LINEAR ALGEBRA

FROM BEGINNER TO EXPERT

A complete, visual, application-driven course — vectors and matrices through the SVD, with a capstone on how it all powers Machine Learning, Deep Learning, and AI.

What you will be able to do

Compute fluently *and* reason geometrically; read the math in modern ML papers; derive the normal equations, the spectral theorem, and the SVD; and explain precisely why linear algebra is the language of AI.

Format: self-paced reference & course | **Prerequisites:** high-school algebra; curiosity

Part of the Comprehensive Machine Learning & Deep Learning Curriculum.

Preface: how to use this book

Linear algebra is the mathematics of *many numbers at once*. The moment you stop thinking about a single quantity and start thinking about lists of quantities — the pixels of an image, the features of a customer, the weights of a neural network, the words of a document — you are doing linear algebra. It is the most useful piece of mathematics for computing, data, and artificial intelligence, and unlike calculus it is concrete: every object is something you can write down, draw, and compute.

Who this is for

This book starts from the very beginning (what a vector *is*) and climbs all the way to the singular value decomposition and the matrix calculus behind backpropagation. It is written for the self-learner who wants both **computational fluency** ("I can do the arithmetic") and **conceptual understanding** ("I know what it means and why it matters"). No proof is included for its own sake; every result is motivated, and most are illustrated with a picture and a small numerical example.

The structure

- **Part I — Foundations** (beginner): vectors, linear systems, matrices.
- **Part II — Core theory** (intermediate): determinants, vector spaces and the four fundamental subspaces, linear transformations, orthogonality and least squares.
- **Part III — Advanced** (expert): eigenvalues, symmetric matrices and the spectral theorem, the SVD, numerical linear algebra, and matrix calculus.
- **Part IV — Applications**: a capstone chapter showing how every concept reappears in machine learning, deep learning, and AI.

How to read it

Read with a pen. When you meet a *Definition* box, restate it in your own words; when you meet an *Example*, cover the solution and try it first; when you meet a *Try it in NumPy* box, actually run the code. The colored boxes recur throughout:

- **Key idea** — the one sentence to remember.
- **Intuition** — the geometric or mental picture.
- **Common pitfall** — mistakes to avoid.
- **How this powers ML / AI** — the forward link to applications.
- **Try it in NumPy** — a hands-on check.

Key idea

Three mental models recur on every page. A matrix is simultaneously (1) a *table of numbers*, (2) a *linear transformation* that moves space, and (3) a *collection of vectors* (its rows or columns). Fluency means switching between these three views at will. The whole book trains that switch.

Contents

Preface: how to use this book 1

I Foundations 5

1 Vectors and the Geometry of Space 6
1.1 What is a vector? . . . 6
1.2 Vector operations . . . 7
1.2.1 Linear combinations — the central concept . . . 7
1.3 The dot product . . . 8
1.4 Length, distance, and norms . . . 8
1.5 Angle, orthogonality, and projection . . . 9
1.5.1 Projection onto a line . . . 9
1.6 Standard basis vectors . . . 10

2 Systems of Linear Equations 11
2.1 What a linear system is . . . 11
2.2 Two geometric pictures . . . 11
2.3 Gaussian elimination . . . 12
2.3.1 Reduced row echelon form (RREF) . . . 12
2.4 Existence and uniqueness . . . 13
2.5 Homogeneous systems . . . 13

3 Matrices and Matrix Algebra 15
3.1 Matrices as tables, transformations, and vector collections . . . 15
3.2 Addition and scalar multiplication . . . 15
3.3 The matrix–vector product (three readings) . . . 15
3.4 Matrix–matrix multiplication . . . 16
3.5 The transpose . . . 16
3.6 Special matrices to recognize . . . 17
3.7 The inverse . . . 17
3.8 Block matrices and the LU factorization . . . 17

II Core Theory 19

4 Determinants 20
4.1 The geometric definition . . . 20
4.2 Computing determinants . . . 20
4.3 Properties that make determinants useful . . . 21
4.4 Determinant, invertibility, and volume . . . 21
4.5 Cramer's rule (and why we avoid it) . . . 21

5 Vector Spaces and the Four Fundamental Subspaces **23**
5.1 Vector spaces and subspaces . . . 23
5.2 Span, independence, basis, dimension . . . 23
5.3 The four fundamental subspaces . . . 24
5.3.1 The rank–nullity theorem . . . 24
5.3.2 The Fundamental Theorem of Linear Algebra . . . 24
5.4 Why the null space governs regularization . . . 25

6 Linear Transformations and Change of Basis **27**
6.1 Linear transformations . . . 27
6.1.1 Kernel and image . . . 28
6.2 Composition is multiplication . . . 28
6.3 Change of basis . . . 28

7 Orthogonality, Projections, and Least Squares **30**
7.1 Orthogonal and orthonormal sets . . . 30
7.2 Orthogonal complements . . . 30
7.3 Projection onto a subspace . . . 30
7.4 Least squares: the normal equations . . . 31
7.5 Gram–Schmidt and the QR factorization . . . 32

III Advanced Topics **33**

8 Eigenvalues and Eigenvectors **34**
8.1 The eigenvalue equation . . . 34
8.2 Finding eigenvalues: the characteristic polynomial . . . 34
8.3 Diagonalization . . . 35
8.4 Applications: Markov chains and PageRank . . . 35

9 Symmetric Matrices, the Spectral Theorem, and Quadratic Forms **37**
9.1 The spectral theorem . . . 37
9.2 Quadratic forms . . . 38
9.3 Positive definiteness . . . 38
9.4 The Rayleigh quotient . . . 38
9.5 Covariance, Gram, and kernel matrices . . . 39

10 The Singular Value Decomposition **40**
10.1 The decomposition . . . 40
10.2 Relation to eigendecomposition . . . 40
10.3 The four subspaces, revealed . . . 41
10.4 Low-rank approximation: the Eckart–Young theorem . . . 41
10.5 The pseudoinverse . . . 42

11 Matrix Decompositions, Norms, and Numerical Linear Algebra **44**
11.1 A field guide to the decompositions . . . 44
11.1.1 Cholesky in one line . . . 45
11.2 Vector and matrix norms . . . 45
11.3 The condition number . . . 45
11.4 Algorithms behind the libraries . . . 46

12 Tensors and Matrix Calculus **48**
12.1 Tensors: arrays of any order . . . 48
12.1.1 Broadcasting and einsum . . . 48
12.2 Derivatives in many dimensions . . . 49
12.3 Essential gradient identities . . . 49
12.4 The chain rule and backpropagation . . . 49

IV Applications **52**

13 Linear Algebra in Machine Learning, Deep Learning, and AI **53**
13.1 Data is linear algebra . . . 53
13.2 Linear and ridge regression: projection and the pseudoinverse . . . 53
13.3 Principal Component Analysis: the SVD of data . . . 53
13.4 Neural networks: stacks of affine maps . . . 54
13.5 Convolution and weight sharing . . . 54
13.6 Embeddings and similarity . . . 54
13.7 Attention and Transformers . . . 55
13.8 Low-rank structure: LoRA and model compression . . . 55
13.9 Graphs, spectra, and PageRank . . . 55
13.10 Probabilistic models and Gaussian processes . . . 55
13.11 Optimization landscapes: curvature is eigenvalues . . . 55
13.12 The grand summary table . . . 56

A Notation and Formula Reference **58**
A.1 Notation . . . 58
A.2 Core identities . . . 58
A.3 Equivalent conditions for an invertible $n \times n$ matrix . . . 59
A.4 Greek letters used . . . 59
A.5 Minimal NumPy / PyTorch reference . . . 59

PART I

Foundations

1

Vectors and the Geometry of Space

Level: Beginner

Everything in linear algebra is built from one object: the **vector**. Before we can multiply matrices or diagonalize transformations, we need to be completely comfortable with what a vector is, how to combine vectors, and how to measure them. This chapter builds that foundation and — crucially — two ways of *seeing* it: the geometric picture (arrows in space) and the data picture (lists of numbers).

1.1 What is a vector?

Definition 1.1. A **vector** in $\mathbb{R}^n$ is an ordered list of n real numbers, written as a column:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \qquad v_i \in \mathbb{R}.$$

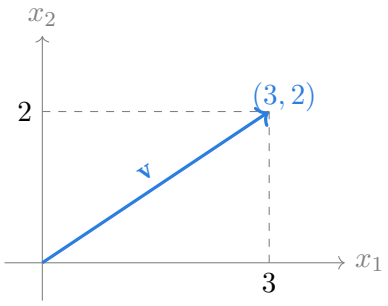
The numbers v_i are the **components** (or entries) of the vector, and n is its **dimension**. The set of all such vectors is the space $\mathbb{R}^n$.

There are two complementary ways to interpret this list, and switching between them fluently is the first real skill of linear algebra.

Two pictures of one vector

The geometric picture. A vector in $\mathbb{R}^2$ or $\mathbb{R}^3$ is an *arrow* from the origin to the point with those coordinates. It has a length and a direction. This is how we reason about angles, distance, and projection.

The data picture. A vector is simply a *record*: one house is $[1800, 3, 1995]^\top$ (square feet, bedrooms, year). Here there is no "arrow"; the vector is a point in an abstract 3-dimensional feature space. Machine learning lives almost entirely in this picture — but it *borrows* all the geometric tools (distance, angle, projection) from the first.



1.2 Vector operations

Two operations generate *all* of linear algebra: adding vectors, and scaling them by a number.

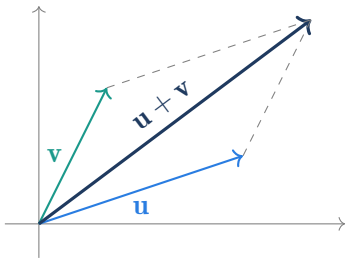
Definition 1.2. For $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ and a scalar $c \in \mathbb{R}$:

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_1 + v_1 \\ \vdots \\ u_n + v_n \end{bmatrix}, \qquad c\,\mathbf{v} = \begin{bmatrix} c\,v_1 \\ \vdots \\ c\,v_n \end{bmatrix}.$$

Addition is componentwise; scalar multiplication stretches ($|c| > 1$), shrinks ($|c| < 1$), or flips ($c < 0$) the arrow.

Intuition

Geometrically, $\mathbf{u} + \mathbf{v}$ is the *tip-to-tail* rule: walk along $\mathbf{u}$, then along $\mathbf{v}$; you land at $\mathbf{u} + \mathbf{v}$ (the diagonal of the parallelogram). Scalar multiplication slides a vector along its own line. These two pictures — the parallelogram and the line — underlie every later idea about spans and subspaces.



1.2.1 Linear combinations — the central concept

Definition 1.3. A **linear combination** of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ is any vector of the form

$$c_1\mathbf{v}_1 + c_2\mathbf{v}_2 + \cdots + c_k\mathbf{v}_k, \qquad c_i \in \mathbb{R}.$$

Key idea

The linear combination is the single most important idea in the subject. Almost every question in linear algebra is a disguised version of: "Can vector $\mathbf{b}$ be written as a linear combination of these other vectors, and if so, how?" Solving $\mathbf{Ax} = \mathbf{b}$, finding a basis, computing rank — all are linear-combination questions.

The set of *all* linear combinations of a set of vectors is called their **span**; we study it carefully in Chapter 5. For now, hold this picture: two non-parallel vectors in $\mathbb{R}^3$ span a plane through the origin; their linear combinations sweep out that entire plane.

1.3 The dot product

The dot product takes two vectors and returns a single number that measures how much they point the same way. It is the bridge between algebra and geometry.

Definition 1.4. The **dot product** (or inner product) of $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ is

$$\mathbf{u} \cdot \mathbf{v} \;=\; \mathbf{u}^\top \mathbf{v} \;=\; \sum_{i=1}^{n} u_i v_i \;=\; u_1 v_1 + \cdots + u_n v_n.$$

It is *symmetric* ($\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{u}$), *linear* in each argument, and connects to geometry through

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \, \|\mathbf{v}\| \cos\theta,$$

where θ is the angle between the two vectors.

Example 1.5. Let $\mathbf{u} = [2, 1, 3]^\top$ and $\mathbf{v} = [1, 0, -1]^\top$. Then $\mathbf{u}\cdot\mathbf{v} = (2)(1)+(1)(0)+(3)(-1) = 2 + 0 - 3 = -1$. The result is negative, so the vectors point in broadly *opposing* directions (the angle between them exceeds $90°$).

Intuition

The sign of the dot product is a quick compass: *positive* means the vectors broadly agree in direction, *zero* means they are perpendicular, *negative* means they broadly oppose. "Alignment" is the word to remember — and it is exactly the notion that reappears as *similarity* between embeddings in AI.

How this powers ML / AI

A single artificial neuron computes $\mathbf{w}^\top \mathbf{x} + b$ — a dot product of a weight vector with an input, plus a bias. "Similarity" between two pieces of text, two images, or two users in a recommender system is almost always a (normalized) dot product. When a vector database returns "nearest" results by *cosine similarity*, it is ranking by $\frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} = \cos\theta$. The dot product is the most-executed operation in all of machine learning.

1.4 Length, distance, and norms

Definition 1.6. The **Euclidean norm** (length) of $\mathbf{v} \in \mathbb{R}^n$ is

$$\|\mathbf{v}\|_2 = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \sqrt{v_1^2 + \cdots + v_n^2}.$$

The **distance** between two vectors is the length of their difference, $\|\mathbf{u} - \mathbf{v}\|_2$. A vector with $\|\mathbf{v}\| = 1$ is a **unit vector**; we *normalize* any nonzero $\mathbf{v}$ by $\mathbf{v}/\|\mathbf{v}\|$.

Different ways of measuring "size" are useful in different settings. The general family is the

p-norms:

$$\|\mathbf{v}\|_1 = \sum_i |v_i| \text{ (“Manhattan”)}, \qquad \|\mathbf{v}\|_2 = \sqrt{\textstyle\sum_i v_i^2} \text{ (Euclidean)}, \qquad \|\mathbf{v}\|_\infty = \max_i |v_i|.$$

How this powers ML / AI

These norms are exactly the regularizers of machine learning. Penalizing $\|\mathbf{w}\|_2^2$ is **Ridge** regression (smooth shrinkage of weights); penalizing $\|\mathbf{w}\|_1$ is **Lasso**, which drives some weights to *exactly* zero and so performs feature selection. The geometric difference — a round ball versus a pointed diamond — is *why* L1 produces sparsity, a fact we will make precise later.

1.5 Angle, orthogonality, and projection

Rearranging the geometric form of the dot product gives the angle directly:

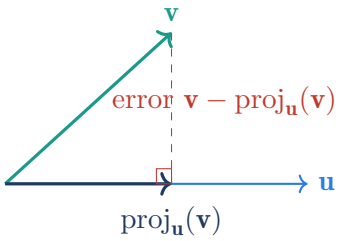
$$\cos\theta = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\,\|\mathbf{v}\|}.$$

Definition 1.7. Two vectors are **orthogonal** (perpendicular) when $\mathbf{u}\cdot\mathbf{v} = 0$. Orthogonality is the most important special relationship between vectors; entire chapters (projections, least squares, the SVD) are built on it.

1.5.1 Projection onto a line

How much of $\mathbf{v}$ lies along the direction of $\mathbf{u}$? Drop a perpendicular from the tip of $\mathbf{v}$ onto the line through $\mathbf{u}$; the shadow is the **projection**:

$$\text{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{u}\cdot\mathbf{v}}{\mathbf{u}\cdot\mathbf{u}}\,\mathbf{u}.$$



Key idea

The projection is the point on the line *closest* to $\mathbf{v}$, and the error $\mathbf{v} - \text{proj}_{\mathbf{u}}(\mathbf{v})$ is orthogonal to $\mathbf{u}$. “Closest point = orthogonal error” is the seed of least-squares regression (Chapter 7): fitting a model is projecting the data onto the space of possible predictions.

Try it in NumPy

```python
import numpy as np
u = np.array([2.0, 1.0, 3.0]); v = np.array([1.0, 0.0, -1.0])
dot   = u @ v                          # -1.0
norm  = np.linalg.norm(u)              # sqrt(14)
```

```
cos   = (u @ v) / (np.linalg.norm(u) * np.linalg.norm(v))
proj  = (u @ v) / (u @ u) * u        # projection of v onto u
print(dot, norm, cos, proj)
```

1.6 Standard basis vectors

The vectors $\mathbf{e}_1 = [1, 0, \ldots, 0]^\top$, $\mathbf{e}_2 = [0, 1, 0, \ldots]^\top, \ldots, \mathbf{e}_n$ are the **standard basis** of $\mathbb{R}^n$. Every vector is a unique linear combination of them: $\mathbf{v} = v_1\mathbf{e}_1 + \cdots + v_n\mathbf{e}_n$. They are mutually orthogonal unit vectors — the cleanest possible coordinate system, and the prototype for the *orthonormal bases* that make the SVD so powerful.

Chapter summary

- A vector is a list of numbers, seen either as an arrow (geometry) or a record (data).
- Vector addition and scalar multiplication generate **linear combinations**, the core concept.
- The **dot product** $\mathbf{u}^\top\mathbf{v} = \|\mathbf{u}\|\,\|\mathbf{v}\|\cos\theta$ measures alignment; it underlies neurons and similarity.
- Norms measure size; L2/L1 norms are Ridge/Lasso regularizers.
- **Orthogonality** ($\mathbf{u}\cdot\mathbf{v} = 0$) and **projection** (closest point, orthogonal error) seed least squares.

2

Systems of Linear Equations

Level: Beginner

A **system of linear equations** asks: which values of the unknowns satisfy several linear constraints at once? This is the oldest problem in the subject and still its computational core — training a linear model, balancing a chemical equation, and solving a circuit all reduce to it. The goal of this chapter is the algorithm of **elimination** and, more importantly, the two geometric pictures that make solvability obvious.

2.1 What a linear system is

Definition 2.1. A **linear equation** in unknowns $x_1, \dots, x_n$ has the form $a_1x_1 + \cdots + a_nx_n = b$ — each unknown appears only to the first power, never multiplied together or inside a nonlinear function. A **system** of m such equations can be written compactly as

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \qquad \mathbf{A} \in \mathbb{R}^{m\times n},\ \mathbf{x} \in \mathbb{R}^{n},\ \mathbf{b} \in \mathbb{R}^{m},$$

where $\mathbf{A}$ holds the coefficients, $\mathbf{x}$ the unknowns, and $\mathbf{b}$ the right-hand sides.

Example 2.2. The system $2x + y = 5,\ \ x - y = 1$ becomes

$$\underbrace{\begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} 5 \\ 1 \end{bmatrix}}_{\mathbf{b}}.$$

Its solution is $x = 2,\ y = 1$.

2.2 Two geometric pictures

Row picture vs. column picture

There are two ways to "see" $\mathbf{A}\mathbf{x} = \mathbf{b}$, and Gilbert Strang's central teaching is that the second is the more powerful.

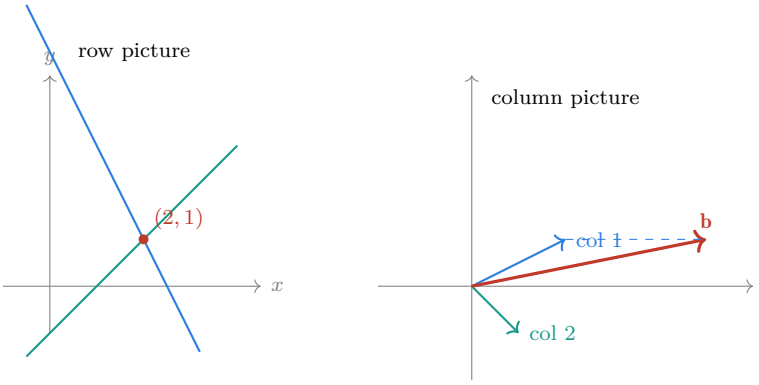
Row picture. Each equation is a line (in $\mathbb{R}^2$), plane (in $\mathbb{R}^3$), or hyperplane. A solution is a point lying on *all* of them — the intersection.

Column picture. Write the system as a linear combination of the *columns* of $\mathbf{A}$:

$$x \begin{bmatrix} 2 \\ 1 \end{bmatrix} + y \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}.$$

Now the question is: *what combination of the column vectors produces* $\mathbf{b}$*?* The system is

solvable exactly when $\mathbf{b}$ lies in the span of the columns of $\mathbf{A}$.



Key idea

The column picture is the one to internalize: *solving* $\mathbf{Ax} = \mathbf{b}$ *means expressing* $\mathbf{b}$ *as a linear combination of* $\mathbf{A}$*'s columns.* This reframing is what later turns "does a solution exist?" into "is $\mathbf{b}$ in the column space?" — the question at the heart of Chapter 5.

2.3 Gaussian elimination

The systematic method to solve any linear system is **Gaussian elimination**: use simple row operations to reduce the system to an easily-solved triangular form.

Definition 2.3. The three **elementary row operations** do not change the solution set:
(1) swap two rows;
(2) multiply a row by a nonzero scalar;
(3) add a multiple of one row to another.

Applying these to drive entries below the diagonal to zero yields **row echelon form**; **back-substitution** then solves from the bottom up. The leading nonzero entry in each row is a **pivot**.

Example 2.4 (Elimination step by step)**.** Solve $x + 2y + z = 2$, $2x + 5y + 3z = 7$, $x + 3y + 3z = 6$. The augmented matrix and its reduction:

$$\left[\begin{array}{ccc|c} 1 & 2 & 1 & 2 \\ 2 & 5 & 3 & 7 \\ 1 & 3 & 3 & 6 \end{array}\right] \xrightarrow[R_3 - R_1]{R_2 - 2R_1} \left[\begin{array}{ccc|c} 1 & 2 & 1 & 2 \\ 0 & 1 & 1 & 3 \\ 0 & 1 & 2 & 4 \end{array}\right] \xrightarrow{R_3 - R_2} \left[\begin{array}{ccc|c} 1 & 2 & 1 & 2 \\ 0 & 1 & 1 & 3 \\ 0 & 0 & 1 & 1 \end{array}\right].$$

Back-substitution: $z = 1$, then $y = 3 - z = 2$, then $x = 2 - 2y - z = 2 - 4 - 1 = -3$. Solution $(-3, 2, 1)$. The three pivots $(1, 1, 1)$ sit on the diagonal — a full set of pivots signals a unique solution.

2.3.1 Reduced row echelon form (RREF)

Continuing until each pivot equals 1 and is the only nonzero entry in its column gives the **reduced row echelon form**, the canonical endpoint of elimination. RREF makes the solution (and the structure of all solutions) readable directly, and it is the engine we will use to compute the four fundamental subspaces.

2.4 Existence and uniqueness

After elimination, three outcomes are possible, and the pivots tell you which.

Situation	Geometric meaning	Solutions
pivot in every column, consistent	lines meet at one point	exactly one
a free column (fewer pivots than unknowns)	lines/planes overlap	infinitely many
a pivot in the augmented column only	parallel, no common point	none (inconsistent)

Definition 2.5. A column without a pivot corresponds to a **free variable** — it can take any value, and each choice yields a different solution. The presence of free variables is exactly why a system has *infinitely many* solutions.

Common pitfall

"More equations than unknowns" does *not* guarantee a unique solution, and "more unknowns than equations" does not guarantee infinitely many. What matters is the number of **pivots** (the rank), not the raw counts of rows and columns. A tall system can still be inconsistent; a wide one can still be uniquely solvable on its pivot variables.

2.5 Homogeneous systems

When $\mathbf{b} = \mathbf{0}$, the system $\mathbf{Ax} = \mathbf{0}$ is **homogeneous**. It always has at least the trivial solution $\mathbf{x} = \mathbf{0}$. It has *nontrivial* solutions exactly when there is a free variable. The set of all solutions is the **null space** of $\mathbf{A}$ (Chapter 5) — our first fundamental subspace, hiding in plain sight.

How this powers ML / AI

"Solve $\mathbf{Ax} = \mathbf{b}$" is the skeleton of linear regression: stacking data rows into $\mathbf{A}$ and targets into $\mathbf{b}$, we seek weights $\mathbf{x}$. Real data has more equations (samples) than unknowns (features) and no exact solution, so we will not solve it exactly — we will find the *closest* fit by least squares (Chapter 7). When features are redundant, free variables appear, the solution is non-unique, and **regularization** is needed to pick one. Elimination is also how libraries actually solve the linear systems inside Newton's method and Gaussian processes.

Try it in NumPy

```python
import numpy as np
A = np.array([[1,2,1],[2,5,3],[1,3,3]], float)
b = np.array([2,7,6], float)
x = np.linalg.solve(A, b)      # prefer solve(...) over inv(A) @ b
print(x)                       # [-3.  2.  1.]
print(np.linalg.matrix_rank(A))   # 3 pivots -> unique solution
```

Chapter summary

- A linear system is $\mathbf{Ax} = \mathbf{b}$; read it through the **row** picture (intersecting hyperplanes) and the more powerful **column** picture (combination of columns equal to $\mathbf{b}$).

- **Gaussian elimination** with three row operations reduces any system to (reduced) row echelon form; **pivots** drive the analysis.
- Solutions number one, infinitely many, or none — determined by pivots and consistency, not by counting equations.
- **Free variables** cause infinitely many solutions; the homogeneous system's solutions form the null space.

3

Matrices and Matrix Algebra

Level: Beginner

A matrix is the workhorse object of linear algebra. This chapter develops the algebra of matrices — how to add, multiply, transpose, and invert them — always keeping sight of the three views from the preface: a matrix is a *table*, a *transformation*, and a *collection of vectors*.

3.1 Matrices as tables, transformations, and vector collections

Definition 3.1. A **matrix** $\mathbf{A} \in \mathbb{R}^{m\times n}$ is a rectangular array of numbers with m rows and n columns; the entry in row i, column j is a_{ij}. When $m = n$ the matrix is **square**.

Intuition

The three views, on one matrix $\mathbf{A}$:

- **Table:** a spreadsheet — e.g. rows are customers, columns are features.
- **Transformation:** a machine that takes a vector $\mathbf{x} \in \mathbb{R}^n$ and returns $\mathbf{Ax} \in \mathbb{R}^m$, rotating/stretching/projecting space.
- **Vector collection:** a bundle of n column vectors (or m row vectors). Its column space and row space (Chapter 5) come from here.

3.2 Addition and scalar multiplication

These are componentwise and require matching shapes: $(\mathbf{A}+\mathbf{B})_{ij} = a_{ij} + b_{ij}$ and $(c\mathbf{A})_{ij} = c\,a_{ij}$. They obey the obvious commutative/associative/distributive rules, so matrices of a fixed shape form a vector space themselves (Chapter 5).

3.3 The matrix–vector product (three readings)

The product $\mathbf{Ax}$ is the heart of the subject; read it three ways.

Definition 3.2. For $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{x} \in \mathbb{R}^n$, the product $\mathbf{Ax} \in \mathbb{R}^m$ can be computed as:

(1) **Row view:** entry i is the dot product of row i of $\mathbf{A}$ with $\mathbf{x}$.

(2) **Column view:** $\mathbf{Ax} = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n$ — a *linear combination of the columns* $\mathbf{a}_j$ of $\mathbf{A}$, weighted by the entries of $\mathbf{x}$.

(3) **Transformation view:** $\mathbf{A}$ maps the input vector $\mathbf{x}$ to an output vector in $\mathbb{R}^m$.

Key idea

The column view, $\mathbf{Ax} = \sum_j x_j \mathbf{a}_j$, is the most important identity in elementary linear algebra. It says the *outputs* of $\mathbf{A}$ are exactly the linear combinations of its columns — so the reachable outputs form the column space, and $\mathbf{Ax} = \mathbf{b}$ is solvable iff $\mathbf{b}$ lies there. Hold this beside the column picture of Chapter 2.

3.4 Matrix–matrix multiplication

Definition 3.3. For $\mathbf{A} \in \mathbb{R}^{m\times k}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$, the product $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m\times n}$ has entries

$$c_{ij} = \sum_{\ell=1}^{k} a_{i\ell}\, b_{\ell j}$$

— the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$. The *inner* dimensions must match ($k = k$); the *outer* dimensions give the result's shape ($m \times n$).

Intuition

Matrix multiplication is **composition of transformations**: $(\mathbf{AB})\mathbf{x} = \mathbf{A}(\mathbf{Bx})$ means "apply $\mathbf{B}$, then $\mathbf{A}$." That is why the order matters and why the inner dimensions must agree — the output of $\mathbf{B}$ must be a valid input to $\mathbf{A}$. A second useful reading: column j of $\mathbf{AB}$ is $\mathbf{A}$ applied to column j of $\mathbf{B}$.

Example 3.4.

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 3 & 1 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 1\cdot 3 + 2\cdot 2 & 1\cdot 1 + 2\cdot 4 \\ 0\cdot 3 + 1\cdot 2 & 0\cdot 1 + 1\cdot 4 \end{bmatrix} = \begin{bmatrix} 7 & 9 \\ 2 & 4 \end{bmatrix}.$$

Common pitfall

Matrix multiplication is **not commutative**: in general $\mathbf{AB} \neq \mathbf{BA}$ (often they do not even have compatible shapes). It *is* associative ($\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$) and distributive. Also beware: $\mathbf{AB} = \mathbf{0}$ does *not* imply $\mathbf{A} = \mathbf{0}$ or $\mathbf{B} = \mathbf{0}$, and $\mathbf{AB} = \mathbf{AC}$ does not imply $\mathbf{B} = \mathbf{C}$ unless $\mathbf{A}$ is invertible.

How this powers ML / AI

A neural-network layer is one matrix multiply plus a bias and a nonlinearity: $\mathbf{h} = \phi(\mathbf{Wx}+\mathbf{b})$. A whole batch of inputs is processed at once as $\mathbf{XW}^\top$ — this is why GPUs, which are matrix-multiply engines, power deep learning. Stacking layers is composing linear maps separated by nonlinearities; without the nonlinearity the composition $\mathbf{W}_2\mathbf{W}_1$ collapses to a single matrix. Essentially all training time is spent on matrix multiplications.

3.5 The transpose

The **transpose** $\mathbf{A}^\top$ flips rows and columns: $(\mathbf{A}^\top)_{ij} = a_{ji}$. Key rules: $(\mathbf{A}^\top)^\top = \mathbf{A}$, $(\mathbf{A}+\mathbf{B})^\top = \mathbf{A}^\top + \mathbf{B}^\top$, and the order-reversing

$$(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top.$$

The combination $\mathbf{A}^\top\mathbf{A}$ (always square, symmetric, and positive semidefinite) appears constantly — it is the matrix behind the normal equations and PCA.

3.6 Special matrices to recognize

Type	Property	Why it matters
Identity $\mathbf{I}$	$\mathbf{Ix} = \mathbf{x}$; ones on diagonal	the "do nothing" map
Diagonal $\mathbf{D}$	off-diagonal zeros	scales each axis independently
Symmetric	$\mathbf{A}^\top = \mathbf{A}$	real eigenvalues; spectral theorem (Ch. 9)
Triangular	zeros above/below diagonal	easy to solve; output of elimination
Orthogonal $\mathbf{Q}$	$\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$	rotations/reflections; preserve length
Permutation	rows of $\mathbf{I}$ reordered	row swaps in elimination

Intuition

Orthogonal matrices are the "rigid motions": they preserve every length and angle because $\|\mathbf{Qx}\| = \|\mathbf{x}\|$. They are numerically golden — they never amplify error — which is exactly why the QR and SVD factorizations (built from orthogonal matrices) are the stable backbone of numerical computing.

3.7 The inverse

Definition 3.5. A square matrix $\mathbf{A}$ is **invertible** (nonsingular) if there exists $\mathbf{A}^{-1}$ with $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. Then $\mathbf{Ax} = \mathbf{b}$ has the unique solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. Equivalent conditions: $\mathbf{A}$ has full rank, nonzero determinant, no zero eigenvalue, and trivial null space.

For 2×2, $\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$, valid when $ad - bc \neq 0$. The order-reversing rule holds: $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$.

Common pitfall

In numerical code, **do not compute an explicit inverse to solve a system.** Use `np.linalg.solve(A, b)` rather than `np.linalg.inv(A) @ b`: it is faster, more accurate, and avoids amplifying round-off. Forming $\mathbf{A}^{-1}$ is almost always a sign of a suboptimal algorithm. (Inverses are fine for derivations and for small fixed matrices.)

3.8 Block matrices and the LU factorization

Matrices can be partitioned into **blocks** and multiplied block-by-block (provided shapes conform) — a perspective that makes many proofs and efficient algorithms transparent.

Elimination itself is a factorization. Recording the multipliers used to zero out entries gives the **LU decomposition**:

$$\mathbf{A} = \mathbf{LU},$$

with $\mathbf{L}$ lower-triangular (the multipliers) and $\mathbf{U}$ upper-triangular (the echelon form). With row swaps it becomes $\mathbf{PA} = \mathbf{LU}$. LU is how production solvers handle $\mathbf{Ax} = \mathbf{b}$: factor once, then solve cheaply for many right-hand sides by forward/back substitution.

Try it in NumPy

```python
import numpy as np
from scipy.linalg import lu
A = np.array([[2.0,1,1],[4,3,3],[8,7,9]])
P, L, U = lu(A)                 # A = P @ L @ U
print(np.allclose(P @ L @ U, A))   # True
print(np.linalg.inv(A) @ A)        # ~ identity (but prefer solve in
    practice)
```

Chapter summary

- A matrix is a table, a transformation, and a collection of vectors — switch views freely.
- **Ax** is a **linear combination of the columns** of **A**; this is the key identity.
- Matrix multiplication composes transformations: associative and distributive, but *not* commutative.
- Transpose reverses order: $(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top$; $\mathbf{A}^\top\mathbf{A}$ is symmetric PSD and ubiquitous.
- Know the special matrices (identity, diagonal, symmetric, orthogonal, triangular); orthogonal matrices preserve length.
- The inverse undoes a map but is rarely computed numerically; elimination = the $\mathbf{A} = \mathbf{LU}$ factorization.

PART II

Core Theory

4

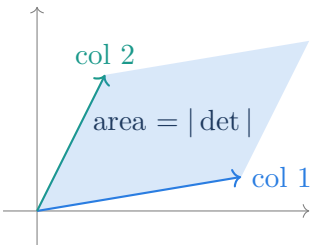
Determinants

Level: Intermediate

The determinant compresses an entire square matrix into a single number that answers two questions at once: *is this matrix invertible?* and *by how much does it scale volume?* It is less central to computation than students often think (we rarely compute large determinants), but its geometric meaning illuminates invertibility, eigenvalues, and change of variables.

4.1 The geometric definition

Definition 4.1. The **determinant** $\det(\mathbf{A})$ of a square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is the signed volume of the parallelepiped spanned by its columns. In $\mathbb{R}^2$ it is the signed area of the parallelogram formed by the two columns; in $\mathbb{R}^3$, the signed volume of the parallelepiped formed by the three columns.



Intuition

A matrix transforms the unit square (or cube) into a parallelogram (or parallelepiped); the determinant is the factor by which area/volume changes. So $\det(\mathbf{A}) = 0$ means the transformation *collapses* space into a lower dimension — the columns are linearly dependent and the matrix is not invertible. A negative determinant means the transformation flips orientation (like a mirror).

4.2 Computing determinants

For small matrices:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc, \qquad \det \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = a(ei - fh) - b(di - fg) + c(dh - eg).$$

In general, **cofactor (Laplace) expansion** along any row or column reduces an $n \times n$ determinant to a signed sum of $(n-1) \times (n-1)$ determinants:

$$\det(\mathbf{A}) = \sum_{j=1}^{n} (-1)^{i+j} \, a_{ij} \, M_{ij},$$

where M_{ij} is the minor (determinant of $\mathbf{A}$ with row i, column j deleted). This is exponential in cost, so it is for theory and small cases only.

Example 4.2. $\det \begin{bmatrix} 1 & 2 & 0 \\ 3 & -1 & 2 \\ 0 & 4 & 1 \end{bmatrix} = 1\big((-1)(1) - (2)(4)\big) - 2\big((3)(1) - (2)(0)\big) + 0 = 1(-9) - 2(3) = -15.$

4.3 Properties that make determinants useful

Proposition 4.3. For square matrices $\mathbf{A}, \mathbf{B}$ of the same size:
(1) $\det(\mathbf{I}) = 1$.
(2) Swapping two rows flips the sign; a repeated row gives $\det = 0$.
(3) Scaling one row by c scales the determinant by c; hence $\det(c\mathbf{A}) = c^n \det(\mathbf{A})$.
(4) Adding a multiple of one row to another leaves the determinant unchanged.
(5) $\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$ and $\det(\mathbf{A}^\top) = \det(\mathbf{A})$.
(6) For triangular matrices, det is the product of the diagonal entries.

Key idea

Properties (4) and (6) give the *practical* algorithm: run Gaussian elimination (which does not change the determinant except for sign-flips on row swaps), then multiply the pivots. This is how determinants are actually computed — $O(n^3)$, not the exponential cofactor sum. The product-of-pivots fact also reveals: **a matrix is invertible iff its determinant is nonzero iff it has a full set of nonzero pivots**.

4.4 Determinant, invertibility, and volume

Theorem 4.4. $\mathbf{A}$ is invertible $\iff \det(\mathbf{A}) \neq 0$. When invertible, $\det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$.

The volume interpretation also drives the **change-of-variables** formula in multivariable calculus: integrating under a transformation $\mathbf{x} \mapsto \mathbf{Ax}$ multiplies volumes by $|\det \mathbf{A}|$. The local version, with the Jacobian determinant, is what lets probability densities transform correctly — the foundation of *normalizing flows* in generative modeling.

4.5 Cramer's rule (and why we avoid it)

Cramer's rule expresses each unknown as a ratio of determinants, $x_i = \det(\mathbf{A}_i)/\det(\mathbf{A})$ where $\mathbf{A}_i$ replaces column i with $\mathbf{b}$. It is elegant and useful in proofs, but catastrophically slow and numerically poor for $n > 3$; elimination wins in practice.

Common pitfall

Determinants are rarely the right computational tool for large matrices. To test (near-)singularity, the determinant is unreliable — it can underflow or overflow wildly (e.g. a 100×100 matrix with entries ~ 2 has determinant $\sim 2^{100}$). Use the **condition number** or the smallest singular value (Chapters 10, 11) instead. For likelihoods one works with the *log*-determinant, often via a Cholesky factor.

How this powers ML / AI

The determinant appears in the multivariate Gaussian density through $\det(\boldsymbol{\Sigma})$ (the normalizing constant), and its logarithm shows up in Gaussian-process marginal likelihoods — computed stably as twice the sum of logs of a Cholesky diagonal. The Jacobian determinant is central to **normalizing flows**, which build flexible probability distributions by tracking how an invertible neural map rescales volume. And $\det = 0$ is the algebraic signature of the redundancy (collinearity) that forces regularization.

Try it in NumPy

```python
import numpy as np
A = np.array([[1.,2,0],[3,-1,2],[0,4,1]])
print(np.linalg.det(A))                  # -15.0
sign, logabsdet = np.linalg.slogdet(A)  # stable: sign and log|det|
print(sign, logabsdet)
```

Chapter summary

- $\det(\mathbf{A})$ is the signed volume-scaling factor of the transformation $\mathbf{A}$.
- $\det(\mathbf{A}) = 0 \iff$ columns are dependent $\iff \mathbf{A}$ is singular (not invertible).
- Key rules: $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$, triangular det = product of diagonal, elimination + pivots is the practical method.
- Avoid determinants for large-scale numerical work; use condition numbers / log-determinants instead.

5

Vector Spaces and the Four Fundamental Subspaces

Level: Intermediate

This is the conceptual heart of linear algebra. Everything so far — vectors, systems, matrices — now gets organized by a single powerful abstraction: the **vector space** and its **subspaces**. The payoff is the *Fundamental Theorem of Linear Algebra*, which describes the four subspaces hidden inside every matrix and how they fit together. Mastering this chapter is what separates "I can do the arithmetic" from "I understand what is going on."

5.1 Vector spaces and subspaces

Definition 5.1. A **vector space** is a set V closed under addition and scalar multiplication, obeying the usual algebraic rules (associativity, distributivity, a zero vector, additive inverses). The prototype is $\mathbb{R}^n$, but matrices, polynomials, and functions also form vector spaces.

Definition 5.2. A **subspace** of V is a nonempty subset $W \subseteq V$ that is itself a vector space: it contains $\mathbf{0}$ and is closed under linear combinations (if $\mathbf{u}, \mathbf{w} \in W$ then $c\mathbf{u}+d\mathbf{w} \in W$). Geometrically, subspaces of $\mathbb{R}^n$ are lines, planes, and hyperplanes *through the origin*.

Common pitfall

A line or plane that does *not* pass through the origin is **not** a subspace — it fails to contain $\mathbf{0}$ and is not closed under scaling. (Such "shifted" sets are *affine* sets.) Always check the origin first.

5.2 Span, independence, basis, dimension

Definition 5.3. The **span** of vectors $\mathbf{v}_1, \dots, \mathbf{v}_k$ is the set of all their linear combinations — the smallest subspace containing them. The vectors are **linearly independent** if no one of them is a linear combination of the others, equivalently if $c_1\mathbf{v}_1 + \cdots + c_k\mathbf{v}_k = \mathbf{0}$ forces all $c_i = 0$.

Definition 5.4. A **basis** of a subspace is a linearly independent set that spans it. Every basis of a given subspace has the same number of vectors; that number is the **dimension**. A basis gives every vector *unique* coordinates.

Intuition

A basis is a minimal, non-redundant coordinate system: enough vectors to reach everything (spanning), with none wasted (independent). "Dimension" is the number of independent directions — the true number of degrees of freedom, regardless of how the space is presented. In data terms, the dimension of the span of your features is how much *non-redundant* information they carry.

Example 5.5. In $\mathbb{R}^3$, the vectors $[1,0,0]^\top, [0,1,0]^\top, [1,1,0]^\top$ are *dependent* (the third is the sum of the first two) and span only the xy-plane (dimension 2). Dropping the redundant third vector leaves a basis for that plane.

5.3 The four fundamental subspaces

Every matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ gives rise to four subspaces. They are the key to understanding what $\mathbf{A}$ does and when $\mathbf{Ax} = \mathbf{b}$ is solvable.

Definition 5.6. For $\mathbf{A} \in \mathbb{R}^{m\times n}$ with rank r:

- **Column space** $\mathrm{Col}(\mathbf{A}) \subseteq \mathbb{R}^m$: all linear combinations of the columns; equivalently all outputs $\mathbf{Ax}$. Dimension r.
- **Null space** $\mathrm{null}(\mathbf{A}) \subseteq \mathbb{R}^n$: all solutions of $\mathbf{Ax} = \mathbf{0}$. Dimension $n - r$.
- **Row space** $\mathrm{Col}(\mathbf{A}^\top) \subseteq \mathbb{R}^n$: all combinations of the rows. Dimension r.
- **Left null space** $\mathrm{null}(\mathbf{A}^\top) \subseteq \mathbb{R}^m$: all $\mathbf{y}$ with $\mathbf{A}^\top\mathbf{y} = \mathbf{0}$. Dimension $m - r$.

Definition 5.7. The **rank** r is the number of pivots in the row echelon form — equivalently the dimension of the column space, and (remarkably) also of the row space. The number of independent columns equals the number of independent rows.

5.3.1 The rank–nullity theorem

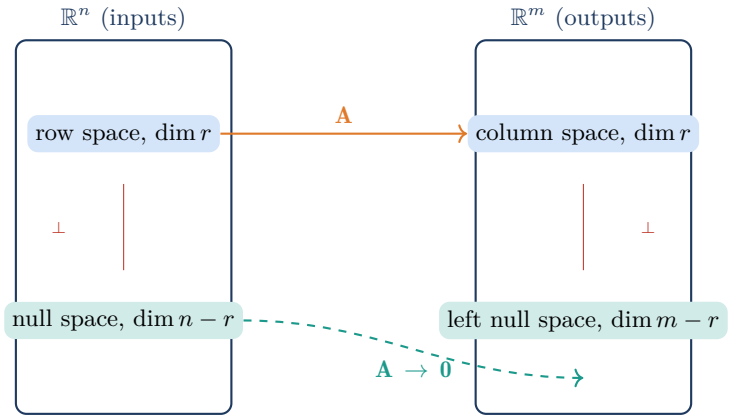
Theorem 5.8 (Rank–Nullity)**.** For $\mathbf{A} \in \mathbb{R}^{m\times n}$,

$$\mathrm{rank}(\mathbf{A}) + \dim \mathrm{null}(\mathbf{A}) = n.$$

Each of the n columns either supports a pivot (contributing to rank) or a free variable (contributing to the null space). There is no third option, so the two counts must add to n.

5.3.2 The Fundamental Theorem of Linear Algebra

Theorem 5.9 (Orthogonality of the four subspaces)**.** In $\mathbb{R}^n$, the row space and the null space are **orthogonal complements**: every row-space vector is orthogonal to every null-space vector, and together they fill $\mathbb{R}^n$. In $\mathbb{R}^m$, the column space and the left null space are orthogonal complements and together fill $\mathbb{R}^m$.



Key idea

This single picture explains a matrix completely. $\mathbf{A}$ sends the row space *bijectively* onto the column space (the "real action"), and crushes the null space to zero. The left null space is the part of the output space that $\mathbf{A}$ can never reach. Solvability of $\mathbf{Ax} = \mathbf{b}$ is now obvious: it is solvable iff $\mathbf{b}$ lies in the column space, and the solution is unique iff the null space is trivial.

Example 5.10 (Reading subspaces from RREF)**.** Let $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix}$. Row reduction gives $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \end{bmatrix}$, one pivot, so $r = 1$. The column space is the line through $[1, 2]^\top$ (all columns are multiples of it). The null space ($n - r = 2$ dimensional) is spanned by $[-2, 1, 0]^\top$ and $[-3, 0, 1]^\top$ (from the free variables). The row space is the line through $[1, 2, 3]^\top$, orthogonal to that null space.

5.4 Why the null space governs regularization

How this powers ML / AI

If a data matrix $\mathbf{X}$ has a nontrivial null space — which happens whenever there are more features than samples, or features are collinear — then $\mathbf{Xw} = \mathbf{y}$ has *infinitely many* equally good solutions: add any null-space vector to $\mathbf{w}$ and predictions on the training data are unchanged. The model is unidentifiable. **Regularization** (Ridge/Lasso) breaks the tie by selecting the minimum-norm solution from this infinite family; it is not a mere nicety but a structural necessity. The rank of $\mathbf{X}$ is the intrinsic number of independent features, and the null space is precisely the directions in which the data tells you nothing.

Try it in NumPy

```
import numpy as np
from scipy.linalg import null_space
A = np.array([[1.,2,3],[2,4,6]])
print(np.linalg.matrix_rank(A))   # 1
print(null_space(A))              # basis for the 2-D null space
# Column space basis: independent columns (here just the first column
    ).
```

Chapter summary

- Subspaces are lines/planes through the origin, closed under linear combinations.
- **Span, independence, basis, dimension** formalize "non-redundant coordinate system."
- Every matrix has four fundamental subspaces; **rank** r governs all their dimensions.
- **Rank–nullity**: $\text{rank} + \dim(\text{null}) = n$.
- Row space $\perp$ null space (in $\mathbb{R}^n$); column space $\perp$ left null space (in $\mathbb{R}^m$).
- A nontrivial null space $\Rightarrow$ non-unique solutions $\Rightarrow$ regularization is required.

6

Linear Transformations and Change of Basis

Level: Intermediate

We have treated matrices as arrays of numbers. This chapter elevates them to their true identity: matrices *are* linear transformations once a basis is fixed. This view explains why matrix multiplication is defined the way it is, what "similar matrices" means, and how changing coordinates can dramatically simplify a problem — the idea that culminates in diagonalization and the SVD.

6.1 Linear transformations

Definition 6.1. A function $T : \mathbb{R}^n \to \mathbb{R}^m$ is a **linear transformation** if it respects linear combinations:

$$T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}) \quad \text{and} \quad T(c\mathbf{v}) = cT(\mathbf{v}) \qquad \text{for all } \mathbf{u}, \mathbf{v}, c.$$

Equivalently, $T(c\mathbf{u} + d\mathbf{v}) = cT(\mathbf{u}) + dT(\mathbf{v})$. A linear map always sends $\mathbf{0}$ to $\mathbf{0}$ and sends lines to lines.

Theorem 6.2 (Every linear map is a matrix)**.** If $T : \mathbb{R}^n \to \mathbb{R}^m$ is linear, then $T(\mathbf{x}) = \mathbf{A}\mathbf{x}$ where the j-th column of $\mathbf{A}$ is $T(\mathbf{e}_j)$ — the image of the j-th standard basis vector. Conversely, every matrix defines a linear map.

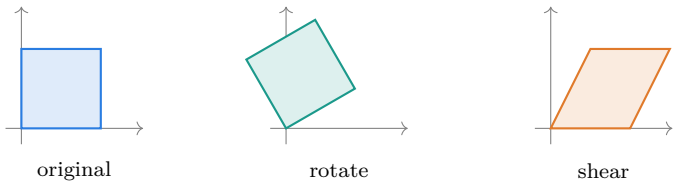
Intuition

This is profound and practical: *to know a linear transformation, you only need to know what it does to the basis vectors.* Their images become the columns of the matrix, and linearity fills in everything else. Because $T(\sum_j x_j \mathbf{e}_j) = \sum_j x_j T(\mathbf{e}_j)$, the output is the same linear combination of the columns — which is exactly the column view of $\mathbf{A}\mathbf{x}$.

Example 6.3 (Geometric transformations in $\mathbb{R}^2$)**.**

$$\text{rotate by } \theta: \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad \text{scale: } \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \quad \text{shear: } \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}, \quad \text{project onto } x: \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

Each column tells you where $\mathbf{e}_1$ and $\mathbf{e}_2$ land. The projection has determinant 0 — it collapses the plane onto a line (a nontrivial null space).



6.1.1 Kernel and image

The **kernel** (null space) of T is $\{\mathbf{x} : T(\mathbf{x}) = \mathbf{0}\}$; the **image** (range, column space) is $\{T(\mathbf{x})\}$. These are exactly two of the four fundamental subspaces, now wearing their transformation hats. T is one-to-one iff its kernel is trivial, and onto iff its image is all of $\mathbb{R}^m$.

6.2 Composition is multiplication

If $S(\mathbf{x}) = \mathbf{B}\mathbf{x}$ then $T(\mathbf{x}) = \mathbf{A}\mathbf{x}$, the composition $(T \circ S)(\mathbf{x}) = T(S(\mathbf{x})) = \mathbf{A}\mathbf{B}\mathbf{x}$. So **composing transformations corresponds to multiplying their matrices** — and this is precisely why matrix multiplication is defined by the row-times-column rule. The non-commutativity of multiplication is just the fact that "rotate then shear" differs from "shear then rotate."

6.3 Change of basis

The same vector has different coordinates in different bases, and the same transformation has different matrices. Choosing the *right* basis can turn a messy matrix into a simple one.

Definition 6.4. Let the columns of an invertible matrix $\mathbf{P}$ be a new basis (written in the standard basis). A vector with standard coordinates $\mathbf{x}$ has new coordinates $\mathbf{x}' = \mathbf{P}^{-1}\mathbf{x}$, and conversely $\mathbf{x} = \mathbf{P}\mathbf{x}'$.

Definition 6.5. Two square matrices $\mathbf{A}$ and $\mathbf{B}$ are **similar** if $\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$ for some invertible $\mathbf{P}$. Similar matrices represent the *same* linear transformation in different bases. They share determinant, trace, rank, and eigenvalues.

Key idea

Change of basis is the strategic heart of advanced linear algebra. "Diagonalization" (Chapter 8) means: find a basis of eigenvectors in which the transformation is just independent scalings — $\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$. The SVD (Chapter 10) finds *two* orthonormal bases (input and output) in which any matrix becomes diagonal. Almost every powerful factorization is the sentence "in the right coordinates, this map is simple."

How this powers ML / AI

Change of basis is what PCA does: rotate the data into the coordinate system of its principal axes, where the covariance is diagonal and the variance is concentrated in the first few coordinates. A neural network can be read as a learned sequence of changes of representation ("basis"), each layer re-expressing the data in coordinates where the next task is easier — ultimately a basis where the classes are linearly separable. The recurring engineering move "project into a better space, act, project back" is change of basis.

Try it in NumPy

```python
import numpy as np
theta = np.pi/4
R = np.array([[np.cos(theta), -np.sin(theta)],
              [np.sin(theta),  np.cos(theta)]])   # rotation
print(R @ np.array([1,0]))          # where e1 lands = first column of R
# Similarity: same map, new basis P
P = np.array([[1.,1],[0,1]])
A = np.array([[2.,0],[0,3]])
B = np.linalg.inv(P) @ A @ P        # similar to A: same eigenvalues 2,3
print(np.linalg.eigvals(B))
```

Chapter summary

- A linear map preserves linear combinations; it is fully determined by what it does to a basis.
- The columns of its matrix are the images of the basis vectors $T(\mathbf{e}_j)$.
- Composition of maps = multiplication of matrices (hence the multiplication rule and its non-commutativity).
- **Change of basis** re-expresses vectors and maps; **similar** matrices ($\mathbf{B} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$) are one map in different coordinates.
- "In the right basis, the map is simple" previews diagonalization and the SVD.

7

Orthogonality, Projections, and Least Squares

Level: Intermediate

Orthogonality is where linear algebra becomes a tool for data. When an exact solution to $\mathbf{Ax} = \mathbf{b}$ does not exist — the everyday situation with real, noisy, over-determined data — orthogonality tells us how to find the *best* approximate solution. This chapter builds projections, the normal equations of least squares, the Gram–Schmidt process, and the QR factorization: the machinery directly behind linear regression.

7.1 Orthogonal and orthonormal sets

Definition 7.1. A set of vectors is **orthogonal** if every pair has zero dot product, and **orthonormal** if in addition each is a unit vector. A square matrix $\mathbf{Q}$ whose columns are orthonormal satisfies $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$ and is called an **orthogonal matrix**; it preserves lengths and angles ($\|\mathbf{Qx}\| = \|\mathbf{x}\|$).

Intuition

Orthonormal bases are the best coordinate systems in existence. In one, computing a vector's coordinates is just taking dot products ($x_i = \mathbf{q}_i^\top\mathbf{x}$), there is no distortion of length or angle, and the inverse of the basis matrix is simply its transpose. Numerically they never amplify error. Much of advanced linear algebra is an effort to *manufacture* orthonormal bases (Gram–Schmidt, QR, eigenvectors of symmetric matrices, the SVD).

7.2 Orthogonal complements

The **orthogonal complement** $W^\perp$ of a subspace W is the set of all vectors orthogonal to everything in W. The Fundamental Theorem (Chapter 5) said it already: row space and null space are orthogonal complements in $\mathbb{R}^n$; column space and left null space are orthogonal complements in $\mathbb{R}^m$. Any vector splits uniquely into a part in W and a part in $W^\perp$.

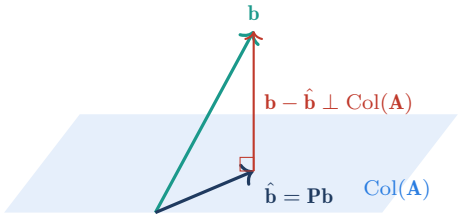
7.3 Projection onto a subspace

Generalizing the projection-onto-a-line from Chapter 1: given a subspace spanned by the columns of $\mathbf{A}$, the projection of $\mathbf{b}$ onto that column space is the closest point in it.

Theorem 7.2 (Projection matrix)**.** If $\mathbf{A}$ has independent columns, the orthogonal projection of $\mathbf{b}$ onto $\mathrm{Col}(\mathbf{A})$ is

$$\hat{\mathbf{b}} = \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b} = \mathbf{P}\mathbf{b}, \qquad \mathbf{P} = \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top.$$

The projection matrix $\mathbf{P}$ satisfies $\mathbf{P}^\top = \mathbf{P}$ and $\mathbf{P}^2 = \mathbf{P}$ (projecting twice changes nothing), and the residual $\mathbf{b} - \hat{\mathbf{b}}$ is orthogonal to $\mathrm{Col}(\mathbf{A})$.



7.4 Least squares: the normal equations

When $\mathbf{A}\mathbf{x} = \mathbf{b}$ has no solution (more equations than unknowns, $\mathbf{b} \notin \mathrm{Col}(\mathbf{A})$), we instead minimize the squared error $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$. The minimizer makes the residual orthogonal to the column space, which gives the celebrated **normal equations**.

Theorem 7.3 (Least squares)**.** The vector $\hat{\mathbf{x}}$ minimizing $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$ solves

$$\boxed{\mathbf{A}^\top\mathbf{A}\,\hat{\mathbf{x}} = \mathbf{A}^\top\mathbf{b}} \quad \Longrightarrow \quad \hat{\mathbf{x}} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$$

(the last step assuming $\mathbf{A}$ has independent columns). The fitted values are $\hat{\mathbf{b}} = \mathbf{A}\hat{\mathbf{x}} = \mathbf{P}\mathbf{b}$.

Key idea

Fitting a linear model is projection. The achievable predictions form the column space of the design matrix; the targets $\mathbf{b}$ usually lie outside it; least squares drops a perpendicular onto that subspace. The orthogonality of the residual is not a trick — it is the geometric definition of "best fit." This single picture is the foundation of linear regression, and via the pseudoinverse (Chapter 10) of much more.

Example 7.4 (Fitting a line)**.** To fit $y = c + dx$ through points (x_i, y_i), set $\mathbf{A} = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$, and solve $\mathbf{A}^\top\mathbf{A}\,\hat{\mathbf{x}} = \mathbf{A}^\top\mathbf{b}$. The 2×2 system encodes the familiar slope/intercept formulas of statistics — here derived purely geometrically.

Common pitfall

Forming and solving $\mathbf{A}^\top\mathbf{A}$ directly (the textbook normal equations) *squares the condition number* of $\mathbf{A}$, losing up to twice as many digits of precision. With nearly-collinear features this is numerically dangerous. Production solvers therefore fit least squares via **QR** or **SVD** of $\mathbf{A}$ directly — never by inverting $\mathbf{A}^\top\mathbf{A}$. Use `np.linalg.lstsq`, not `inv(A.T@A)@A.T@b`.

7.5 Gram–Schmidt and the QR factorization

Given independent (but not orthogonal) vectors, the **Gram–Schmidt process** manufactures an orthonormal basis for the same span: take each vector, subtract its projections onto the directions already fixed, and normalize.

Definition 7.5. Applying Gram–Schmidt to the columns of $\mathbf{A}$ yields the **QR factorization**

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ has orthonormal columns ($\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$) spanning $\mathrm{Col}(\mathbf{A})$ and $\mathbf{R}$ is upper-triangular. Least squares then reduces to the well-conditioned triangular solve $\mathbf{R}\hat{\mathbf{x}} = \mathbf{Q}^\top\mathbf{b}$.

Intuition

QR is "orthogonalize the columns and remember how." Because $\mathbf{Q}$ is perfectly conditioned, QR is the numerically stable way to solve least-squares problems and a building block of eigenvalue algorithms (the QR algorithm). In practice the modified Gram–Schmidt or Householder reflections are used for stability.

How this powers ML / AI

Least squares *is* linear regression, and the projection view explains regularization geometrically. **Orthogonality** also underlies whitening/decorrelation of features, orthogonal weight initialization (which preserves signal norms across deep layers and combats vanishing/exploding gradients), and the orthonormal bases that make PCA and the SVD numerically trustworthy. When you call a regression or PCA routine, QR or SVD is almost certainly running underneath.

Try it in NumPy

```
import numpy as np
A = np.c_[np.ones(50), np.linspace(0,5,50)]      # design matrix (1, x)
b = 2.0 + 1.5*A[:,1] + 0.3*np.random.randn(50)   # noisy line
x_hat, *_ = np.linalg.lstsq(A, b, rcond=None)    # stable least squares
Q, R = np.linalg.qr(A)
x_qr = np.linalg.solve(R, Q.T @ b)               # same answer via QR
print(x_hat, x_qr)
```

Chapter summary

- Orthonormal sets are the ideal coordinates: coordinates by dot products, inverse by transpose, no distortion.
- Projection onto $\mathrm{Col}(\mathbf{A})$ is $\mathbf{Pb} = \mathbf{A}(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$; the residual is orthogonal to the subspace.
- **Least squares** solves $\mathbf{A}^\top\mathbf{A}\hat{\mathbf{x}} = \mathbf{A}^\top\mathbf{b}$: best fit = projection. This is linear regression.
- **Gram–Schmidt** → **QR**; fit least squares via QR/SVD, never by inverting $\mathbf{A}^\top\mathbf{A}$.

PART III

Advanced Topics

8

Eigenvalues and Eigenvectors

Level: Advanced

Eigenvalues reveal the hidden axes of a transformation — the special directions a matrix merely stretches without rotating. They explain the long-term behavior of dynamical systems, the principal directions of data, the curvature of loss surfaces, and the ranking behind PageRank. This chapter develops eigenvalues, eigenvectors, and diagonalization; the next specializes to the beautifully behaved symmetric case.

8.1 The eigenvalue equation

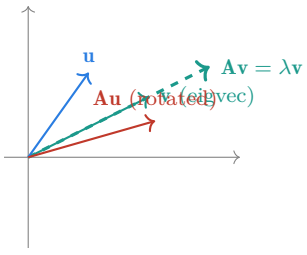
Definition 8.1. For a square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, a nonzero vector $\mathbf{v}$ is an **eigenvector** with **eigenvalue** λ if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

Along $\mathbf{v}$, the transformation acts as pure scaling by λ — no change of direction (unless $\lambda < 0$, which flips it).

Intuition

Most vectors are knocked off their line when you apply $\mathbf{A}$. Eigenvectors are the rare directions that stay on their own line; the eigenvalue says by how much they stretch. These special directions are the "natural axes" of the matrix — in the eigenvector basis, the tangled action of $\mathbf{A}$ becomes a simple set of independent scalings. That is the whole motivation for diagonalization.



8.2 Finding eigenvalues: the characteristic polynomial

Rewrite $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ as $(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0}$. A nonzero $\mathbf{v}$ exists exactly when $\mathbf{A} - \lambda\mathbf{I}$ is singular:

Theorem 8.2. The eigenvalues of $\mathbf{A}$ are the roots of the **characteristic polynomial**

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

For each eigenvalue λ, the corresponding eigenvectors span the **eigenspace** $\text{null}(\mathbf{A} - \lambda\mathbf{I})$.

Example 8.3. For $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, $\det(\mathbf{A}-\lambda\mathbf{I}) = (2-\lambda)^2-1 = \lambda^2-4\lambda+3 = (\lambda-1)(\lambda-3)$. So $\lambda_1 = 3$ with eigenvector $[1,1]^\top$ and $\lambda_2 = 1$ with eigenvector $[1,-1]^\top$. The matrix stretches by 3 along the diagonal and by 1 perpendicular to it.

Proposition 8.4 (Trace and determinant)**.** The sum of the eigenvalues equals the trace, $\sum_i \lambda_i = \text{tr}(\mathbf{A})$, and their product equals the determinant, $\prod_i \lambda_i = \det(\mathbf{A})$. Hence $\mathbf{A}$ is invertible iff no eigenvalue is zero.

8.3 Diagonalization

Theorem 8.5 (Diagonalization)**.** If $\mathbf{A} \in \mathbb{R}^{n\times n}$ has n linearly independent eigenvectors, gather them as the columns of $\mathbf{P}$ and the eigenvalues into a diagonal $\mathbf{D}$. Then

$$\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}, \qquad \mathbf{D} = \text{diag}(\lambda_1, \ldots, \lambda_n).$$

$\mathbf{A}$ is then called **diagonalizable**: in the eigenvector basis it is just the diagonal scaling $\mathbf{D}$.

Key idea

Diagonalization makes matrix powers trivial: $\mathbf{A}^k = \mathbf{P}\mathbf{D}^k\mathbf{P}^{-1}$, and $\mathbf{D}^k$ just raises each eigenvalue to the k. This is why eigenvalues control *long-term behavior*. Powering a matrix repeatedly (a Markov chain, a recurrent network, an iterative solver) amplifies directions with $|\lambda| > 1$ and kills those with $|\lambda| < 1$; the largest-magnitude eigenvalue eventually dominates.

Example 8.6 (Powers and dynamics)**.** A system $\mathbf{x}_{t+1} = \mathbf{A}\mathbf{x}_t$ evolves as $\mathbf{x}_t = \mathbf{A}^t\mathbf{x}_0 = \mathbf{P}\mathbf{D}^t\mathbf{P}^{-1}\mathbf{x}_0 = \sum_i c_i\lambda_i^t\mathbf{v}_i$. If one $|\lambda_i| > 1$, that mode blows up; if all $|\lambda_i| < 1$, the system decays to zero; the steady state corresponds to $\lambda = 1$.

Common pitfall

Not every matrix is diagonalizable. *Defective* matrices (e.g. $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$) lack a full set of independent eigenvectors. Also, real matrices can have *complex* eigenvalues (rotations: $\left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$ has eigenvalues $\pm i$). Both problems vanish for *symmetric* matrices (Chapter 9) — which is one reason symmetric matrices are so beloved in ML. When diagonalization fails, the SVD (Chapter 10) always works.

8.4 Applications: Markov chains and PageRank

A **Markov matrix** (nonnegative columns summing to 1) always has $\lambda = 1$ as its largest eigenvalue; the corresponding eigenvector is the **stationary distribution** the chain converges to. Google's original **PageRank** is exactly this: model a random web surfer as a Markov chain over links, and rank pages by the stationary distribution — the dominant eigenvector of the link matrix, found by the *power method* (repeated multiplication).

How this powers ML / AI

Eigen-thinking is everywhere in ML. The eigenvectors of a **covariance matrix** are the principal components, and the eigenvalues are the variances along them (PCA, Chapter 9). The eigenvalues of the **Hessian** of a loss describe its curvature: their ratio (the condition number) sets how badly gradient descent zig-zags and how small the learning rate must be. **Spectral clustering** and graph neural networks use eigenvectors of the graph Laplacian. PageRank and many recommendation/ranking methods are dominant-eigenvector computations.

Try it in NumPy

```python
import numpy as np
A = np.array([[2.,1],[1,2]])
vals, vecs = np.linalg.eig(A)          # eigenvalues, eigenvectors (columns)
print(vals)                            # [3., 1.]
# Reconstruct A = P D P^{-1}
P, D = vecs, np.diag(vals)
print(np.allclose(P @ D @ np.linalg.inv(P), A))   # True
# Power method for the dominant eigenvector:
v = np.random.randn(2)
for _ in range(100): v = A @ v; v /= np.linalg.norm(v)
print(v)                               # aligns with eigenvector for lambda=3
```

Chapter summary

- Eigenvectors are directions scaled (not rotated) by $\mathbf{A}$; eigenvalues are the scale factors: $\mathbf{Av} = \lambda\mathbf{v}$.
- Eigenvalues are roots of $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$; $\sum \lambda_i = \text{tr}$, $\prod \lambda_i = \det$.
- Diagonalization $\mathbf{A} = \mathbf{PDP}^{-1}$ makes powers and dynamics easy; the dominant eigenvalue governs long-term behavior.
- Not all matrices diagonalize (defective or complex eigenvalues) — but symmetric ones always do, and the SVD always exists.
- Eigen-methods power PCA, Hessian/curvature analysis, spectral clustering, and PageRank.

9

Symmetric Matrices, the Spectral Theorem, and Quadratic Forms

Level: Advanced

Symmetric matrices ($\mathbf{A}^\top = \mathbf{A}$) are the best-behaved objects in linear algebra, and — not coincidentally — the ones that appear most in machine learning: covariance matrices, Gram matrices, kernel matrices, and Hessians are all symmetric. They have real eigenvalues, orthogonal eigenvectors, and an especially clean diagonalization (the spectral theorem). This chapter also introduces quadratic forms and positive definiteness, the language of optimization landscapes.

9.1 The spectral theorem

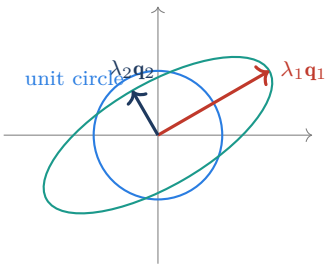
Theorem 9.1 (Spectral theorem)**.** Every real symmetric matrix $\mathbf{A}$ can be diagonalized by an **orthogonal** matrix:

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top, \qquad \mathbf{Q}^\top\mathbf{Q} = \mathbf{I}, \quad \mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_n).$$

All eigenvalues λ_i are *real*, and the eigenvectors (columns of $\mathbf{Q}$) can be chosen *orthonormal*. Equivalently $\mathbf{A} = \sum_i \lambda_i \, \mathbf{q}_i \mathbf{q}_i^\top$, a sum of scaled orthogonal projections.

Intuition

The spectral theorem says a symmetric matrix is, in the right orthonormal coordinates, just a set of independent stretches along perpendicular axes — no shear, no rotation, no complex numbers. Geometrically it maps the unit sphere to an ellipsoid whose axes are the eigenvectors and whose semi-axis lengths are the eigenvalues. This is the cleanest possible structure, and it is why ML leans so heavily on symmetric matrices.



9.2 Quadratic forms

A **quadratic form** is a function $Q(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x}$ with $\mathbf{A}$ symmetric — a pure "second-degree" function of the coordinates, with no linear or constant term. Examples: $\|\mathbf{x}\|^2 = \mathbf{x}^\top \mathbf{I}\mathbf{x}$, and the exponent of a multivariate Gaussian.

Using the spectral theorem and the substitution $\mathbf{y} = \mathbf{Q}^\top \mathbf{x}$,

$$Q(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x} = \sum_i \lambda_i \, y_i^2.$$

So in the eigenvector coordinates a quadratic form is just a weighted sum of squares — the eigenvalues are the weights, and their signs determine its shape.

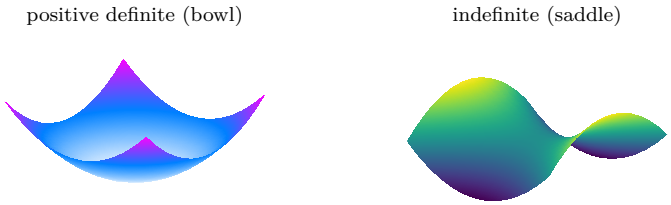
9.3 Positive definiteness

Definition 9.2. A symmetric matrix $\mathbf{A}$ is

- **positive definite** if $\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$ (all $\lambda_i > 0$);
- **positive semidefinite (PSD)** if $\mathbf{x}^\top \mathbf{A}\mathbf{x} \geq 0$ (all $\lambda_i \geq 0$);
- **indefinite** if it takes both signs (mixed-sign eigenvalues).

Intuition

Positive definiteness is the matrix version of "positive number," and it is the multivariable test for a minimum. A quadratic form is a bowl (unique minimum) iff its matrix is positive definite, a dome (maximum) iff negative definite, and a saddle iff indefinite. For a loss function, the Hessian's definiteness at a critical point tells you whether it is a minimum, maximum, or saddle — the central question of optimization.



Proposition 9.3 (Tests for positive definiteness)**.** For symmetric $\mathbf{A}$, the following are equivalent: (1) all eigenvalues positive; (2) all leading principal minors positive (Sylvester's criterion); (3) all pivots positive in elimination; (4) $\mathbf{A} = \mathbf{B}^\top \mathbf{B}$ for some $\mathbf{B}$ with independent columns. Test (4) is why $\mathbf{A}^\top \mathbf{A}$ and covariance matrices are always at least PSD.

9.4 The Rayleigh quotient

For symmetric $\mathbf{A}$, the **Rayleigh quotient** $R(\mathbf{x}) = \dfrac{\mathbf{x}^\top \mathbf{A}\mathbf{x}}{\mathbf{x}^\top \mathbf{x}}$ is bounded by the extreme eigenvalues:

$$\lambda_{\min} \leq R(\mathbf{x}) \leq \lambda_{\max},$$

with the maximum attained at the top eigenvector. This variational characterization — "the top eigenvector maximizes $\mathbf{x}^\top \mathbf{A}\mathbf{x}$ on the unit sphere" — is exactly the optimization PCA solves, and it underlies the power method and many spectral algorithms.

9.5 Covariance, Gram, and kernel matrices

How this powers ML / AI

The symmetric matrices of ML are all here. The **covariance matrix** $\boldsymbol{\Sigma} = \frac{1}{n}\mathbf{X}^\top\mathbf{X}$ (for centered data) is symmetric PSD; its spectral decomposition *is* PCA — eigenvectors are principal directions, eigenvalues are variances, and the Rayleigh-quotient view says the first PC maximizes explained variance. **Gram matrices** $\mathbf{X}\mathbf{X}^\top$ and **kernel matrices** $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ are symmetric PSD and power kernel methods, SVMs, and Gaussian processes. The **Hessian** of a loss is symmetric; its eigenvalues (curvature) determine convergence speed and whether a critical point is a minimum or a saddle — the latter being the norm in high-dimensional deep learning.

Try it in NumPy

```python
import numpy as np
X = np.random.randn(200, 3) @ np.array([[3,1,0],[0,2,0],[0,0,0.5]])
Xc = X - X.mean(0)
C = (Xc.T @ Xc) / len(Xc)                # symmetric PSD covariance
vals, vecs = np.linalg.eigh(C)           # eigh: for symmetric matrices (real, sorted)
print(vals[::-1])                        # variances along principal components
# PCA: project onto top-2 eigenvectors (largest eigenvalues)
PCs = vecs[:, ::-1][:, :2]
Z = Xc @ PCs
```

Common pitfall

For symmetric matrices use `np.linalg.eigh`, not `eig`: `eigh` exploits symmetry to return real, sorted eigenvalues with orthonormal eigenvectors, and is faster and more accurate. Using the general `eig` can return tiny spurious imaginary parts from round-off.

Chapter summary

- **Spectral theorem**: symmetric $\mathbf{A} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top$ with real eigenvalues and orthonormal eigenvectors.
- A quadratic form $\mathbf{x}^\top\mathbf{A}\mathbf{x} = \sum_i \lambda_i y_i^2$ in eigen-coordinates; eigenvalue signs set its shape.
- **Positive definite** $\Leftrightarrow$ all $\lambda_i > 0$ $\Leftrightarrow$ a bowl with a unique minimum; the Hessian test for optimization.
- The Rayleigh quotient's maximum is the top eigenvector — the optimization behind PCA.
- Covariance, Gram, kernel, and Hessian matrices are all symmetric (P)SD — the workhorses of ML.

10

The Singular Value Decomposition

Level: Advanced

If one theorem deserves the title "fundamental theorem of data science," it is the singular value decomposition. The SVD factors *any* matrix — square or rectangular, full-rank or not — into a rotation, a scaling, and another rotation. It generalizes eigendecomposition to all matrices, exposes the four fundamental subspaces, gives the best low-rank approximation, defines the pseudoinverse, and underlies PCA, recommender systems, latent semantic analysis, and low-rank fine-tuning of large models. This is the summit of the core theory.

10.1 The decomposition

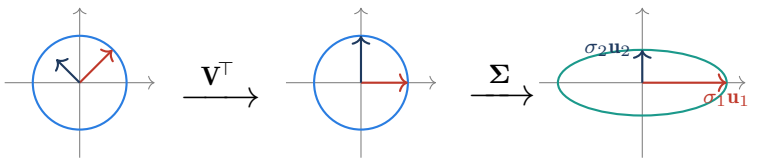
Theorem 10.1 (Singular Value Decomposition)**.** Every matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ factors as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top,$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are **orthogonal** (columns orthonormal), and $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ is "diagonal" with nonnegative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ called the **singular values**. The columns of $\mathbf{U}$ are the **left singular vectors**, those of $\mathbf{V}$ the **right singular vectors**.

Rotate, stretch, rotate

Read $\mathbf{Ax} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top\mathbf{x}$ right to left: $\mathbf{V}^\top$ *rotates* the input to align with the right singular vectors, $\mathbf{\Sigma}$ *stretches* each axis by a singular value (and changes dimension), and $\mathbf{U}$ *rotates* the result into the output space. **Every linear map, no matter how complicated, is just a rotation, an axis-aligned scaling, and another rotation.** The SVD finds the two special orthonormal bases (one for the input space, one for the output) in which the matrix is diagonal.



unit circle, $\mathbf{v}_i$ axis-aligned ellipse (then $\mathbf{U}$ rotates)

10.2 Relation to eigendecomposition

The SVD is built from two symmetric matrices you already understand:

$$\mathbf{A}^\top\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^\top\mathbf{\Sigma}\,\mathbf{V}^\top, \qquad \mathbf{A}\mathbf{A}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^\top\mathbf{U}^\top.$$

So the right singular vectors $\mathbf{V}$ are the eigenvectors of $\mathbf{A}^\top\mathbf{A}$, the left singular vectors $\mathbf{U}$ are the eigenvectors of $\mathbf{A}\mathbf{A}^\top$, and the **singular values are the square roots of the (shared, nonnegative) eigenvalues**: $\sigma_i = \sqrt{\lambda_i}$. Because $\mathbf{A}^\top\mathbf{A}$ is symmetric PSD, this always exists — which is why the SVD exists for every matrix even when eigendecomposition fails.

10.3 The four subspaces, revealed

The SVD lays out all four fundamental subspaces with orthonormal bases. For rank r (the number of nonzero singular values):

Subspace	Orthonormal basis from SVD
Column space $\mathrm{Col}(\mathbf{A})$	first r columns of $\mathbf{U}$
Left null space $\mathrm{null}(\mathbf{A}^\top)$	last $m-r$ columns of $\mathbf{U}$
Row space $\mathrm{Col}(\mathbf{A}^\top)$	first r columns of $\mathbf{V}$
Null space $\mathrm{null}(\mathbf{A})$	last $n-r$ columns of $\mathbf{V}$

The number of nonzero singular values *is* the rank — and, numerically, the number of singular values above a small threshold is the practical ("numerical") rank.

10.4 Low-rank approximation: the Eckart–Young theorem

Writing the SVD as a sum of rank-one pieces,

$$\mathbf{A} = \sum_{i=1}^{r} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top,$$

and truncating after k terms gives $\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top$. This is the *best possible* rank-k approximation.

Theorem 10.2 (Eckart–Young–Mirsky)**.** Among all matrices $\mathbf{B}$ of rank at most k, the truncated SVD $\mathbf{A}_k$ minimizes the approximation error, and

$$\|\mathbf{A} - \mathbf{A}_k\|_2 = \sigma_{k+1}, \qquad \|\mathbf{A} - \mathbf{A}_k\|_F = \sqrt{\sigma_{k+1}^2 + \cdots + \sigma_r^2}.$$

The proof rests on the orthogonal invariance of these norms: multiplying by an orthogonal matrix does not change them, so the problem reduces to the transparent diagonal case.

Key idea

This one theorem is the mathematical core of data compression and dimensionality reduction. "Keep the largest singular values, discard the small ones" is provably the optimal way to approximate a matrix with a simpler one. The discarded tail (small σ_i) is usually noise; the retained head captures the signal. PCA, image compression, recommender systems, latent semantic analysis, and embedding compression are all this single move.

Example 10.3 (Image compression)**.** A grayscale image is a matrix of pixel intensities with a quickly-decaying singular-value spectrum. Reconstructing from the top k singular triplets stores only $k(m+n+1)$ numbers instead of mn, yet looks almost identical for modest k — a vivid demonstration that real data lives near a low-dimensional subspace.

10.5 The pseudoinverse

The SVD defines an inverse for *any* matrix, even non-square or singular ones.

Definition 10.4. The **Moore–Penrose pseudoinverse** of $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ is

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^\top,$$

where $\mathbf{\Sigma}^+$ inverts each nonzero singular value (and transposes the shape). The pseudoinverse gives the *minimum-norm least-squares* solution $\hat{\mathbf{x}} = \mathbf{A}^+\mathbf{b}$ of $\mathbf{A}\mathbf{x} = \mathbf{b}$: when there is no exact solution it returns the least-squares fit, and when there are infinitely many it returns the smallest one.

Intuition

The pseudoinverse unifies the whole book. For an invertible square matrix it is the ordinary inverse. For a tall full-rank matrix it equals $(\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top$ — the least-squares operator of Chapter 7. For a wide or rank-deficient matrix it picks the minimum-norm solution from the infinite family — exactly what L2 regularization approaches as $\lambda \to 0$. One formula, every case.

Common pitfall

Compute least squares and the pseudoinverse via the SVD (`np.linalg.lstsq`, `np.linalg.pinv`), not by forming $(\mathbf{A}^\top\mathbf{A})^{-1}$. Forming $\mathbf{A}^\top\mathbf{A}$ *squares* the condition number σ_1/σ_r, so you lose up to twice as many digits of accuracy. This is the same reason production PCA uses the SVD of the data matrix rather than the eigendecomposition of the covariance matrix.

How this powers ML / AI

The SVD is the most useful single tool in applied linear algebra for ML. **PCA** is the SVD of centered data: principal components are right singular vectors, explained variance comes from σ_i^2. **Recommender systems** factor the user–item matrix as a low-rank product (the Netflix-Prize approach). **Latent semantic analysis** applies truncated SVD to term–document matrices. **Embedding compression** for vector databases is a truncated SVD/PCA step before quantization (8–32× smaller). And **LoRA**, the dominant way to fine-tune large language models, freezes the pretrained weights and learns a low-rank update $\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$ — a direct bet that the needed change is low-rank, exactly the Eckart–Young intuition.

Try it in NumPy

```
import numpy as np
A = np.random.randn(8, 5)
U, s, Vt = np.linalg.svd(A, full_matrices=False)   # reduced SVD
print(np.allclose(A, (U * s) @ Vt))                # True
k = 2
A_k = (U[:, :k] * s[:k]) @ Vt[:k, :]               # best rank-2
    approx
print(np.linalg.norm(A - A_k, 2), s[k])            # equals sigma_{k
    +1}
```

```
x = np.linalg.pinv(A) @ np.random.randn(8)          # min-norm least
    squares
```

Chapter summary

- Every matrix factors as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$: rotate, stretch by singular values, rotate.
- $\mathbf{V}$ are eigenvectors of $\mathbf{A}^\top\mathbf{A}$, $\mathbf{U}$ of $\mathbf{A}\mathbf{A}^\top$, and $\sigma_i = \sqrt{\lambda_i}$; the SVD always exists.
- Nonzero singular values = rank; $\mathbf{U}, \mathbf{V}$ give orthonormal bases for all four subspaces.
- **Eckart–Young**: the truncated SVD $\mathbf{A}_k$ is the best rank-k approximation, with error σ_{k+1}.
- The **pseudoinverse** $\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^\top$ gives minimum-norm least squares — one formula for all cases.
- SVD powers PCA, recommender systems, LSA, embedding compression, and LoRA.

11

Matrix Decompositions, Norms, and Numerical Linear Algebra

Level: Expert

On a real computer, linear algebra is done in finite-precision arithmetic, where the order of operations and the conditioning of a problem decide how many correct digits survive. This chapter consolidates the major matrix factorizations, introduces matrix norms and the all-important condition number, and surveys the algorithms that power `numpy`, `scipy`, BLAS, and LAPACK — the libraries every ML framework calls.

11.1 A field guide to the decompositions

Each factorization expresses a matrix as a product of structured pieces, trading a hard problem for easy ones (triangular or diagonal). Choosing the right one is the essence of practical numerical work.

Factorization	Form	Use it for
LU (with pivoting)	$\mathbf{PA} = \mathbf{LU}$	solving square systems $\mathbf{Ax} = \mathbf{b}$; determinants
Cholesky	$\mathbf{A} = \mathbf{LL}^\top$	symmetric positive-definite systems (2× faster); GP/covariance; sampling Gaussians
QR	$\mathbf{A} = \mathbf{QR}$	least squares; orthonormal bases; eigenvalue iteration
Eigen	$\mathbf{A} = \mathbf{PDP}^{-1}$	square matrices; dynamics, powers, PCA via covariance
Spectral	$\mathbf{A} = \mathbf{Q\Lambda Q}^\top$	symmetric matrices; PCA, kernels, Hessians
SVD	$\mathbf{A} = \mathbf{U\Sigma V}^\top$	*everything*: any matrix, rank, low-rank, pseudoinverse

Key idea

A practical decision rule. Square and generic → **LU**. Symmetric positive definite → **Cholesky** (the fastest and most stable; it also fails precisely when the matrix is not positive definite, a useful test). Rectangular/least squares → **QR**. Symmetric and you need eigen-structure → **spectral (eigh)**. Anything else, or when in doubt, or when rank/sta-

bility matters → **SVD**. The SVD is the most informative and most stable, at the highest cost.

11.1.1 Cholesky in one line

For symmetric positive-definite $\mathbf{A}$, $\mathbf{A} = \mathbf{L}\mathbf{L}^\top$ with $\mathbf{L}$ lower-triangular. It uses half the work of LU, is very stable, and gives the log-determinant for free as $2\sum_i \log L_{ii}$ — which is why it is the engine of Gaussian-process likelihoods and of drawing samples from a multivariate Gaussian ($\mathbf{x} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ with $\mathbf{z}$ standard normal).

11.2 Vector and matrix norms

Norms measure size and, crucially, how errors propagate. Beyond the vector p-norms (Chapter 1), the key matrix norms are:

- **Spectral norm** $\|\mathbf{A}\|_2 = \sigma_1$, the largest singular value — the maximum factor by which $\mathbf{A}$ stretches any vector.
- **Frobenius norm** $\|\mathbf{A}\|_F = \sqrt{\sum_{ij} a_{ij}^2} = \sqrt{\sum_i \sigma_i^2}$ — the Euclidean norm of the flattened matrix.
- **Nuclear norm** $\|\mathbf{A}\|_* = \sum_i \sigma_i$ — the convex surrogate for rank, used to encourage low-rank solutions (matrix completion).

Intuition

The spectral norm is an "operator" norm: it answers "what is the worst-case amplification of $\mathbf{A}$?" The Frobenius norm treats the matrix as a long vector and is what appears in most ML loss functions (e.g. reconstruction error). The nuclear norm, being the sum of singular values, plays the role for rank that the L1 norm plays for sparsity — minimizing it tends to produce low-rank matrices.

11.3 The condition number

Definition 11.1. The **condition number** of an invertible matrix (in the 2-norm) is

$$\operatorname{cond}(\mathbf{A}) = \|\mathbf{A}\|_2 \, \left\|\mathbf{A}^{-1}\right\|_2 = \frac{\sigma_{\max}}{\sigma_{\min}}.$$

It measures how much relative error in $\mathbf{b}$ (or in $\mathbf{A}$) can be amplified in the solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$. A small condition number (≈ 1) is *well-conditioned*; a huge one is *ill-conditioned*.

Key idea

The condition number is the most important number in numerical linear algebra. Rule of thumb: if $\operatorname{cond}(\mathbf{A}) \approx 10^k$, you can lose about k digits of accuracy when solving with $\mathbf{A}$. In double precision (about 16 digits), a condition number of 10^{16} means *no* correct digits. Ill-conditioning — not just singularity — is what makes problems hard, and it is why we avoid forming $\mathbf{A}^\top\mathbf{A}$ (which squares the condition number) and prefer QR/SVD.

Common pitfall

A matrix can be "technically invertible" yet useless: a tiny nonzero $\sigma_{\min}$ gives an enormous condition number, so the computed inverse and solution are dominated by round-off. Never test singularity with the determinant; check $\sigma_{\min}$ or the condition number. In ML, an ill-conditioned data matrix (from collinear features) both destabilizes the fit and signals that regularization is needed.

11.4 Algorithms behind the libraries

- **Direct solvers** (LU, Cholesky, QR) compute an exact answer in $O(n^3)$ up to round-off; ideal for dense, moderate-size systems.
- **Iterative solvers** (conjugate gradient for symmetric PD systems; GMRES for general) only multiply by $\mathbf{A}$ and converge gradually — essential for the huge, sparse systems of scientific computing and some ML.
- **The power method** and its refinements (Lanczos, Arnoldi, randomized SVD) find a few dominant eigen/singular vectors of enormous matrices without factoring the whole thing — exactly what PCA on big data and PageRank need.

Intuition

Numerical linear algebra is the invisible foundation under every ML framework: `numpy` and `scipy` dispatch to BLAS/LAPACK, decades-old, exquisitely tuned Fortran/C kernels. Matrix multiply (GEMM) is so heavily optimized that the practical advice is always "express your computation as big matrix multiplications," which is precisely how GPUs accelerate deep learning. You rarely write these algorithms, but knowing which one runs — and its conditioning behavior — is what separates robust code from mysterious numerical bugs.

How this powers ML / AI

Conditioning explains real training pathologies. The condition number of the loss Hessian sets gradient descent's zig-zag and the maximum stable learning rate (Chapter 8); normalization layers and good initialization improve conditioning. Collinear features make $\mathbf{X}^\top\mathbf{X}$ ill-conditioned, motivating Ridge regularization, which *adds* $\lambda\mathbf{I}$ and so raises $\sigma_{\min}$ and lowers the condition number. Cholesky drives Gaussian processes; randomized SVD makes PCA feasible on web-scale matrices; iterative solvers appear inside second-order optimizers. Stability is not a footnote — it decides whether training converges.

Try it in NumPy

```
import numpy as np
A = np.array([[1.0, 1.0], [1.0, 1.0001]])    # nearly singular
print(np.linalg.cond(A))                       # ~ 4e4  (ill-
    conditioned)
s = np.linalg.svd(A, compute_uv=False)
print(s[0] / s[-1])                            # same as cond(A)
# Cholesky for an SPD system, and a stable log-determinant:
M = A.T @ A + 1e-3*np.eye(2)                   # SPD
L = np.linalg.cholesky(M)
logdet = 2*np.sum(np.log(np.diag(L)))
```

Chapter summary

- Pick the factorization by structure: LU (square), Cholesky (SPD), QR (least squares), spectral (symmetric), SVD (anything/rank/stability).
- Matrix norms: spectral (σ_1, worst-case stretch), Frobenius (flattened L2), nuclear ($\sum \sigma_i$, low-rank surrogate).
- The **condition number** $\sigma_{\max}/\sigma_{\min}$ predicts digit loss; ill-conditioning, not just singularity, is what makes problems hard.
- Libraries call BLAS/LAPACK; express computation as matrix multiplies; prefer stable factorizations over explicit inverses.

12

Tensors and Matrix Calculus

Level: Expert

Two ideas carry linear algebra the last mile into deep learning. First, **tensors** generalize vectors and matrices to higher-order arrays — the natural container for images, sequences, and batches. Second, **matrix calculus** extends derivatives to vector- and matrix-valued functions, giving the gradients and Jacobians that backpropagation multiplies together. This chapter is the bridge from "linear algebra" to "how neural networks actually train."

12.1 Tensors: arrays of any order

Definition 12.1. A **tensor** (in the computational sense) is an n-dimensional array. Order 0 is a scalar, order 1 a vector, order 2 a matrix, and order ≥ 3 a higher tensor. Its **shape** lists the size along each axis; e.g. a batch of RGB images is shape (N, C, H, W).

Intuition

Tensors are bookkeeping for "many matrices at once." Deep learning almost never works with a single vector; it works with batches, channels, time steps, and attention heads stacked along extra axes. The operations are still fundamentally linear-algebraic — they are matrix multiplies applied along chosen axes — but the index bookkeeping needs a richer notation.

12.1.1 Broadcasting and einsum

Broadcasting stretches a smaller array across a larger one without copying (adding a bias vector to every row of a matrix). **Einstein summation** (`einsum`) names each axis and sums over repeated indices, expressing any contraction unambiguously:

$$C_{ij} = \sum_k A_{ik} B_{kj} \iff \texttt{einsum("ik,kj->ij", A, B)}.$$

Batched matrix multiply, attention scores, and convolutions are all one `einsum` away — the notation is the practical lingua franca of tensor code.

Common pitfall

Tensor reshaping is where bugs hide. `reshape` reinterprets the same data in row-major order; `transpose` / `permute` reorders axes and changes the memory layout. Confusing them silently scrambles your data. Always reason about *shapes* (and print them); a shape mismatch caught early saves hours, and many "model not learning" bugs are really an axis

that was summed or transposed wrongly.

12.2 Derivatives in many dimensions

Generalizing the scalar derivative, we organize partial derivatives into vectors and matrices.

Definition 12.2. For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the **gradient** is the vector of partial derivatives $\nabla f = \left[\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}\right]^\top$. For a vector function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the **Jacobian** is the $m \times n$ matrix

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}.$$

Intuition

The gradient points in the direction of steepest increase of a scalar function — it is what gradient descent steps against. The Jacobian is the *best linear approximation* of a vector function near a point: locally, $\mathbf{f}(\mathbf{x} + \Delta) \approx \mathbf{f}(\mathbf{x}) + \mathbf{J}\,\Delta$. For a linear map $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$, the Jacobian is simply $\mathbf{A}$ itself — "the derivative of a linear map is the map." Linear algebra and calculus meet exactly here.

12.3 Essential gradient identities

A few results cover most of machine learning (here $\mathbf{A}$ is constant, $\mathbf{x}$ the variable):

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \qquad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}, \qquad \nabla_{\mathbf{x}} \|\mathbf{x}\|_2^2 = 2\mathbf{x}.$$

Applied to the least-squares loss $L(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$:

$$\nabla_{\mathbf{x}} L = 2\mathbf{A}^\top(\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{0} \implies \mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b},$$

recovering the normal equations of Chapter 7 by pure calculus — the same answer the geometry gave.

12.4 The chain rule and backpropagation

A neural network is a composition $\mathbf{f} = \mathbf{f}_L \circ \cdots \circ \mathbf{f}_1$. The multivariable chain rule says the Jacobian of a composition is the *product* of the Jacobians:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{f}_L} \frac{\partial \mathbf{f}_L}{\partial \mathbf{f}_{L-1}} \cdots \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}}.$$

Key idea

Backpropagation is the chain rule evaluated right-to-left. Because the loss is scalar, the leftmost factor is a row vector, and multiplying by Jacobians from the left keeps a vector (never a giant matrix) flowing backward. Each step is a **vector–Jacobian product** (VJP): we never build the full Jacobian — for a layer with millions of inputs and outputs that matrix would be astronomically large. Instead we use a closed-form rule for "upstream gradient $\mapsto$ downstream gradient."

Example 12.3 (Backprop through a linear layer)**.** For $\mathbf{Y} = \mathbf{X}\mathbf{W}$ with scalar loss L and upstream gradient $\mathbf{G} = \partial L/\partial \mathbf{Y}$, the downstream gradients are obtained *without* forming any Jacobian:

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^\top \mathbf{G}, \qquad \frac{\partial L}{\partial \mathbf{X}} = \mathbf{G}\,\mathbf{W}^\top.$$

Note the transposes and the order: the backward pass of a matrix multiply is two more matrix multiplies. This is the single most-executed gradient computation in deep learning.

Forward vs. reverse mode

Automatic differentiation can multiply the chain of Jacobians in two orders. **Forward mode** goes input-to-output and is efficient when there are few inputs. **Reverse mode** (backprop) goes output-to-input and computes the gradient with respect to *all* parameters in roughly one backward pass — ideal when there is one scalar loss and millions of parameters, exactly the deep-learning regime. That asymmetry is why every framework defaults to reverse-mode `.backward()`.

How this powers ML / AI

This chapter *is* the mechanism of training. Frameworks (PyTorch, JAX, TensorFlow) build a computation graph of tensor operations, attach a VJP rule to each, and call them in reverse to get $\nabla_{\boldsymbol{\theta}} L$ for every parameter; an optimizer then steps $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L$. Gradients are matrix products; activations are tensors; the whole forward and backward pass is linear algebra interleaved with cheap nonlinearities. Understanding Jacobians and VJPs lets you derive custom layers, debug exploding/vanishing gradients, and read the math of modern papers.

Try it in NumPy

```python
import torch
X = torch.randn(4, 3, requires_grad=False)
W = torch.randn(3, 2, requires_grad=True)
Y = X @ W
L = (Y**2).sum()
L.backward()                          # reverse-mode autodiff
print(torch.allclose(W.grad, X.T @ (2*Y)))   # matches X^T G, G = dL/dY
# einsum: batched matmul over a batch axis b
A = torch.randn(8, 4, 3); B = torch.randn(8, 3, 5)
C = torch.einsum("bij,bjk->bik", A, B)        # shape (8,4,5)
```

Chapter summary

- **Tensors** are n-D arrays; broadcasting and `einsum` express batched linear-algebra concisely; reason in shapes.
- The **gradient** (steepest ascent) and **Jacobian** (best local linear map) organize derivatives; for $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x}$, $\mathbf{J} = \mathbf{A}$.
- Key identities give the least-squares gradient and re-derive the normal equations.
- **Backprop** = chain rule right-to-left, a sequence of **vector–Jacobian products**; the Jacobian is never explicitly formed.

- Reverse-mode autodiff yields all parameter gradients in one backward pass — the engine of deep learning.

PART IV

Applications

13

Linear Algebra in Machine Learning, Deep Learning, and AI

Level: Capstone synthesis

This final chapter is the payoff. Every concept in the book reappears here as the working machinery of modern AI. The thesis is simple and, by now, earned: *linear algebra is the language in which machine learning is written.* Data are tensors; models are matrices; learning is optimization over matrix-valued functions; and the deepest tools — the SVD, the spectral theorem, the pseudoinverse — are exactly the ones that make AI tractable. We tour the landscape application by application, pointing back to the chapter where each idea was built.

13.1 Data is linear algebra

Before any algorithm runs, the data is already a linear-algebra object.

- A **tabular dataset** is a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$: n samples (rows) $\times$ d features (columns).
- An **image** is a matrix (grayscale) or order-3 tensor (color); a batch is order-4.
- **Text** becomes a matrix of token embeddings; a **corpus** a term–document matrix.
- A **graph** is its adjacency or Laplacian matrix; a **user–item** ratings table is a (sparse) matrix.

Choosing a representation *is* choosing a matrix, and the geometry of that matrix — its rank, its singular values, its null space — already constrains what any model can learn from it (Chapters 5, 10).

13.2 Linear and ridge regression: projection and the pseudoinverse

The foundational supervised model is linear regression, and we have already solved it three ways. Geometrically it is the projection of the target $\mathbf{y}$ onto the column space of the design matrix (Chapter 7); algebraically it is the normal equations $\mathbf{X}^\top \mathbf{X}\hat{\mathbf{w}} = \mathbf{X}^\top \mathbf{y}$; computationally it is the pseudoinverse $\hat{\mathbf{w}} = \mathbf{X}^+ \mathbf{y}$ via SVD (Chapter 10). **Ridge regression** adds $\lambda \|\mathbf{w}\|_2^2$, giving

$$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}.$$

The $\lambda \mathbf{I}$ term lifts every eigenvalue of $\mathbf{X}^\top \mathbf{X}$ by λ, raising $\sigma_{\min}$, lowering the condition number (Chapter 11), and selecting the minimum-norm solution when the null space is nontrivial (Chapter 5). Regularization is linear algebra repairing ill-posedness.

13.3 Principal Component Analysis: the SVD of data

PCA is the most-used unsupervised method, and it is nothing but the SVD of the centered data matrix (Chapters 9, 10).

Key idea

Center $\mathbf{X}$, take its SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$. The right singular vectors (columns of $\mathbf{V}$) are the **principal directions**; the singular values give the **variance** captured (σ_i^2/n); projecting onto the top k directions, $\mathbf{Z} = \mathbf{X}\mathbf{V}_k$, is the optimal k-dimensional linear summary of the data by Eckart–Young. PCA = change of basis to the variance-maximizing axes + truncation.

PCA powers visualization, denoising, decorrelation/whitening, and compression. The reason production code computes PCA from the SVD of $\mathbf{X}$ rather than the eigendecomposition of the covariance $\mathbf{X}^\top\mathbf{X}$ is numerical: forming $\mathbf{X}^\top\mathbf{X}$ squares the condition number (Chapter 11).

13.4 Neural networks: stacks of affine maps

A feedforward network alternates linear algebra with cheap nonlinearities:

$$\mathbf{h}^{(\ell)} = \phi\big(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)}\big).$$

Each layer is an affine transformation (matrix multiply + bias, Chapters 3, 6) followed by an activation ϕ. The nonlinearity is essential: without it, the composition $\mathbf{W}^{(L)}\cdots\mathbf{W}^{(1)}$ collapses to a single matrix and the network could only fit lines. Training computes $\nabla_{\boldsymbol{\theta}} L$ by backpropagation — the chain rule as a sequence of vector–Jacobian products (Chapter 12) — and every forward/backward pass is dominated by matrix multiplication, which is why GPUs (matrix-multiply engines) define the field's hardware.

Intuition

Read a deep network as a learned sequence of *changes of representation*: each layer re-expresses the input in new coordinates where the next sub-task is easier, until the final space makes the classes linearly separable. "Representation learning" is, very literally, learning a useful basis (Chapter 6).

13.5 Convolution and weight sharing

A convolutional layer is still a linear map — a highly structured matrix with shared, sparse weights (a Toeplitz/circulant structure). Expressing it as such explains its efficiency and its equivariance to translation, and reveals that the backward pass is again a (transposed) convolution, i.e. another structured matrix multiply (Chapter 12).

13.6 Embeddings and similarity

Words, items, users, and images are mapped to vectors in a learned space where *geometry encodes meaning*. Similarity is the (normalized) dot product — cosine similarity (Chapter 1):

$$\cos\theta = \frac{\mathbf{u}^\top\mathbf{v}}{\|\mathbf{u}\|\,\|\mathbf{v}\|}.$$

Nearest-neighbor retrieval in a **vector database** ranks by this quantity; analogies ("king − man + woman ≈ queen") are vector arithmetic. Embedding tables are matrices, and compressing them with truncated SVD/PCA (Chapter 10) shrinks storage 8–32× while often improving recall by discarding noisy directions.

13.7 Attention and Transformers

The Transformer — the architecture behind modern large language models — is built almost entirely from matrix multiplications. Packing a sequence of token vectors into rows of $\mathbf{X}$, three learned projections produce queries, keys, and values, and attention is

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}.$$

Key idea

Unpack the linear algebra. $\mathbf{Q}\mathbf{K}^\top$ is a matrix of *all pairwise dot products* between query and key vectors — token-to-token similarity (Chapter 1). Scaling by $1/\sqrt{d_k}$ controls the variance so the softmax stays well-conditioned. The softmax turns each row into attention weights, and multiplying by $\mathbf{V}$ forms a *weighted combination of value vectors* — a linear combination (Chapter 1) whose weights are data-dependent. Multi-head attention runs several such projections in parallel (block/tensor structure, Chapter 12). Attention is dot products, softmax, and matrix products — linear algebra end to end.

13.8 Low-rank structure: LoRA and model compression

Eckart–Young (Chapter 10) is now a daily engineering tool. **LoRA** fine-tunes a giant pretrained model by freezing its weight matrix $\mathbf{W}$ and learning a *low-rank* update $\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$ with $\mathbf{A} \in \mathbb{R}^{r\times d}$, $\mathbf{B} \in \mathbb{R}^{d\times r}$, $r \ll d$. This trains a tiny fraction of the parameters on the bet — empirically sound — that the needed adaptation is low-rank. The same low-rank idea compresses layers, accelerates inference, and underlies recommender-system matrix factorization (the Netflix Prize: complete a sparse ratings matrix with a low-rank model).

13.9 Graphs, spectra, and PageRank

Graph data lives in the adjacency matrix and the **graph Laplacian** $\mathbf{L} = \mathbf{D} - \mathbf{A}$ (symmetric PSD, Chapter 9). Its smallest eigenvectors give **spectral clustering** and the low-frequency features used by graph neural networks; **PageRank** is the dominant eigenvector of a Markov matrix (Chapter 8), historically found by the power method (Chapter 11). Spectral graph theory is eigenvalues doing applied work.

13.10 Probabilistic models and Gaussian processes

The multivariate Gaussian is pure linear algebra: a quadratic form $(\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})$ in the exponent (Chapter 9) and $\det(\boldsymbol{\Sigma})$ in the normalizer (Chapter 4). **Gaussian processes** hinge on Cholesky factorization of the (symmetric PD) kernel matrix for both the predictive mean and a stable log-determinant in the marginal likelihood (Chapter 11). Sampling $\mathbf{x} = \boldsymbol{\mu} + \mathbf{L}\mathbf{z}$ uses the Cholesky factor $\mathbf{L}$. **Normalizing flows** track the Jacobian determinant of an invertible neural map to transform densities exactly (Chapter 4).

13.11 Optimization landscapes: curvature is eigenvalues

Training is optimization, and its difficulty is governed by the spectrum of the loss Hessian (Chapters 8, 9). The eigenvalues are the curvatures along the principal directions; their ratio is the condition number, which sets how badly gradient descent zig-zags and bounds the stable

learning rate. Indefinite Hessians mean **saddle points**, which dominate high-dimensional deep-learning loss surfaces. Normalization, good initialization (often *orthogonal*, Chapter 7), and second-order/preconditioned methods are all attempts to improve conditioning.

13.12 The grand summary table

Linear-algebra concept	Ch.	Where it powers AI
Dot product / cosine	1	neurons, similarity, attention scores, retrieval
Norms (L1/L2/nuclear)	1,11	Lasso/Ridge, weight decay, low-rank penalties
Matrix multiply	3	every layer; batched on GPUs
Linear systems / null space	2,5	regression solvability; why regularization is needed
Rank & four subspaces	5	feature redundancy, identifiability
Linear maps / change of basis	6	layers as representation learning; PCA, whitening
Projection / least squares / QR	7	linear regression, orthogonal init, decorrelation
Eigenvalues / diagonalization	8	PageRank, spectral clustering, dynamics, curvature
Spectral theorem / PSD / quadratic forms	9	PCA, kernels, Gaussians, Hessian/saddle analysis
SVD / Eckart–Young / pseudoinverse	10	PCA, LSA, recommenders, LoRA, compression
Condition number / decompositions	11	training stability, GP Cholesky, randomized SVD
Tensors / Jacobians / VJP	12	autodiff, backpropagation, custom layers
Determinant / Jacobian det	4	Gaussian densities, normalizing flows

The one-paragraph thesis

Strip away the branding and modern AI is linear algebra running at scale. Data are tensors; a model is a pipeline of matrix multiplies separated by nonlinearities; training projects, factors, and differentiates those matrices; and the field's hardest problems — compression, stability, generalization, efficient adaptation — are solved with the SVD, the spectral theorem, conditioning, and low-rank structure. Master the twelve concepts in the table and the equations in any deep-learning paper stop being intimidating: you will recognize them as old friends from this book.

Where to go next

With this foundation, the natural next steps are multivariable calculus and optimization (gradient methods, convexity), probability and statistics (estimation, the Gaussian, Bayes), and then the deep-learning curriculum proper — where you will see these matrices come alive in convolutional networks, Transformers, and generative models. You now read the language; the rest is vocabulary.

Chapter summary

- Data (tables, images, text, graphs) are matrices and tensors; their geometry bounds what models can learn.
- Regression is projection / pseudoinverse; ridge is conditioning repair; PCA is the SVD of data.
- Neural nets are stacked affine maps; attention is dot products + softmax + matrix products; LoRA is low-rank Eckart–Young.
- Eigenvalues run PageRank, spectral clustering, and curvature/optimization analysis; Cholesky and determinants run probabilistic models.
- The whole of modern AI is linear algebra executed at scale — which is exactly why this subject was worth mastering.

A

Notation and Formula Reference

A compact reference for the symbols and identities used throughout the book.

A.1 Notation

Symbol	Meaning
$\mathbb{R}$, $\mathbb{R}^n$, $\mathbb{R}^{m\times n}$	reals; n-vectors; $m \times n$ matrices
$\mathbf{x}, \mathbf{v}, \mathbf{w}$ (bold lower)	column vectors
$\mathbf{A}, \mathbf{B}, \mathbf{Q}$ (bold upper)	matrices
a_{ij}	entry in row i, column j of $\mathbf{A}$
$\mathbf{A}^\top$, $\mathbf{A}^{-1}$, $\mathbf{A}^+$	transpose, inverse, pseudoinverse
$\mathbf{u}\cdot\mathbf{v} = \mathbf{u}^\top\mathbf{v}$	dot (inner) product
$\|\mathbf{x}\|_p$	p-norm ($p = 1, 2, \infty$)
span, rank, null	span, rank, null space
$\mathrm{Col}(\mathbf{A})$, $\mathrm{Row}(\mathbf{A})$	column space, row space
$\det(\mathbf{A})$, $\mathrm{tr}(\mathbf{A})$	determinant, trace
λ, $\mathbf{v}$	eigenvalue, eigenvector
σ_i	i-th singular value
$\mathbf{\Sigma}$, $\mathbf{\Lambda}$	diagonal of singular values; of eigenvalues
∇f, $\mathbf{J}$	gradient, Jacobian
$\mathrm{cond}(\mathbf{A}) = \sigma_{\max}/\sigma_{\min}$	condition number

A.2 Core identities

Products and transposes.

$$(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top, \quad (\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}, \quad (\mathbf{A}^\top)^{-1} = (\mathbf{A}^{-1})^\top.$$

Determinant and trace.

$$\det(\mathbf{AB}) = \det\mathbf{A}\det\mathbf{B}, \quad \det(\mathbf{A}^\top) = \det\mathbf{A}, \quad \mathrm{tr}(\mathbf{AB}) = \mathrm{tr}(\mathbf{BA}), \quad \det\mathbf{A} = \prod_i \lambda_i, \quad \mathrm{tr}\,\mathbf{A} = \sum_i \lambda_i.$$

Rank–nullity. $\mathrm{rank}(\mathbf{A}) + \dim\mathrm{null}(\mathbf{A}) = n$.

Factorizations.

$$\mathbf{A} = \mathbf{LU}, \quad \mathbf{A} = \mathbf{QR}, \quad \mathbf{A} = \mathbf{LL}^\top \text{ (SPD)}, \quad \mathbf{A} = \mathbf{PDP}^{-1}, \quad \mathbf{A} = \mathbf{Q\Lambda Q}^\top \text{ (sym.)}, \quad \mathbf{A} = \mathbf{U\Sigma V}^\top.$$

Least squares & projection.

$$\mathbf{A}^\top \mathbf{A}\,\hat{\mathbf{x}} = \mathbf{A}^\top \mathbf{b}, \qquad \mathbf{P} = \mathbf{A}(\mathbf{A}^\top \mathbf{A})^{-1}\mathbf{A}^\top, \qquad \hat{\mathbf{x}} = \mathbf{A}^{+}\mathbf{b}.$$

Low-rank (Eckart–Young).

$$\mathbf{A}_k = \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^\top, \qquad \|\mathbf{A} - \mathbf{A}_k\|_2 = \sigma_{k+1}.$$

Matrix calculus.

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \quad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}, \quad \nabla_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 = 2\mathbf{A}^\top(\mathbf{A}\mathbf{x} - \mathbf{b}).$$

A.3 Equivalent conditions for an invertible $n \times n$ matrix

The following are all equivalent (the "invertible matrix theorem"): $\mathbf{A}$ is invertible; $\det \mathbf{A} \neq 0$; $\operatorname{rank} \mathbf{A} = n$; columns (and rows) are linearly independent; $\operatorname{null}(\mathbf{A}) = \{\mathbf{0}\}$; $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a unique solution for every $\mathbf{b}$; 0 is not an eigenvalue; all singular values are positive; the columns are a basis of $\mathbb{R}^n$.

A.4 Greek letters used

α, β, γ (scalars/coefficients), θ (angle, parameters $\boldsymbol{\theta}$), λ (eigenvalue, regularization), σ (singular value, std. dev.), μ (mean), Σ (covariance, sum), Λ (eigenvalue matrix), ϕ (activation), η (learning rate).

A.5 Minimal NumPy / PyTorch reference

Task	Call
solve $\mathbf{A}\mathbf{x} = \mathbf{b}$	`np.linalg.solve(A, b)`
least squares	`np.linalg.lstsq(A, b, rcond=None)`
inverse / pseudoinverse	`np.linalg.inv(A)` / `np.linalg.pinv(A)`
determinant / log-det	`np.linalg.det(A)` / `np.linalg.slogdet(A)`
rank / condition number	`np.linalg.matrix_rank(A)` / `np.linalg.cond(A)`
eigen (general / symmetric)	`np.linalg.eig(A)` / `np.linalg.eigh(A)`
SVD	`np.linalg.svd(A, full_matrices=False)`
QR / Cholesky	`np.linalg.qr(A)` / `np.linalg.cholesky(A)`
autodiff gradient	`loss.backward()` (PyTorch)

End of book. You now read the language of machine learning.