1.]
 [1. 0. 1. 0.]]
```

3. Degree and Laplacian matrices

```
D = np.diag(A.sum(axis=1))
L = D - A
print("Degree matrix:\n", D)
print("Graph Laplacian:\n", L)
```

```
Degree matrix:
 [[3. 0. 0. 0.]
 [0. 2. 0. 0.]
```

```
 [0. 0. 3. 0.]
 [0. 0. 0. 2.]]
Graph Laplacian:
 [[ 3. -1. -1. -1.]
 [-1.  2. -1.  0.]
 [-1. -1.  3. -1.]
 [-1.  0. -1.  2.]]
```

4. Eigenvalues of Laplacian (connectivity check)

```
eigvals, eigvecs = np.linalg.eigh(L)
print("Laplacian eigenvalues:", eigvals)
```

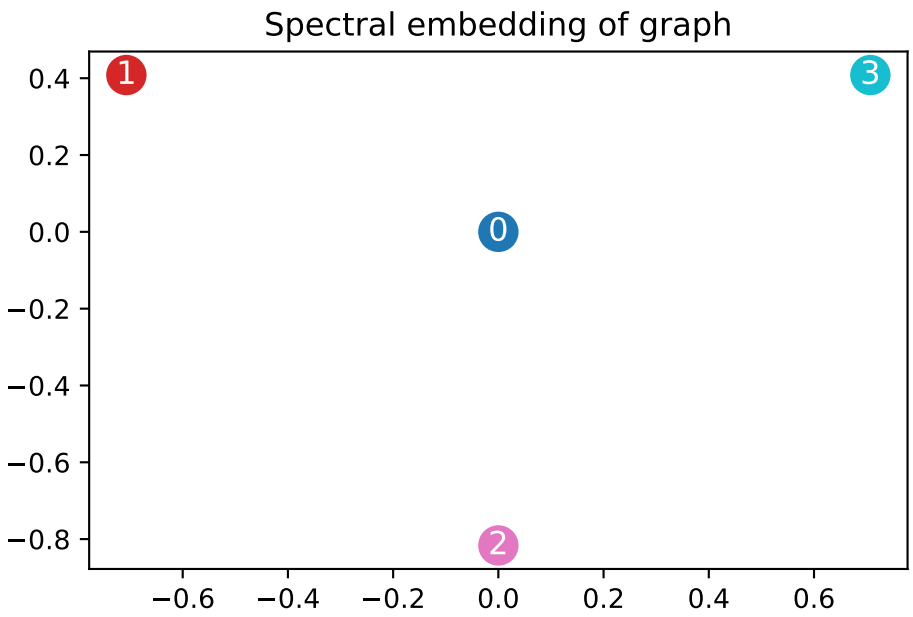
```
Laplacian eigenvalues: [1.11022302e-16 2.00000000e+00 4.00000000e+00 4.00000000e+00]
```

- The number of zero eigenvalues = number of connected components.

5. Spectral embedding (clustering)

Use Laplacian eigenvectors to embed nodes in low dimensions.

```
coords = eigvecs[:,1:3]  # skip the trivial first eigenvector
plt.scatter(coords[:,0], coords[:,1], c=range(len(coords)), cmap="tab10", s=200)
for i, (x,y) in enumerate(coords):
    plt.text(x, y, str(i), fontsize=12, ha="center", va="center", color="white")
plt.title("Spectral embedding of graph")
plt.show()
```



Try It Yourself

1. Remove one edge from the graph and see how Laplacian eigenvalues change.
2. Add a disconnected node - does an extra zero eigenvalue appear?
3. Try a random graph and compare adjacency vs Laplacian spectra.

The Takeaway

- Adjacency matrices describe direct graph structure.
- Laplacians capture connectivity and diffusion.
- Eigenvalues of L reveal graph properties like connectedness and clustering - bridging networks with linear algebra.

94. Data Preprocessing as Linear Ops (Centering, Whitening, Scaling)

Many machine learning and data analysis workflows begin with preprocessing, and linear algebra provides the tools.

- Centering: subtract the mean $\rightarrow$ move data to origin.
- Scaling: divide by standard deviation $\rightarrow$ normalize feature ranges.
- Whitening: decorrelate features $\rightarrow$ make covariance matrix the identity.

Each step can be written as a matrix operation.

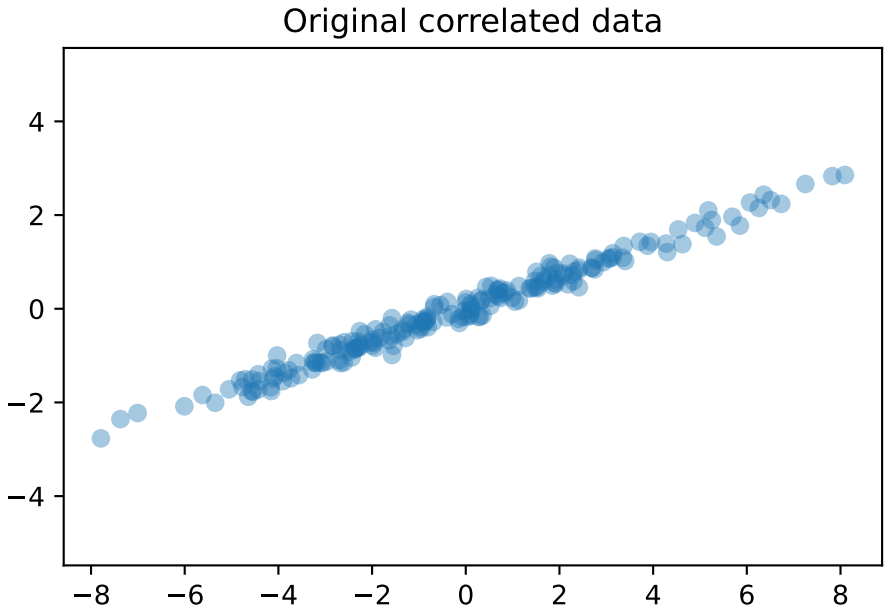
Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Generate correlated data

```
np.random.seed(0)
X = np.random.randn(200, 2) @ np.array([[3,1],[1,0.5]])
plt.scatter(X[:,0], X[:,1], alpha=0.4)
plt.title("Original correlated data")
plt.axis("equal")
plt.show()
```



2. Centering (subtract mean)

```
X_centered = X - X.mean(axis=0)
print("Mean after centering:", X_centered.mean(axis=0))
```

```
Mean after centering: [-7.10542736e-17 -1.33226763e-17]
```

3. Scaling (normalize features)

```
X_scaled = X_centered / X_centered.std(axis=0)
print("Std after scaling:", X_scaled.std(axis=0))
```

```
Std after scaling: [1. 1.]
```

4. Whitening via eigen-decomposition

Covariance of centered data:

```
C = np.cov(X_centered.T)
eigvals, eigvecs = np.linalg.eigh(C)

W = eigvecs @ np.diag(1/np.sqrt(eigvals)) @ eigvecs.T
X_white = X_centered @ W
```

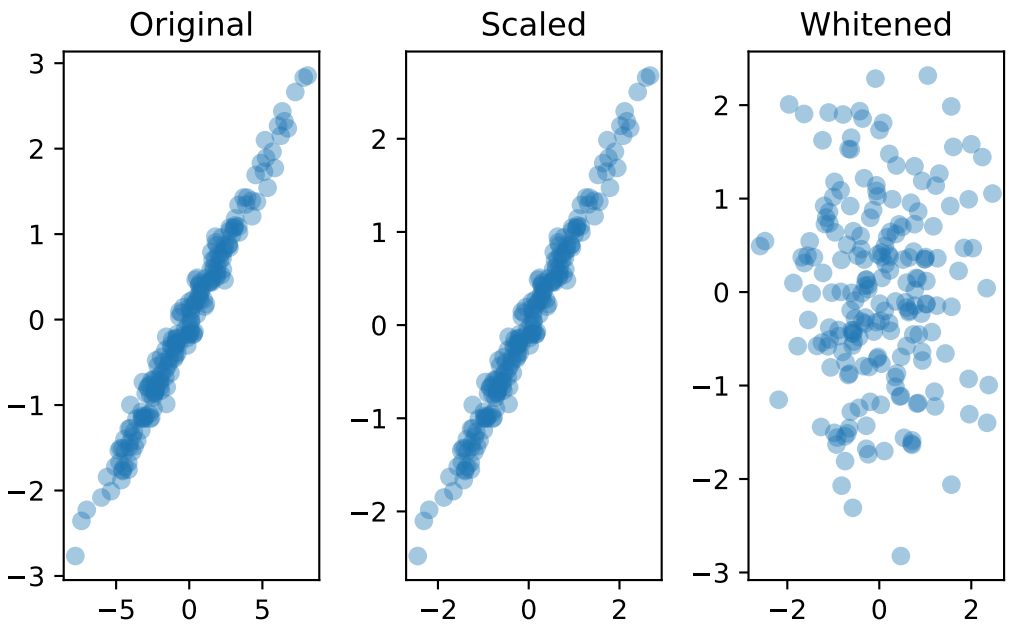
Check covariance:

```
print("Whitened covariance:\n", np.cov(X_white.T))
```

```
Whitened covariance:
 [[ 1.00000000e+00 -1.16104501e-14]
 [-1.16104501e-14  1.00000000e+00]]
```

5. Compare scatter plots

```
plt.subplot(1,3,1)
plt.scatter(X[:,0], X[:,1], alpha=0.4)
plt.title("Original")

plt.subplot(1,3,2)
plt.scatter(X_scaled[:,0], X_scaled[:,1], alpha=0.4)
plt.title("Scaled")

plt.subplot(1,3,3)
plt.scatter(X_white[:,0], X_white[:,1], alpha=0.4)
plt.title("Whitened")

plt.tight_layout()
plt.show()
```



- Original: elongated ellipse.
- Scaled: axis-aligned ellipse.
- Whitened: circular cloud (uncorrelated, unit variance).

Try It Yourself

1. Add a third feature and apply centering, scaling, whitening.
2. Compare whitening with PCA - they use the same eigen-decomposition.
3. Test what happens if you skip centering before whitening.

The Takeaway

- Centering $\rightarrow$ mean zero.
- Scaling $\rightarrow$ unit variance.
- Whitening $\rightarrow$ features uncorrelated, variance = 1. Linear algebra provides the exact matrix operations to make preprocessing systematic and reliable.

95. Linear Regression and Classification (From Model to Matrix)

Linear regression and classification problems can be written neatly in matrix form. This unifies data, models, and solutions under the framework of least squares and linear decision boundaries.

Linear Regression Model

For data (x_i, y_i):

$$y \approx X\beta$$

- X: design matrix (rows = samples, columns = features).
- β: coefficients to solve for.
- Solution (least squares):

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
```

Step-by-Step Code Walkthrough

1. Linear regression example

```
np.random.seed(0)
X = np.linspace(0, 10, 30).reshape(-1,1)
y = 3*X.squeeze() + 5 + np.random.randn(30)*2
```

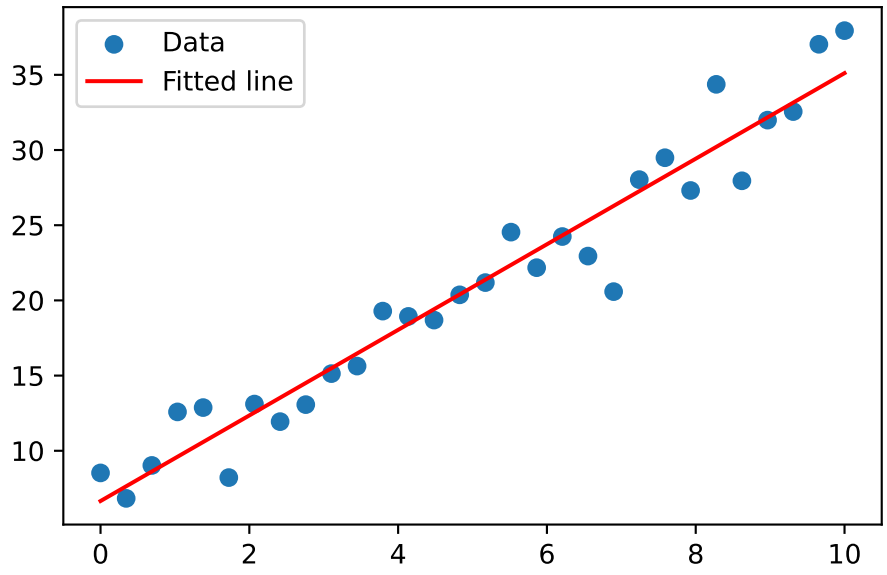
Construct design matrix with bias term:

```
X_design = np.column_stack([np.ones_like(X), X])
beta_hat, *_ = np.linalg.lstsq(X_design, y, rcond=None)
print("Fitted coefficients:", beta_hat)
```

```
Fitted coefficients: [6.65833151 2.84547628]
```

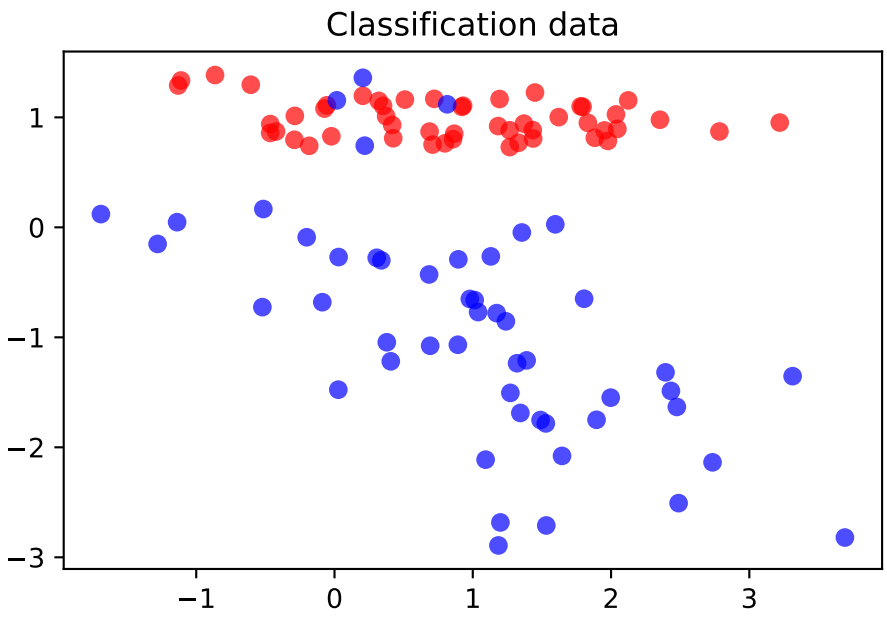
2. Visualize regression line

```
y_pred = X_design @ beta_hat

plt.scatter(X, y, label="Data")
plt.plot(X, y_pred, 'r-', label="Fitted line")
plt.legend()
plt.show()
```



3. Logistic classification with linear decision boundary

```
Xc, yc = make_classification(n_features=2, n_redundant=0, n_informative=2,
                             n_clusters_per_class=1, n_samples=100, random_state=0)

plt.scatter(Xc[:,0], Xc[:,1], c=yc, cmap="bwr", alpha=0.7)
plt.title("Classification data")
plt.show()
```



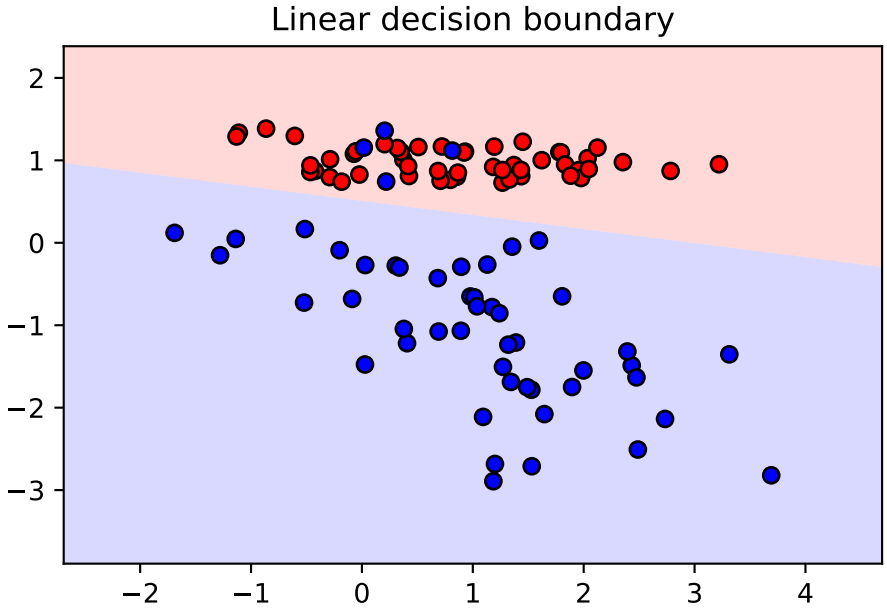
4. Logistic regression via gradient descent

```
def sigmoid(z):
    return 1/(1+np.exp(-z))

X_design = np.column_stack([np.ones(len(Xc)), Xc])
y = yc

w = np.zeros(X_design.shape[1])
lr = 0.1

for _ in range(2000):
    preds = sigmoid(X_design @ w)
    grad = X_design.T @ (preds - y) / len(y)
    w -= lr * grad

print("Learned weights:", w)
```

```
Learned weights: [-2.10451116  0.70752542  4.13295129]
```

5. Plot decision boundary

```
xx, yy = np.meshgrid(np.linspace(Xc[:,0].min()-1, Xc[:,0].max()+1, 200),
                     np.linspace(Xc[:,1].min()-1, Xc[:,1].max()+1, 200))

grid = np.c_[np.ones(xx.size), xx.ravel(), yy.ravel()]
probs = sigmoid(grid @ w).reshape(xx.shape)

plt.contourf(xx, yy, probs, levels=[0,0.5,1], alpha=0.3, cmap="bwr")
plt.scatter(Xc[:,0], Xc[:,1], c=yc, cmap="bwr", edgecolor="k")
plt.title("Linear decision boundary")
plt.show()
```



Try It Yourself

1. Add polynomial features to regression and refit. Does the line bend into a curve?
2. Change learning rate in logistic regression - what happens?
3. Generate data that is not linearly separable. Can a linear model still classify well?

The Takeaway

- Regression and classification fit naturally into linear algebra with matrix formulations.
- Least squares solves regression directly; logistic regression requires optimization.

- Linear models are simple, interpretable, and still form the foundation of modern machine learning.

96. PCA in Practice (Dimensionality Reduction Workflow)

Principal Component Analysis (PCA) is widely used to reduce dimensions, compress data, and visualize high-dimensional datasets. Here, we'll walk through a full PCA workflow: centering, computing components, projecting, and visualizing.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
```

Step-by-Step Code Walkthrough

1. Load dataset (digits)

```
digits = load_digits()
X = digits.data  # shape (1797, 64)
y = digits.target
print("Data shape:", X.shape)
```

```
Data shape: (1797, 64)
```

Each sample is an 8×8 grayscale image flattened into 64 features.

2. Center the data

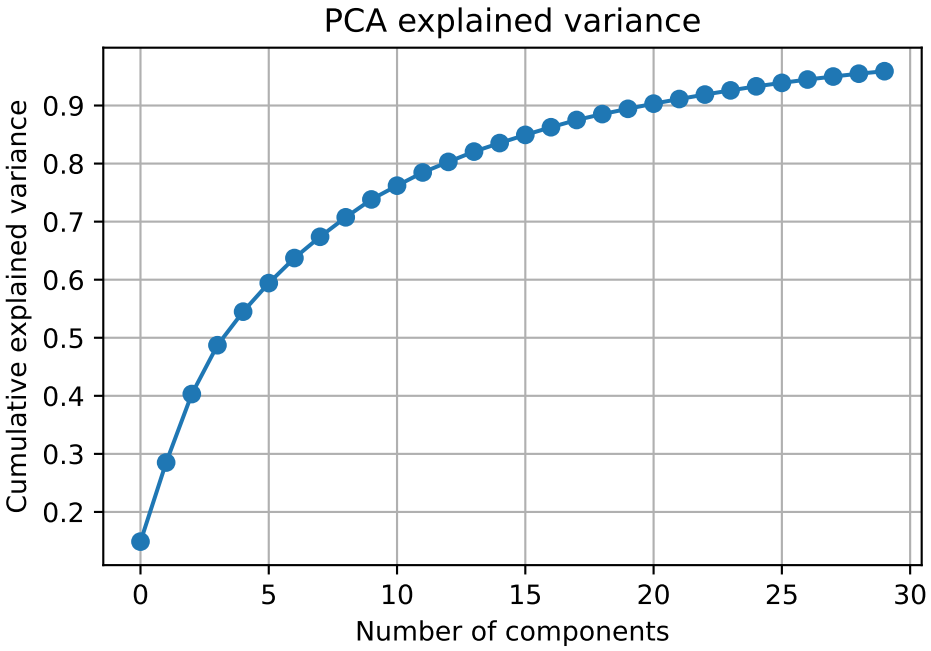
```
X_centered = X - X.mean(axis=0)
```

3. Compute PCA via SVD

```
U, S, Vt = np.linalg.svd(X_centered, full_matrices=False)
explained_variance = (S**2) / (len(X) - 1)
explained_ratio = explained_variance / explained_variance.sum()
```

4. Plot explained variance ratio

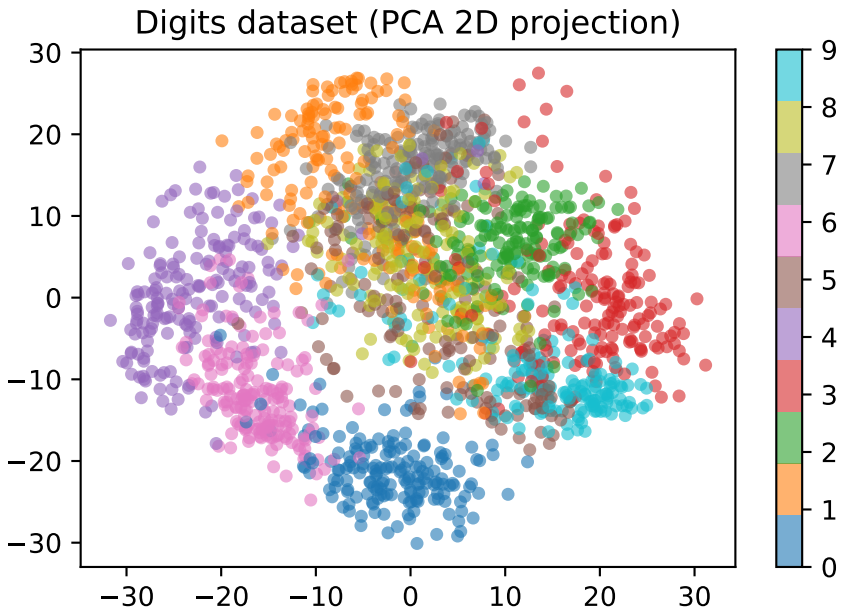
```
plt.plot(np.cumsum(explained_ratio[:30]), 'o-')
plt.xlabel("Number of components")
plt.ylabel("Cumulative explained variance")
plt.title("PCA explained variance")
plt.grid(True)
plt.show()
```



This shows how many components are needed to capture most variance.

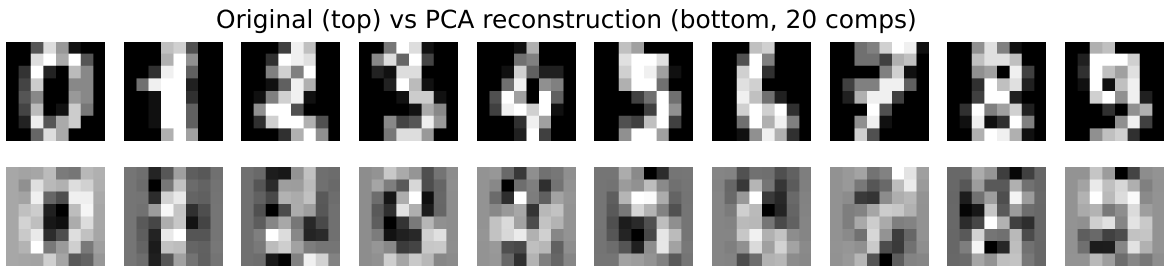
5. Project onto top 2 components for visualization

```
X_pca2 = X_centered @ Vt[:2].T
plt.scatter(X_pca2[:,0], X_pca2[:,1], c=y, cmap="tab10", alpha=0.6, s=15)
plt.colorbar()
plt.title("Digits dataset (PCA 2D projection)")
plt.show()
```



6. Reconstruct images from reduced dimensions

```
k = 20
X_pca20 = X_centered @ Vt[:k].T
X_reconstructed = X_pca20 @ Vt[:k]

fig, axes = plt.subplots(2, 10, figsize=(10,2))
for i in range(10):
    axes[0,i].imshow(X[i].reshape(8,8), cmap="gray")
    axes[0,i].axis("off")
    axes[1,i].imshow(X_reconstructed[i].reshape(8,8), cmap="gray")
    axes[1,i].axis("off")
plt.suptitle("Original (top) vs PCA reconstruction (bottom, 20 comps)")
plt.show()
```



Even with only 20/64 components, the digits remain recognizable.

Try It Yourself

1. Change k to 5, 10, 30 - how do reconstructions change?
2. Use top 2 PCA components to classify digits with k-NN. How does accuracy compare to full 64 features?
3. Try PCA on your own dataset (images, tabular data).

The Takeaway

- PCA reduces dimensions while keeping maximum variance.
- In practice: center → decompose → select top components → project/reconstruct.
- PCA enables visualization, compression, and denoising in real-world workflows.

97. Recommender Systems and Low-Rank Models (Fill the Missing Entries)

Recommender systems often deal with incomplete matrices - rows are users, columns are items, entries are ratings. Most entries are missing, but the matrix is usually close to low-rank (because user preferences depend on only a few hidden factors). SVD and low-rank approximations are powerful tools to fill in these missing values.

Set Up Your Lab

```
import numpy as np
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Simulate a user–item rating matrix

```
np.random.seed(0)
true_users = np.random.randn(10, 3)   # 10 users, 3 latent features
true_items = np.random.randn(3, 8)    # 8 items
R_full = true_users @ true_items      # true low-rank ratings
```

2. Hide some ratings (simulate missing data)

```
mask = np.random.rand(*R_full.shape) > 0.3  # keep 70% of entries
R_obs = np.where(mask, R_full, np.nan)

print("Observed ratings:\n", R_obs)
```

```
Observed ratings:
 [[-1.10781465         nan -3.56526968         nan -2.1729387   1.43510077
   1.46641178  0.79023284]
 [ 0.84819453         nan         nan         nan         nan         nan
   2.30434358  3.03008138]
 [        nan  0.32479187 -0.51818422         nan  0.02013802         nan
   1.29874918  1.33053637]
 [-1.81407786  1.24241182         nan -1.32723907         nan         nan
  -0.31110699         nan]
 [-0.48527696         nan -1.51957106         nan -0.86984941  0.52807989
          nan  0.33771451]
 [-0.26997359 -0.48498966         nan -2.73891459 -2.48167957  2.88740609
  -0.24614835         nan]
 [ 3.57769701 -1.608339    4.73789234  1.13583164  3.63451505 -2.60495928
   2.12453635  3.76472563]
 [ 0.69623809 -0.59117353 -0.28890188 -2.36431192         nan  1.50136796
   0.74268078         nan]
 [ 0.85768141  1.33357168         nan         nan  1.65089037 -2.46456289
   3.51030491  3.31220347]
 [-2.463496    0.60826298 -3.81241599 -2.11839267 -3.86597359  3.52934055
  -1.76203083 -2.63130953]]
```

3. Simple mean imputation (baseline)

```
R_mean = np.where(np.isnan(R_obs), np.nanmean(R_obs), R_obs)
```

4. Apply SVD for low-rank approximation

```
# Replace NaNs with zeros for SVD step
R_filled = np.nan_to_num(R_obs, nan=0.0)

U, S, Vt = np.linalg.svd(R_filled, full_matrices=False)

k = 3  # latent dimension
R_approx = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]
```

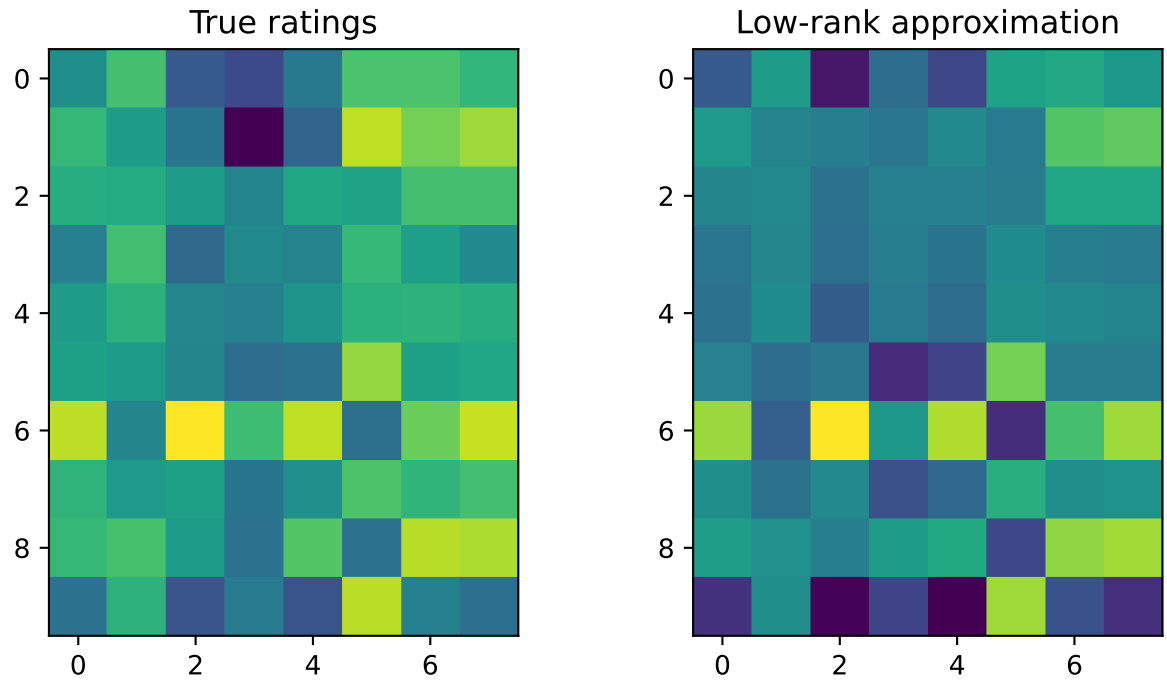
5. Compare filled matrix with ground truth

```
error = np.nanmean((R_full - R_approx)**2)
print("Approximation error (MSE):", error)
```

```
Approximation error (MSE): 1.4862378490976194
```

6. Visualize original vs reconstructed

```
fig, axes = plt.subplots(1, 2, figsize=(8,4))
axes[0].imshow(R_full, cmap="viridis")
axes[0].set_title("True ratings")
axes[1].imshow(R_approx, cmap="viridis")
axes[1].set_title("Low-rank approximation")
plt.show()
```



Try It Yourself

1. Vary k (2, 3, 5). Does error go down?
2. Mask more entries (50%, 80%) - how does SVD reconstruction perform?
3. Use iterative imputation: alternate filling missing entries with low-rank approximations.

The Takeaway

- Recommender systems rely on low-rank structure of user–item matrices.
- SVD provides a natural way to approximate and fill missing ratings.
- This low-rank modeling idea underpins modern collaborative filtering systems like Netflix and Spotify recommenders.

98. PageRank and Random Walks (Ranking with Eigenvectors)

The PageRank algorithm, made famous by Google, uses linear algebra and random walks on graphs to rank nodes (webpages, people, items). The idea: importance flows through links - being linked by important nodes makes you important.

The PageRank Idea

- Start a random walk on a graph: at each step, move to a random neighbor.
- Add a "teleportation" step with probability $1 - \alpha$ to avoid dead ends.
- The steady-state distribution of this walk is the PageRank vector, found as the principal eigenvector of the transition matrix.

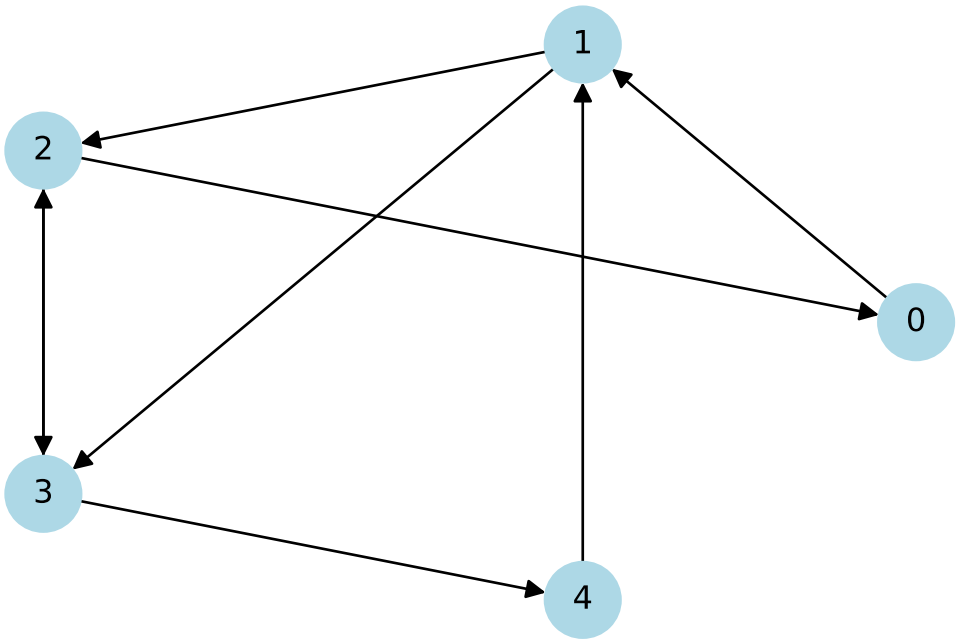
Set Up Your Lab

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

Step-by-Step Code Walkthrough

1. Build a small directed graph

```
G = nx.DiGraph()
G.add_edges_from([
    (0,1), (1,2), (2,0),  # cycle among 0-1-2
    (2,3), (3,2),         # back-and-forth 2-3
    (1,3), (3,4), (4,1)   # small loop with 1-3-4
])
nx.draw_circular(G, with_labels=True, node_color="lightblue", node_size=800, arrowsize=15)
plt.show()
```



2. Build adjacency and transition matrix

```
n = G.number_of_nodes()
A = nx.to_numpy_array(G, nodelist=range(n))
P = A / A.sum(axis=1, keepdims=True)  # row-stochastic transition matrix
```

3. Add teleportation (Google matrix)

```
alpha = 0.85  # damping factor
G_matrix = alpha * P + (1 - alpha) * np.ones((n,n)) / n
```

4. Power iteration to compute PageRank

```
r = np.ones(n) / n  # start uniform
for _ in range(100):
    r = r @ G_matrix
r /= r.sum()
print("PageRank vector:", r)
```

```
PageRank vector: [0.13219034 0.25472358 0.24044787 0.24044787 0.13219034]
```

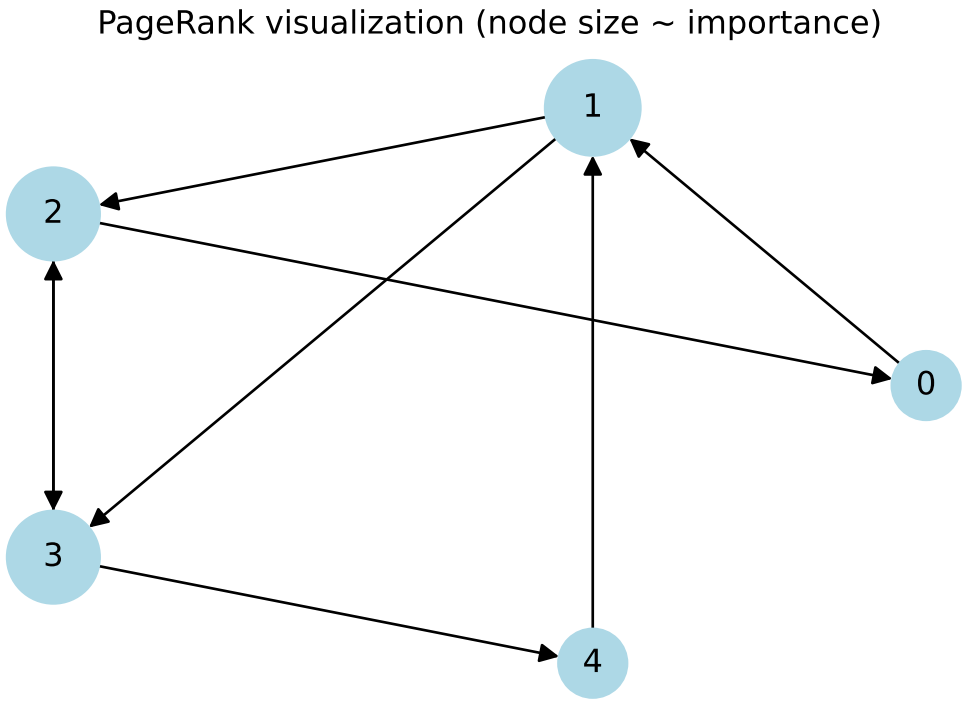
5. Compare with NetworkX built-in

```
pr = nx.pagerank(G, alpha=alpha)
print("NetworkX PageRank:", pr)
```

```
NetworkX PageRank: {0: 0.13219008157546333, 1: 0.2547244023837789, 2: 0.24044771723264727, 3
```

6. Visualize node importance

```
sizes = [5000 * r_i for r_i in r]
nx.draw_circular(G, with_labels=True, node_size=sizes, node_color="lightblue", arrowsize=15)
plt.title("PageRank visualization (node size ~ importance)")
plt.show()
```



Try It Yourself

1. Change α (e.g., 0.6 vs 0.95). Does ranking change?
2. Add a "dangling node" with no outlinks. How does teleportation handle it?
3. Try PageRank on a larger graph (like a random graph with 50 nodes).

The Takeaway

- PageRank is a random-walk steady state problem.
- It reduces to finding the dominant eigenvector of the Google matrix.
- This method generalizes beyond webpages - to influence ranking, recommendation, and network analysis.

99. Numerical Linear Algebra Essentials (Floating Point, BLAS/LAPACK)

When working with linear algebra on computers, numbers are not exact. They live in floating-point arithmetic, and computations rely on highly optimized libraries like BLAS and LAPACK. Understanding these essentials is crucial to doing linear algebra at scale.

Floating Point Basics

- Numbers are stored in base-2 scientific notation:

$$x = \pm(1.b_1 b_2 b_3 \ldots) \times 2^e$$

- Limited precision means rounding errors.
- Two key constants:
  - Machine epsilon ($) : $smallestdifferencedetectable$(10^{-16}$ for double).
  - Overflow/underflow: too large or too small to represent.

Set Up Your Lab

```
import numpy as np
```

Step-by-Step Code Walkthrough

1. Machine epsilon

```
eps = np.finfo(float).eps
print("Machine epsilon:", eps)
```

```
Machine epsilon: 2.220446049250313e-16
```

2. Round-off error demo

```
a = 1e16
b = 1.0
print("a + b - a:", (a + b) - a)  # may lose b due to precision limits
```

```
a + b - a: 0.0
```

3. Stability of matrix inversion

```
A = np.array([[1, 1.0001], [1.0001, 1]])
b = np.array([2, 2.0001])

x_direct = np.linalg.solve(A, b)
x_via_inv = np.linalg.inv(A) @ b

print("Solve:", x_direct)
print("Inverse method:", x_via_inv)
```

```
Solve: [1.499975 0.499975]
Inverse method: [1.499975 0.499975]
```

Notice: using `np.linalg.inv` can be less stable - better to solve directly.

4. Conditioning of a matrix

```
cond = np.linalg.cond(A)
print("Condition number:", cond)
```

```
Condition number: 20000.999999985102
```

- Large condition number → small input changes cause big output changes.

5. BLAS/LAPACK under the hood

```
A = np.random.randn(500, 500)
B = np.random.randn(500, 500)

# Matrix multiplication (calls optimized BLAS under the hood)
C = A @ B
```

This `@` operator is not a naive loop - it calls a highly optimized C/Fortran routine.

Try It Yourself

1. Compare solving `Ax = b` with `np.linalg.solve` vs `np.linalg.inv(A) @ b` for larger, ill-conditioned systems.
2. Use `np.linalg.svd` on a nearly singular matrix. How stable are the singular values?
3. Check performance: time `A @ B` for sizes 100, 500, 1000.

The Takeaway

- Numerical linear algebra = math + floating-point reality.
- Always prefer stable algorithms (`solve`, `qr`, `svd`) over naive inversion.
- Libraries like BLAS/LAPACK make large computations fast, but understanding precision and conditioning prevents nasty surprises.

100. Capstone Problem Sets and Next Steps (A Roadmap to Mastery)

This final section ties everything together. Instead of introducing a new topic, it provides capstone labs that combine multiple ideas from the book. Working through them will give you confidence that you can apply linear algebra to real problems.

Problem Set 1 - Image Compression with SVD

Take an image, treat it as a matrix, and approximate it with low-rank SVD.

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, color

# Load grayscale image
img = color.rgb2gray(data.astronaut())
U, S, Vt = np.linalg.svd(img, full_matrices=False)

# Approximate with rank-k
k = 50
img_approx = U[:, :k] @ np.diag(S[:k]) @ Vt[:k, :]

plt.subplot(1,2,1)
plt.imshow(img, cmap="gray")
plt.title("Original")
plt.axis("off")
```

```
plt.subplot(1,2,2)
plt.imshow(img_approx, cmap="gray")
plt.title(f"Rank-{k} Approximation")
plt.axis("off")

plt.show()
```

Try different k values (5, 20, 100). How does quality vs. compression trade off?

Problem Set 2 - Predictive Modeling with PCA + Regression

Combine PCA for dimensionality reduction with linear regression for prediction.

```
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.decomposition import PCA

# Load dataset
X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# PCA reduce features
pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
```

```
# Regression on reduced space
model = LinearRegression().fit(X_train_pca, y_train)
print("R^2 on test set:", model.score(X_test_pca, y_test))
```

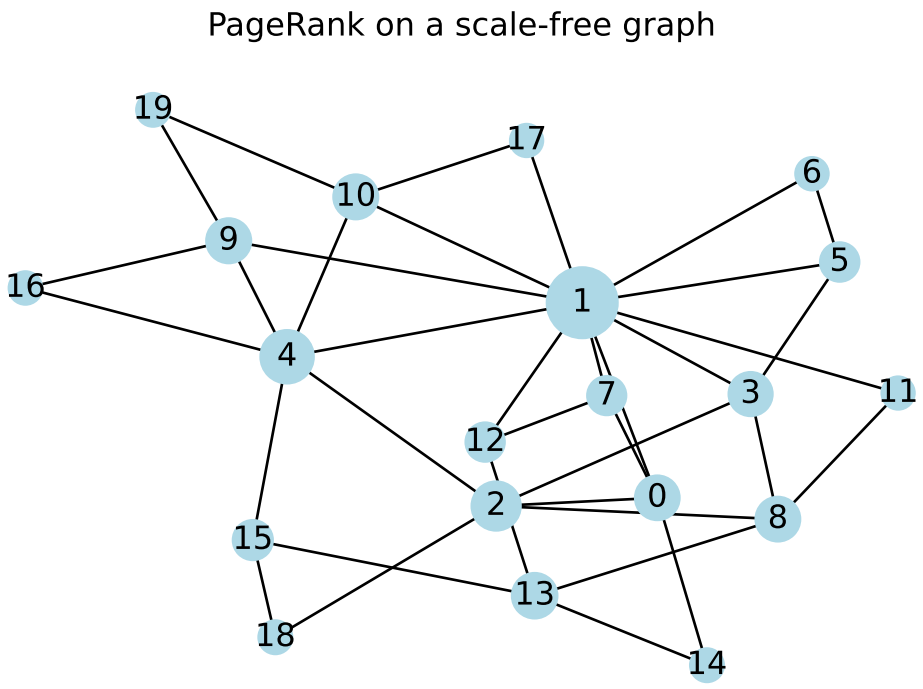
```
R^2 on test set: 0.3691398497153572
```

Does reducing dimensions improve or hurt accuracy?

Problem Set 3 - Graph Analysis with PageRank

Apply PageRank to a custom-built network.

```
import networkx as nx

G = nx.barabasi_albert_graph(20, 2)  # 20 nodes, scale-free graph
pr = nx.pagerank(G, alpha=0.85)

nx.draw(G, with_labels=True, node_size=[5000*pr[n] for n in G], node_color="lightblue")
plt.title("PageRank on a scale-free graph")
plt.show()
```



Which nodes dominate? How does structure affect ranking?

Problem Set 4 - Solving Differential Equations with Eigen Decomposition

Use eigenvalues/eigenvectors to solve a linear dynamical system.

```
A = np.array([[0,1],[-2,-3]])
eigvals, eigvecs = np.linalg.eig(A)

print("Eigenvalues:", eigvals)
print("Eigenvectors:\n", eigvecs)
```

```
Eigenvalues: [-1. -2.]
Eigenvectors:
 [[ 0.70710678 -0.4472136 ]
 [-0.70710678  0.89442719]]
```

Predict long-term behavior: will the system decay, oscillate, or grow?

Problem Set 5 - Least Squares for Overdetermined Systems

```
np.random.seed(0)
X = np.random.randn(100, 3)
beta_true = np.array([2, -1, 0.5])
y = X @ beta_true + np.random.randn(100)*0.1

beta_hat, *_ = np.linalg.lstsq(X, y, rcond=None)
print("Estimated coefficients:", beta_hat)
```

```
Estimated coefficients: [ 1.99371939 -1.00708947  0.50661857]
```

Compare estimated vs. true coefficients. How close are they?

Try It Yourself

1. Combine SVD and recommender systems - build a movie recommender with synthetic data.
2. Implement Gram–Schmidt by hand and test it against `np.linalg.qr`.
3. Write a mini "linear algebra toolkit" with your favorite helper functions.

The Takeaway

- You've practiced vectors, matrices, systems, eigenvalues, SVD, PCA, PageRank, and more.
- Real problems often combine multiple concepts - the labs show how everything fits together.
- Next steps: dive deeper into numerical linear algebra, explore machine learning applications, or study advanced matrix factorizations (Jordan form, tensor decompositions).

This concludes the hands-on journey. By now, you don't just know the theory - you can use linear algebra as a working tool in Python for data, science, and engineering.