Linear Algebra Fundamentals

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Vectors, the dot & cross products · matrices & transformations

Systems of linear equations & Gaussian elimination · the determinant & inverse

Span, basis & dimension · eigenvalues & eigenvectors

Capstone: a 2D sprite transform pipeline feeding into a PCA data-analysis example

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Linear Algebra Matters for Programmers

Linear Algebra Fundamentals

Chapter 1 · Why Linear Algebra Matters for Programmers

Linear algebra is the math of collections of numbers that move and combine together — a vector, a matrix, a transformation. It sounds abstract until you notice how much of it is already sitting quietly inside code you've probably written: an RGB colour is a 3-vector, a CSS `transform` is a matrix, a "feature vector" fed into a machine learning model is, literally, a vector. This course is about making that quiet, already-present math explicit and usable on purpose, rather than leaving it as something that happens to work.

What Linear Algebra Actually Studies

Where Discrete Mathematics Fundamentals is about counting, logic, and discrete structure, linear algebra is about a different question entirely: given a collection of numbers arranged as a vector or a matrix, what operations can be done to it, and what do those operations mean geometrically? A vector can represent a position, a direction, a colour, or a list of features — the same handful of operations (add, scale, multiply by a matrix) apply regardless of what the numbers represent.

	A VECTOR	A MATRIX
What it is	An ordered list of numbers — `[3, 4]`, `[255, 0, 128]`	A rectangular grid of numbers, arranged in rows and columns
One way to think about it	A point, or a direction and distance, in space	A rule for transforming vectors — rotate, scale, skew
Code equivalent	A Python list, a NumPy 1-D array, a tuple of coordinates	A 2-D array, a NumPy matrix, a CSS `matrix()` transform
Example use	A player's (x, y) position; an RGB colour; a document's feature vector	Rotating a sprite; converting one colour space to another; a neural network layer's weights

Five Concrete Connections to Code Already On This Site

Every topic in this course maps onto something already covered elsewhere on this site, usually without the underlying linear algebra ever being named:

LINEAR ALGEBRA TOPIC	WHERE IT ACTUALLY SHOWS UP
Vectors & the dot product (Ch.2)	Positions and directions in any graphics/game code; the dot product is exactly how "how similar are these two things" gets measured in Machine Learning Fundamentals and Neural Networks & Deep Learning
Matrices as transformations (Ch.4–5)	Blender Fundamentals' object transforms, Vector Graphics and Figma's own scale/rotate/skew tools, CSS's own `transform: matrix(...)`
Systems of linear equations (Ch.6)	Fitting a straight line through a set of data points (linear regression) is solving a system of linear equations for the best-fit slope and intercept
Determinant & inverse (Ch.7)	Checking whether a graphics transform can be "undone" at all — a zero determinant means information was irreversibly flattened away
Eigenvalues & eigenvectors (Ch.9)	Principal Component Analysis (dimensionality reduction, touched on in Data Science Fundamentals and Machine Learning Fundamentals) and the mathematics underneath Google's original PageRank algorithm

What This Course Won't Cover

A few genuinely related topics are deliberately left for their own future courses under this same Maths for Programmers subject, rather than folded in here as extra chapters:

- **Calculus-based optimization** — how gradient descent actually derives its update rule, or how backpropagation's chain rule works, get their own future **Calculus & Optimization** course, even though eigenvalues (Chapter 9) sit right next door to that territory
- **Formal vector space theory** — axiomatic proofs about abstract vector spaces over arbitrary fields stay out of scope; Chapter 8 covers span, basis, and dimension, but kept concrete and code-grounded rather than proof-heavy
- **Numerical stability at scale** — why a huge, ill-conditioned matrix can quietly produce garbage results in floating-point arithmetic is real and important, but belongs to this subject's own future **Numerical Methods & Floating-Point Computation** course

> **WHY DRAW THE LINE HERE INSTEAD OF COVERING EVERYTHING AT ONCE**
>
> Each of those three topics is substantial enough to deserve its own real depth rather than a rushed chapter bolted onto this course. This course stays tightly scoped to vectors, matrices, transformations, and the handful of ideas (systems of equations, determinants, eigenvalues) that sit directly on top of them — the concrete foundation the other courses will each build on, once their own turn comes.

Where This Course Is Headed

CHAPTER	TOPIC
2	Vectors — Operations & Geometric Intuition
3	The Cross Product & Working in 3D
4	Matrices — Representation & Basic Operations
5	Matrices as Transformations
6	Systems of Linear Equations & Gaussian Elimination
7	The Determinant & Matrix Inverse
8	Vector Spaces, Span, Basis & Dimension
9	Eigenvalues & Eigenvectors
10	Capstone — Linear Algebra in Practice

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: how do you represent and manipulate multi-dimensional data with a small, consistent set of operations? Once vectors and matrices are second nature, tools that look completely unrelated on the surface — a game engine's transform stack, a machine learning model's weight matrix, a graphic design app's rotate handle — turn out to be the exact same handful of operations wearing different names.

Hands-On Exercises

EXERCISE 1

For each of the following, say whether it's more naturally represented as a vector or a matrix, and briefly justify each answer: (a) an RGB colour, (b) a rotation that can be applied to any point in a 2D scene, (c) a single data point with five numeric features fed into a machine learning model, (d) a full black-and-white image where each pixel is a brightness value.

View solution

EXERCISE 2

A colleague claims "linear algebra is only really relevant if you're doing machine learning." Using this chapter's own five connections, explain at least two places linear algebra shows up in code that has nothing to do with ML.

View solution

EXERCISE 3

For each of the following real tools/code artifacts, name which linear algebra topic from this chapter's own table it most directly maps to, and explain the connection in one or two sentences: (a) a CSS `transform: matrix(a, b, c, d, e, f)` rule, (b) fitting a straight trend line through a scatter plot of data points, (c) reducing a dataset with 50 columns down to the 2 or 3 "directions" that capture most of its variation.

View solution

CHAPTER 1 QUICK REFERENCE

- **Vector** = an ordered list of numbers, often a point or direction; **matrix** = a grid of numbers, often a rule for transforming vectors
- Five direct connections: vectors/dot product → positions & similarity, matrices → transforms, linear systems → curve fitting, determinant/inverse → reversibility, eigenvalues → dimensionality reduction & PageRank
- Deliberately out of scope here: Calculus & Optimization, formal vector space proofs, and Numerical Methods & Floating-Point Computation each get their own future course
- This course stays concrete and code-grounded — vectors, matrices, transformations, and the ideas built directly on top of them
- **Next chapter:** Vectors — operations and the geometric intuition behind them

CHAPTER 2 OF 10

Vectors: Operations & Geometric Intuition

Linear Algebra Fundamentals

Chapter 2 · Vectors: Operations & Geometric Intuition

Chapter 1 introduced a vector as "an ordered list of numbers, often a point or a direction." This chapter makes that concrete: the handful of operations vectors support, what each one actually does to the arrow it represents, and — the operation that does the most real work in practice — the dot product, which turns out to be the mathematical basis for "how similar are these two things."

Vector Addition — Tip to Tail

Adding two vectors is done component by component: $a + b = [a_1+b_1,\ a_2+b_2]$. Geometrically, this is the "tip to tail" picture — place the second vector's tail at the first vector's tip, and the sum is the arrow from the very start to the very end.

```
a = [3, 4]
b = [1, 2]

# component-wise addition
a_plus_b = [a[i] + b[i] for i in range(len(a))]
print(a_plus_b)  # [4, 6]
```

A worked example used throughout this chapter: **a = [3, 4]** and **b = [1, 2]**. So **a + b = [4, 6]** — verified above.

Scalar Multiplication — Stretching, Shrinking & Reversing

Multiplying a vector by a plain number (a **scalar**) scales every component by that number: $k \cdot a = [k \cdot a_1,\ k \cdot a_2]$. Geometrically, a scalar greater than 1 stretches the arrow, a scalar between 0 and 1 shrinks it, and a **negative** scalar reverses its direction entirely while still scaling its length.

SCALAR	2 · A (A = [3, 4])	GEOMETRIC EFFECT
k = 2	[6, 8]	Same direction, twice as long
k = 0.5	[1.5, 2]	Same direction, half as long
k = -1	[-3, -4]	Exactly opposite direction, same length

Magnitude — How Long Is the Arrow

The **magnitude** (or "norm," written `|a|`) of a vector is its length, computed with the Pythagorean theorem generalized to as many dimensions as the vector has:

```
# |a| = sqrt(a1^2 + a2^2 + ... + an^2)
import math

def magnitude(v):
    return math.sqrt(sum(x**2 for x in v))

print(magnitude([3, 4]))  # 5.0
```

For **a = [3, 4]**: `|a| = √(3² + 4²) = √(9 + 16) = √25 = 5` — the classic 3-4-5 right triangle, just relabelled as a vector.

Normalization — Keeping Only the Direction

Dividing a vector by its own magnitude produces a **unit vector** — a vector of length exactly 1 that points in the same direction as the original, with the "how far" information stripped away and only "which way" left behind. This matters constantly in graphics and game code, where a direction is often needed independent of any particular distance.

Normalizing **a = [3, 4]** (magnitude 5): `a / |a| = [3/5, 4/5] = [0.6, 0.8]`. Checking the result is genuinely a unit vector: `√(0.6² + 0.8²) = √(0.36 + 0.64) = √1 = 1` ✓.

> **A REAL BUG: NORMALIZING THE ZERO VECTOR**
>
> The zero vector `[0, 0]` has magnitude 0, and dividing by 0 either crashes or produces `NaN` depending on the language. Any code that normalizes a vector coming from user input or a computed difference (e.g. "direction from A to B" when A and B happen to be the same point) needs to guard against this case explicitly.

The Dot Product — Measuring "How Aligned"

The **dot product** of two vectors is computed algebraically by multiplying corresponding components and summing the results: $a \cdot b = a_1b_1 + a_2b_2 + \ldots + a_nb_n$. It produces a single number, not a vector — and that number turns out to have a precise geometric meaning.

```
def dot(a, b):
    return sum(a[i] * b[i] for i in range(len(a)))

a = [3, 4]
b = [1, 2]
print(dot(a, b))  # 3*1 + 4*2 = 3 + 8 = 11
```

The geometric meaning connects the dot product directly to the angle θ between the two vectors:

THE DOT PRODUCT / ANGLE FORMULA

`a · b = |a| |b| cos(θ)` — rearranged, `cos(θ) = (a · b) / (|a| |b|)`. The dot product is, in effect, "how much of b points in the same direction as a," scaled by both lengths.

Using **a = [3, 4]** and **b = [1, 2]**: `a · b = 11`, `|a| = 5`, `|b| = √5 ≈ 2.236`, so `cos(θ) = 11 / (5 × 2.236) ≈ 0.984`, giving `θ ≈ 10.3°` — a and b point in nearly, but not exactly, the same direction.

SIGN OF A · B	WHAT IT MEANS ABOUT THE ANGLE
Positive	Angle is less than 90° — vectors point in roughly the same general direction
Zero	Angle is exactly 90° — the vectors are **orthogonal** (perpendicular)
Negative	Angle is more than 90° — vectors point in roughly opposite directions

WHY THIS IS "SIMILARITY" IN MACHINE LEARNING

A feature vector is just a vector with more than two or three components. The dot product (usually normalized into what's called cosine similarity) is exactly how tools compare two feature vectors — two documents, two user preference profiles, two embeddings — for "how alike are these," with no geometry required in the visual sense at all. The same formula, unchanged.

Vectors in Code — By Hand vs. NumPy

Every operation above was written from scratch using plain Python lists, which is worth doing once to see exactly what's happening. In real code, especially anything performance-sensitive or higher-dimensional, the NumPy library provides all of these operations directly and considerably faster:

```
import numpy as np

a = np.array([3, 4])
b = np.array([1, 2])

print(a + b)                    # [4 6]
print(2 * a)                      # [6 8]
print(np.linalg.norm(a))         # 5.0  (magnitude)
print(a / np.linalg.norm(a))     # [0.6 0.8]  (normalized)
print(np.dot(a, b))              # 11  (dot product)
```

Hands-On Exercises

EXERCISE 1

Given `c = [6, 8]` and `d = [-2, 1]`, compute by hand: (a) `c + d`, (b) `3 · d`, (c) the magnitude of `c`, (d) the normalized (unit) vector for `c`. Show your working for each.

View solution

EXERCISE 2

Compute the dot product of `c = [6, 8]` and `d = [-2, 1]`, then use it to find the angle between them in degrees. Based only on the sign of the dot product (before finishing the full angle calculation), predict whether the angle should be less than, equal to, or greater than 90° — then confirm your prediction with the final answer.

View solution

EXERCISE 3

A recommendation system represents two users as feature vectors: `user_x = [5, 1, 0]` and `user_y = [4, 2, 1]` (three genres, rated 0–5). Compute the dot product of the two vectors, and explain in plain terms what a high dot product suggests about these two users' tastes — and why a raw dot product alone can be misleading if one user rates everything much higher than the other (hint: think about what normalizing each vector first would fix).

View solution

CHAPTER 2 QUICK REFERENCE

- **Addition:** component-wise, geometrically "tip to tail" — $a + b = [a_1+b_1, a_2+b_2, ...]$
- **Scalar multiplication:** scales every component; a negative scalar reverses direction — $k \cdot a = [k \cdot a_1, k \cdot a_2, ...]$
- **Magnitude:** $|a| = \sqrt{(a_1^2 + a_2^2 + ... + a_n^2)}$ — the vector's length
- **Normalization:** $a / |a|$ gives a unit vector (length 1) pointing in the same direction — guard against dividing by a zero-length vector
- **Dot product:** $a \cdot b = a_1b_1 + a_2b_2 + ...$; geometrically $a \cdot b = |a||b|\cos(\theta)$
- Sign of the dot product tells you the angle category: positive → <90°, zero → exactly 90° (orthogonal), negative → >90°
- The dot product is the mathematical basis of "similarity" between feature vectors in machine learning
- **Next chapter:** The cross product and working specifically in 3D

CHAPTER 3 OF 10

The Cross Product & Working in 3D

Linear Algebra Fundamentals

Chapter 3 · The Cross Product & Working in 3D

Chapter 2's dot product takes two vectors and returns a single number. The **cross product** is a different kind of operation entirely: it takes two 3D vectors and returns a third *vector* — one that's perpendicular to both of the originals. It's specifically a 3D operation, which is itself worth understanding, not just the formula.

The Formula

For $a = [a_1, a_2, a_3]$ and $b = [b_1, b_2, b_3]$, the cross product is:

CROSS PRODUCT FORMULA

$$a \times b = [a_2b_3 - a_3b_2,\ a_3b_1 - a_1b_3,\ a_1b_2 - a_2b_1]$$

It looks arbitrary at first, but there's a pattern: each output component skips the matching input index (the first output component has no a_1/b_1 in it, and so on), and the two products in each pair are subtracted in a "cross" pattern. Chapter 7's own determinant will give this same formula a cleaner, more memorable form once determinants are available.

```
def cross(a, b):
    return [
        a[1]*b[2] - a[2]*b[1],
        a[2]*b[0] - a[0]*b[2],
        a[0]*b[1] - a[1]*b[0],
    ]

i = [1, 0, 0]
j = [0, 1, 0]
print(cross(i, j))  # [0, 0, 1] — this is exactly the identity i x j = k
```

The Right-Hand Rule

The direction of $a \times b$ is found with the **right-hand rule**: point your right hand's fingers along a, curl them toward b, and your thumb points in the direction of the result. This is also why order matters — the cross product is **anti-commutative**:

A × B = −(B × A)

Swapping the order flips the sign of every term in the formula, which reverses the resulting vector's direction while leaving its length unchanged. `i × j = [0, 0, 1]`, but `j × i = [0, 0, -1]` — same axis, opposite direction.

Geometric Meaning: Perpendicular, With a Meaningful Length

Two things are true about `a × b` at once, and both matter in practice:

1. **Direction:** it's perpendicular to *both* `a` and `b` — confirmed by the fact that `(a × b) · a = 0` and `(a × b) · b = 0` always hold, using Chapter 2's own dot product as the perpendicularity test.
2. **Magnitude:** `|a × b| = |a| |b| sin(θ)` — which is exactly the area of the parallelogram that `a` and `b` span.

Worked example: `a = [2, 1, 0]`, `b = [1, 0, 2]`.

QUANTITY	VALUE
a × b	[2, −4, −1]
(a × b) · a	0 ✓ perpendicular to a
(a × b) · b	0 ✓ perpendicular to b
|a × b|	$\sqrt{(2^2 + 4^2 + 1^2)} = \sqrt{21} \approx 4.583$
|a| |b| sin(θ)	$\sqrt{5} \times \sqrt{5} \times 0.9165 \approx 4.583$ — an exact match

Testing for Parallel Vectors

Since `sin(θ) = 0` exactly when `θ` is 0° or 180°, the cross product of two **parallel** (or anti-parallel) vectors is always the zero vector, regardless of their lengths. This gives a direct, practical test: if `a × b = [0, 0, 0]`, the two vectors point along the same line.

FLOATING-POINT REALITY CHECK

In real code, comparing a computed cross product to exactly `[0, 0, 0]` is fragile — rounding error means "should be zero" often comes out as something like `[1e-16, -3e-17, 0]` instead. The practical version of this test checks whether the magnitude of the cross product is *below a small tolerance*, not whether it's exactly zero.

2D vs. 3D: There's No True 2D Cross Product

The cross product as defined above genuinely requires three dimensions — "perpendicular to both a and b" only pins down a unique direction (up to sign) when there's a third axis to be perpendicular *into*. In 2D, a lot of code still uses a "cross product," but it's really a shortcut scalar, not a true cross product:

THE 2D PSEUDO-CROSS-PRODUCT

For 2D vectors $a = [a_1, a_2]$ and $b = [b_1, b_2]$, extending both into 3D with a zero third component ($[a_1, a_2, 0]$) and taking the real cross product leaves only a z-component: $a_1b_2 - a_2b_1$. That single number is what 2D code usually means by "the cross product" — it's really the z-component of the true 3D cross product, and its sign tells you whether b is a clockwise or counter-clockwise turn from a. Reusing Chapter 2's own worked vectors $a = [3, 4]$, $b = [1, 2]$: $a_1b_2 - a_2b_1 = 3\times2 - 4\times1 = 2$, which matches extending both to $[3, 4, 0]$ and $[1, 2, 0]$ and computing the real cross product — $[0, 0, 2]$.

Where This Shows Up: Surface Normals

In 3D graphics, a flat triangular face is usually stored as three corner points. To light that face correctly, the renderer needs its **normal** — a vector pointing straight out from the surface. That's computed by taking two of the triangle's edges as vectors and crossing them:

WHY THE ORDER OF THE TWO EDGE VECTORS MATTERS

Given edges $u = [2, 0, 0]$ and $v = [0, 3, 0]$, $u \times v = [0, 0, 6]$ — pointing out of the page. But $v \times u = [0, 0, -6]$ — pointing straight into it. This is exactly why 3D modelling tools like Blender care about a face's **winding order** (the order its corner points are listed in): it determines which way $u \times v$ points, which determines which side of the surface is treated as the "front" for lighting and backface culling.

Cross Products in Code — NumPy

```
import numpy as np

a = np.array([2, 1, 0])
b = np.array([1, 0, 2])

print(np.cross(a, b))             # [ 2 -4 -1]
print(np.linalg.norm(np.cross(a, b)))  # 4.58257569... (area of the parallelogram)
```

Hands-On Exercises

EXERCISE 1

Compute $e \times f$ by hand for $e = [1, -2, 3]$ and $f = [2, 0, -1]$, showing each component's calculation. Then verify your result is genuinely perpendicular to both e and f by computing the two dot products.

View solution

EXERCISE 2

Given `g = [2, 4, -2]` and `h = [-1, -2, 1]`, use the cross product to determine whether they're parallel. Show the calculation, and separately confirm your conclusion by checking whether one vector is a scalar multiple of the other.

View solution

EXERCISE 3

A triangular face has two edge vectors `u = [2, 0, 0]` and `v = [0, 3, 0]`. Compute the surface normal as `u × v`, then compute it the other order as `v × u`. Explain, in terms of the right-hand rule, why a 3D modelling tool listing this triangle's corner points in the opposite order would flip which side of the surface is treated as the front.

View solution

CHAPTER 3 QUICK REFERENCE

- **Cross product:** $a \times b = [a_2b_3 - a_3b_2,\ a_3b_1 - a_1b_3,\ a_1b_2 - a_2b_1]$ — a 3D-only operation returning a vector, not a scalar
- **Direction:** perpendicular to both inputs, given by the right-hand rule; **anti-commutative** — $a \times b = -(b \times a)$
- **Magnitude:** $|a \times b| = |a||b|\sin(\theta)$ = the area of the parallelogram spanned by a and b
- A zero cross product (within floating-point tolerance) means the two vectors are parallel or anti-parallel
- There's no true 2D cross product — the "2D cross product" ($a_1b_2 - a_2b_1$) is a scalar shortcut equal to the z-component of the 3D version
- Surface normals in graphics are computed by crossing two edge vectors of a face; the order of the edges (winding order) determines which way the normal points
- **Next chapter:** Matrices — representation and basic operations

CHAPTER 4 OF 10

Matrices: Representation & Basic Operations

Linear Algebra Fundamentals

Chapter 4 · Matrices: Representation & Basic Operations

Chapters 2 and 3 worked entirely with vectors — single ordered lists of numbers. This chapter introduces the other core object of linear algebra: the **matrix**, a rectangular grid of numbers arranged in rows and columns. A matrix can represent a table of data, an image's pixel grid, or — most importantly for this course — a *rule* for transforming vectors, which Chapter 5 covers in full.

What a Matrix Is

A matrix with `m` rows and `n` columns is called an "`m × n` matrix." Each individual number is an **entry**, identified by its row and column position.

```
# A 2x2 matrix, represented as a list of rows
A = [
    [1, 2],
    [3, 4],
]
# A[0] is the first row: [1, 2]
# A[1][0] is the entry in row 1, column 0: 3
```

CONCEPT	VECTOR (CH.2–3)	MATRIX (THIS CHAPTER)
Shape	A single row (or column) of numbers	A genuine 2D grid — rows *and* columns
Code equivalent	A flat Python list	A list of lists (or a 2D NumPy array)
One way to think about it	A point or direction	A rule for transforming vectors, or a table of data

Matrix Addition & Scalar Multiplication — The Easy Part

These work exactly like the vector versions from Chapter 2: entry by entry. Addition requires both matrices to have the *exact same shape*.

Worked example: `E = [[2, -1], [0, 4]]`, `F = [[1, 3], [-2, 5]]`.

OPERATION	RESULT
E + F	[[3, 2], [-2, 9]]
3 · E	[[6, -3], [0, 12]]

Matrix Multiplication — The Real Operation

This is the operation that actually does the interesting work, and it does **not** mean multiplying entry-by-entry the way addition does. To find the entry at row `i`, column `j` of the result, take the **dot product** (Chapter 2 again) of row `i` from the first matrix and column `j` from the second.

> **THE DIMENSION RULE**
>
> For `A × B` to be defined, the number of **columns** in `A` must equal the number of **rows** in `B`. If `A` is `m × n` and `B` is `n × p`, the result is `m × p` — it inherits the "outer" dimensions and the shared "inner" dimension disappears entirely.

```
def matmul(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])
    assert cols_A == rows_B, "inner dimensions must match"
    result = [[0]*cols_B for _ in range(rows_A)]
    for i in range(rows_A):
        for j in range(cols_B):
            result[i][j] = sum(A[i][k] * B[k][j] for k in range(cols_A))
    return result
```

Worked example: `A = [[1, 2], [3, 4]]`, `B = [[5, 6], [7, 8]]` — both 2×2, so the result is 2×2.

ENTRY	CALCULATION	RESULT
Row 0, Col 0	(1×5) + (2×7) = 5 + 14	19
Row 0, Col 1	(1×6) + (2×8) = 6 + 16	22
Row 1, Col 0	(3×5) + (4×7) = 15 + 28	43
Row 1, Col 1	(3×6) + (4×8) = 18 + 32	50

So `A × B = [[19, 22], [43, 50]]`.

Why Matrix Multiplication Isn't Commutative

Computing `B × A` instead — same two matrices, reversed order — gives `[[23, 34], [31, 46]]`, a completely different result from `A × B = [[19, 22], [43, 50]]`.

A × B ≠ B × A, IN GENERAL

Unlike ordinary number multiplication, order matters for matrices — even when both `A × B` and `B × A` happen to be defined, they're usually genuinely different matrices. This isn't a technicality: since a matrix will turn out to represent a transformation in Chapter 5 (a rotation, a scale, a reflection), `A × B` means "apply B, then apply A" — and applying a rotation then a scale is visibly not the same as scaling first, then rotating.

The dimension rule can make the non-commutativity even starker. With a non-square 2×3 matrix `C` and a 3×2 matrix `D`, `C × D` is a valid 2×2 matrix — but `D × C` is *also* valid, and produces a 3×3 matrix. Same two matrices, same multiplication rule, two entirely different-shaped results, purely from the order they're multiplied in.

The Identity Matrix

The **identity matrix** `I` is the matrix equivalent of the number 1: multiplying any matrix by it (in either order, where the shapes allow) leaves that matrix completely unchanged. It has 1s down the main diagonal and 0s everywhere else.

```
I = [
    [1, 0],
    [0, 1],
]
# A x I == A, always
```

Confirmed with the earlier example: `A × I = [[1, 2], [3, 4]]` — exactly `A`, unchanged.

FORWARD REFERENCE — WHERE THIS IS GOING

Chapter 5 uses exactly this machinery to represent transformations: a rotation matrix, a scale matrix, a reflection matrix — each one is just a specific grid of numbers that, when multiplied against a vector, produces the transformed version of that vector. The identity matrix is, unsurprisingly, "the transformation that does nothing."

Matrices in Code — NumPy

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(A + B)        # entry-wise addition
print(A @ B)        # real matrix multiplication — NOT A * B
print(np.eye(2))    # the 2x2 identity matrix
```

A REAL NUMPY GOTCHA

In NumPy, `A * B` does **entry-wise** multiplication (multiplying matching positions directly), not the row-by-column matrix multiplication this chapter defines. The `@` operator (or `np.matmul`) is what performs genuine matrix multiplication — mixing the two up is a common source of silently wrong results, since both produce a same-shaped matrix without an obvious error.

Hands-On Exercises

EXERCISE 1

Given `G = [[2, 0], [1, 3]]` and `H = [[4, 1], [2, 5]]`, compute `G × H` and `H × G` by hand, showing each entry's calculation. Confirm the two results are different.

View solution

EXERCISE 2

A 2×3 matrix `J = [[1, 0, 2], [3, 1, 1]]` and a 3×1 matrix (a column vector) `K = [[2], [1], [4]]` are given. Compute `J × K` by hand, stating the resulting shape. Then explain, using this chapter's own dimension rule, exactly why `K × J` is not defined.

View solution

EXERCISE 3

A teammate writes NumPy code using `A * B` where `A` and `B` are both 3×3 matrices, intending to perform real matrix multiplication, and is confused that the result "looks wrong" compared to doing the calculation by hand. Explain what's actually happening, what the code should say instead, and why the bug wasn't caught by an error or crash.

View solution

CHAPTER 4 QUICK REFERENCE

- A matrix is an `m × n` grid of numbers — `m` rows, `n` columns
- **Addition/scalar multiplication:** entry-by-entry, same as vectors; addition requires identical shapes
- **Matrix multiplication:** each output entry is the dot product of a row from the first matrix and a column from the second
- **Dimension rule:** `A (m×n) × B (n×p) → result (m×p)` — the inner dimensions must match and disappear from the result's shape
- **Not commutative:** `A × B ≠ B × A` in general — order matters, and can even change the result's shape entirely
- **Identity matrix:** 1s on the diagonal, 0s elsewhere — `A × I = A`, the "do nothing" transformation

- In NumPy, `@` is real matrix multiplication; `*` is entry-wise — mixing them up produces a same-shaped but silently wrong result
- **Next chapter:** Matrices as transformations — rotation, scaling, reflection, and composing them

CHAPTER 5 OF 10

Matrices as Transformations

Linear Algebra Fundamentals

Chapter 5 · Matrices as Transformations

Chapter 4 ended with a forward reference: a matrix is "a rule for transforming vectors." This chapter makes that literal. Multiplying a matrix by a vector — a special case of Chapter 4's own matrix multiplication, where the second matrix has just one column — produces a new vector, and a small, well-known family of matrices produce specific, useful transformations: scaling, rotating, reflecting, and shearing.

Matrix-Vector Multiplication — The Same Rule, One Column Wide

A vector `v = [3, 4]` can be written as a 2×1 matrix, a single column: `[[3], [4]]`. Multiplying a 2×2 matrix by it uses exactly Chapter 4's row-by-column rule — each output entry is the dot product of a matrix row and the vector's single column.

```
def transform(M, v):
    # M is a 2x2 matrix, v is a 2-element vector
    return [
        M[0][0]*v[0] + M[0][1]*v[1],
        M[1][0]*v[0] + M[1][1]*v[1],
    ]
```

Scaling

A scaling matrix has the scale factors on the diagonal, zeros elsewhere:

SCALING MATRIX

`S = [[sₓ, 0], [0, sᵧ]]` — stretches x by `sₓ`, y by `sᵧ`, independently.

With `S = [[2, 0], [0, 0.5]]` and `v = [3, 4]`: `S v = [2×3, 0.5×4] = [6, 2]` — twice as wide, half as tall.

Rotation

A rotation by angle `θ` (counter-clockwise) uses:

ROTATION MATRIX

`R(θ) = [[cos θ, −sin θ], [sin θ, cos θ]]`

For `θ = 90°`: `cos(90°) = 0`, `sin(90°) = 1`, so `R(90°) = [[0, −1], [1, 0]]`. Applied to `v = [3, 4]`: `R v = [0×3 + (−1)×4, 1×3 + 0×4] = [−4, 3]`.

A ROTATION MATRIX ALWAYS PRESERVES LENGTH

`|v| = √(3² + 4²) = 5`, and `|R v| = √((−4)² + 3²) = √25 = 5` — identical. A genuine rotation never stretches or shrinks anything, which is a useful sanity check: if a "rotation" matrix changes a vector's length, it isn't actually a pure rotation.

Reflection

Reflecting across the x-axis flips the sign of y only; reflecting across the y-axis flips the sign of x only:

REFLECTION	MATRIX	V = [3, 4] BECOMES
Across the x-axis	[[1, 0], [0, −1]]	[3, −4]
Across the y-axis	[[−1, 0], [0, 1]]	[−3, 4]

Shear

A shear slides one axis's coordinates sideways in proportion to the other axis, turning a rectangle into a parallelogram without changing its area:

HORIZONTAL SHEAR MATRIX

`Sh(k) = [[1, k], [0, 1]]`

With `k = 1`: `Sh v = [1×3 + 1×4, 0×3 + 1×4] = [7, 4]` — x shifted by an amount proportional to y, y unchanged.

Composing Transformations — And Why Order Matches Chapter 4

To apply two transformations in sequence, multiply their matrices together — and because Chapter 4 already established that matrix multiplication isn't commutative, **the order genuinely changes the result**. `R × S` means "apply `S` first, then `R`" (read right to left, matching how it's applied to a vector: `(R × S) v = R (S v)`).

Reusing `S = [[2, 0], [0, 0.5]]`, `R = [[0, -1], [1, 0]]`, and `v = [3, 4]`:

ORDER	MEANING	COMBINED MATRIX	RESULT ON V
R × S	Scale first, then rotate	[[0, −0.5], [2, 0]]	[−2, 6]
S × R	Rotate first, then scale	[[0, −2], [0.5, 0]]	[−8, 1.5]

SAME TWO OPERATIONS, GENUINELY DIFFERENT OUTCOMES

Scaling `v` by 2× horizontally and 0.5× vertically, then rotating 90° gives `[-2, 6]`. Rotating 90° first, then applying that exact same scale, gives `[-8, 1.5]` — a visibly different point. This is exactly why 3D modelling and game-engine transform stacks are so careful about ordering "scale, then rotate, then translate" consistently — swapping the order silently changes the result.

Homogeneous Coordinates — Why Translation Needs a Trick

Every transformation above is **linear** — algebraically, a 2×2 matrix multiplication always sends the origin `[0, 0]` to itself (`M × [0,0] = [0,0]`, for any `M`). Translation — sliding every point by a fixed offset — genuinely moves the origin, so no plain 2×2 matrix can represent it at all.

The fix is **homogeneous coordinates**: pad every 2D vector with an extra `1`, turning `[x, y]` into `[x, y, 1]`, and use a 3×3 matrix. The extra row and column let a translation "leak" into the result through that constant 1:

TRANSLATION MATRIX (HOMOGENEOUS FORM)

$$T = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Translating `v = [3, 4]` (as `[3, 4, 1]`) by $(t_x, t_y) = (5, -2)$: $T \times [3, 4, 1] = [1\times3 + 0\times4 + 5\times1,\ 0\times3 + 1\times4 + (-2)\times1,\ 1] = [8, 2, 1]$ — the point `(8, 2)`, exactly `(3+5, 4−2)`, as expected.

WHY THIS MATTERS FOR GRAPHICS TOOLS

Every scaling, rotation, and shear matrix from earlier in this chapter also has a 3×3 homogeneous version (just pad with a row/column of `[0, 0, 1]`), which is precisely why every practical transform in Blender, Figma, or a game engine can be combined into a single matrix multiplication chain — translation included — instead of treating translation as a special case handled separately from everything else.

Hands-On Exercises

EXERCISE 1

Given `w = [2, -1]`, apply the scaling matrix `S = [[3, 0], [0, 2]]`, then separately apply the 90° rotation matrix `R = [[0, -1], [1, 0]]` to the *original* `w` (not the scaled result). Show both calculations, and verify the rotated result has the same magnitude as `w`.

View solution

EXERCISE 2

Using `S = [[3, 0], [0, 2]]` and `R = [[0, -1], [1, 0]]` from Exercise 1, compute the combined matrix `R × S` and the combined matrix `S × R`, then apply each combined matrix to `w = [2, -1]`. Confirm the two final results are different, and state in one sentence which real-world order each one represents ("scale then rotate" or "rotate then scale").

View solution

EXERCISE 3

Using homogeneous coordinates, translate the point `[2, -1]` by $(t_x, t_y) = (-3, 4)$. Write out the 3×3 translation matrix, the homogeneous form of the point, and the full matrix-vector multiplication. Then explain, using this chapter's own "translation moves the origin" argument, why no ordinary 2×2 matrix could have achieved the same result.

View solution

CHAPTER 5 QUICK REFERENCE

- **Scaling:** $\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$ — stretches/shrinks each axis independently
- **Rotation:** $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$ — always preserves length
- **Reflection:** flip the sign of one axis's coefficient in the identity matrix
- **Shear:** $\begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}$ — slides one axis proportionally to the other, preserving area
- **Composing:** multiply the matrices; `A × B` means "apply B first, then A" — order matters, per Chapter 4's non-commutativity
- A plain matrix transformation always fixes the origin — translation genuinely can't be represented by a 2×2 matrix alone
- **Homogeneous coordinates:** pad vectors with a 1 and use a 3×3 matrix so translation becomes just another matrix multiplication
- **Next chapter:** Systems of linear equations and Gaussian elimination

CHAPTER 6 OF 10

Systems of Linear Equations & Gaussian Elimination

Linear Algebra Fundamentals

Chapter 6 · Systems of Linear Equations & Gaussian Elimination

A **system of linear equations** is just several linear equations sharing the same unknowns, all required to hold at once. Chapters 4 and 5 already built the exact notation for this: a system is `A x = b`, where `A` is a matrix of coefficients, `x` is the vector of unknowns being solved for, and `b` is the vector of right-hand-side values.

From Equations to a Matrix

Consider the system:

A WORKED SYSTEM

`x + y = 5`

`2x − y = 1`

This is `A x = b` with:

```
A = [[1,  1],
     [2, -1]]

x = [x, y]   # the unknowns
b = [5, 1]
```

Gaussian Elimination — Systematic Row Reduction

Rather than guessing or using substitution tricks, **Gaussian elimination** works on the **augmented matrix** — `A` with `b` tacked on as an extra column — using three **row operations** that are guaranteed never to change the solution set:

- Swap two rows
- Multiply a row by a nonzero scalar
- Add a multiple of one row to another row

The goal is to use these operations to eliminate variables from rows, one at a time, until the system is easy to solve by simple substitution.

STEP	AUGMENTED MATRIX [A | B]
Start	[1, 1 | 5] [2, −1 | 1]
R2 → R2 − 2·R1	[1, 1 | 5] [0, −3 | −9]

The second row now reads `0x − 3y = −9`, which only involves `y` — the `x` term has been eliminated. This is called **row echelon form**: each row's leading (leftmost nonzero) entry sits strictly to the right of the row above it.

Back-Substitution

Once the system is in echelon form, solve from the bottom row upward, substituting each known value into the row above it:

STEP	WORKING
Solve row 2 for y	−3y = −9 → y = 3
Substitute into row 1	x + 3 = 5 → x = 2

So `x = 2`, `y = 3`. Checking against the original second equation: `2(2) − 3 = 4 − 3 = 1` ✓.

```
def gaussian_eliminate_2x2(A, b):
    # Eliminate x from row 1 using row 0
    factor = A[1][0] / A[0][0]
    A[1] = [A[1][i] - factor * A[0][i] for i in range(2)]
    b[1] = b[1] - factor * b[0]

    # Back-substitute
    y = b[1] / A[1][1]
    x = (b[0] - A[0][1] * y) / A[0][0]
    return x, y

print(gaussian_eliminate_2x2([[1, 1], [2, -1]], [5, 1]))  # (2.0, 3.0)
```

When There's No Solution

Consider instead:

> **AN INCONSISTENT SYSTEM**
>
> `2x + 2y = 4`
>
> `x + y = 5`

Eliminating: `R1 → R1 − 2·R2` gives `[0, 0 | −6]` — the row `0x + 0y = −6`, which is never true for any `x` or `y`. Geometrically, these are two parallel lines (both have the same slope, `x + y = k` for different `k`) that never intersect. Whenever elimination produces a row that's all zeros on the left but nonzero on the right, the system has **no solution**.

When There Are Infinitely Many Solutions

Now consider:

A DEPENDENT SYSTEM

`x + y = 3`

`2x + 2y = 6`

Eliminating: `R2 → R2 − 2·R1` gives `[0, 0 | 0]` — the row `0x + 0y = 0`, which is *always* true, for any `x` and `y`. The second equation was never independent information at all — it's exactly the first equation multiplied by 2. Whenever elimination produces a row that's entirely zero on both sides, that equation contributed nothing new, and there's a **free variable**: letting `x = t` for any value `t`, `y = 3 − t` — every point on the line `x + y = 3` is a valid solution.

FORWARD REFERENCE — THE DETERMINANT WILL EXPLAIN THIS INSTANTLY

Computing `det(A) = a₁₁a₂₂ − a₁₂a₂₁` (Chapter 7's own subject) for each of the three systems above: the first system's matrix has determinant `1(−1) − 1(2) = −3` (nonzero — unique solution). Both the no-solution and infinite-solution systems have coefficient matrices with determinant `0`. A zero determinant is exactly the signal that a system *won't* have a single unique solution — Chapter 7 explains precisely why.

Systems in Code — NumPy

```
import numpy as np

A = np.array([[1, 1], [2, -1]])
b = np.array([5, 1])

x = np.linalg.solve(A, b)
print(x)  # [2. 3.]

# np.linalg.solve raises LinAlgError for a singular (zero-determinant) A —
# it does NOT silently return a wrong answer for the no-solution / infinite cases
```

Hands-On Exercises

EXERCISE 1

Solve the system `3x + y = 11`, `x − y = 1` using Gaussian elimination on the augmented matrix. Show the row operation you use to eliminate `x` from the second row, the resulting echelon form, and the back-substitution steps. Check your final answer against both original equations.

View solution

EXERCISE 2

Apply Gaussian elimination to `4x + 6y = 10`, `2x + 3y = 8`. What does the resulting row tell you about this system? Name which of this chapter's two "no unique solution" cases it falls into, and explain geometrically what's true about the two lines these equations represent.

View solution

EXERCISE 3

A teammate's code calls `np.linalg.solve(A, b)` inside a loop that processes many different systems, and it occasionally crashes with a `LinAlgError`. Explain, using this chapter's own material, what property of a particular `A` matrix would cause that crash, and why that crash is actually more useful behavior than the function silently returning some plausible-looking numbers instead.

View solution

CHAPTER 6 QUICK REFERENCE

- A system of linear equations is `A x = b` — a matrix of coefficients, a vector of unknowns, a vector of results
- **Row operations** (swap, scale, add a multiple of one row to another) never change a system's solution set
- **Gaussian elimination:** use row operations to reach echelon form, then solve via **back-substitution** from the bottom row up
- A row of all zeros on the left but nonzero on the right (`0 = k`, k ≠ 0) means **no solution** — geometrically, parallel lines that never meet
- A row of all zeros on both sides (`0 = 0`) means a **free variable** and infinitely many solutions — one equation added no new information
- A zero determinant (Chapter 7 forward reference) is the single-number signal that a system won't have a unique solution
- **Next chapter:** The determinant and matrix inverse

CHAPTER 7 OF 10

The Determinant & Matrix Inverse

Linear Algebra Fundamentals

Chapter 7 · The Determinant & Matrix Inverse

Chapter 6 ended with a forward reference: a single number, computed directly from a matrix, that instantly reveals whether a system of equations has a unique solution. That number is the **determinant**. This chapter defines it, gives it a genuine geometric meaning, and uses it to build the **matrix inverse** — the closest thing a matrix has to "dividing by" itself.

The Determinant Formula

For a 2×2 matrix, the determinant is refreshingly simple:

2×2 DETERMINANT

`det([[a, b], [c, d]]) = ad − bc`

This is exactly the calculation Chapter 6 used to flag a "no unique solution" system, and it's exactly Chapter 3's own 2D "pseudo-cross-product" ($a_1b_2 - a_2b_1$) — stacking two vectors as the rows of a matrix and taking the determinant is the same number as taking their 2D cross product.

Geometric Meaning: Area Scaling

A matrix, per Chapter 5, transforms every point in the plane. The unit square (corners at `[0,0]`, `[1,0]`, `[0,1]`, `[1,1]`, area exactly 1) gets mapped to some parallelogram — and `|det(A)|` is *exactly* that parallelogram's area. The determinant is the transformation's area-scaling factor.

TRANSFORMATION (FROM CH.5)	MATRIX	DET	WHAT IT MEANS
Scale (2×, 0.5×)	[[2, 0], [0, 0.5]]	1.0	Stretched one way, squeezed the other — net area unchanged
Rotate 90°	[[0, -1], [1, 0]]	1	Area exactly preserved — rotations never change area
Reflect (x-axis)	[[1, 0], [0, -1]]	-1	Area magnitude preserved, but the sign flips

WHAT A NEGATIVE DETERMINANT MEANS

The *sign* of the determinant reveals whether a transformation flips orientation — whether a shape's "clockwise" and "counter-clockwise" get swapped, the way a mirror reflection does. A positive determinant preserves orientation (like the rotation above); a negative one reverses it (like the reflection above), even though both have the same area-scaling magnitude, `|det| = 1`.

The Determinant as a Singularity Test

If `det(A) = 0`, the transformation squashes the entire plane down onto a line (or a single point) — all area is destroyed, mapped to zero. A matrix with a zero determinant is called **singular**. This is precisely why Chapter 6's two "no unique solution" systems both had determinant 0: their coefficient matrices collapse the plane, so there's either no point that lands exactly on `b`, or a whole line of points that do.

The Matrix Inverse

The **inverse** of a matrix `A`, written `A⁻¹`, is the matrix satisfying `A A⁻¹ = A⁻¹ A = I` — the matrix equivalent of a reciprocal. For 2×2 matrices, it has a direct formula built from the determinant:

2×2 INVERSE FORMULA

`A⁻¹ = (1 / det(A)) × [[d, −b], [−c, a]]`

Worked example, reusing Chapter 6's own system matrix `A = [[1, 1], [2, -1]]`, `det(A) = -3`:

STEP	RESULT
Swap diagonal, negate off-diagonal	[[-1, -1], [-2, 1]]
Multiply by 1/det = 1/(-3)	A^{-1} = [[1/3, 1/3], [2/3, -1/3]]

Verifying `A × A⁻¹ = I`: `[[1×⅓+1×⅔, 1×⅓+1×(-⅓)], [2×⅓+(-1)×⅔, 2×⅓+(-1)×(-⅓)]] = [[1, 0], [0, 1]]` ✓.

WHEN THE INVERSE DOESN'T EXIST

The formula divides by `det(A)` — so whenever `det(A) = 0`, the inverse is undefined. This is exactly consistent with the geometric picture: a transformation that collapses the plane to a line has thrown information away, and there's no way to "un-throw" it back. A singular matrix has no inverse, full stop.

Solving a System With the Inverse

Since `A x = b` and `A⁻¹ A = I`, multiplying both sides by `A⁻¹` gives `x = A⁻¹ b` directly — no elimination needed, if the inverse is already known. Reusing Chapter 6's exact system (`A = [[1,1],[2,-1]]`, `b = [5,`

1]): `A⁻¹ b = [⅗×5 + ⅕×1, ⅗×5 + (-⅕)×1] = [2, 3]` — exactly the `x = 2, y = 3` Chapter 6 found by elimination.

WHY ELIMINATION IS USUALLY PREFERRED IN PRACTICE ANYWAY

Computing a full inverse and then multiplying is more arithmetic than directly eliminating toward the answer, and for large systems it's also less numerically stable — small floating-point errors get amplified more. Real numerical libraries almost always solve `A x = b` using elimination-based methods internally (Chapter 6's own `np.linalg.solve`), even when they could compute `A⁻¹` instead. The inverse is conceptually clean and useful when the *same* `A` needs to be solved against many different `b` vectors, but it's rarely the practical first choice for a single system.

Determinants & Inverses in Code

```
import numpy as np

A = np.array([[1, 1], [2, -1]])

print(np.linalg.det(A))   # -3.0
print(np.linalg.inv(A))   # [[0.333  0.333] [0.667 -0.333]]

# Both raise/return inf or nan for a singular matrix — never a plausible-looking wrong answer
singular = np.array([[2, 2], [1, 1]])
print(np.linalg.det(singular))  # 0.0
# np.linalg.inv(singular) raises LinAlgError: Singular matrix
```

Hands-On Exercises

EXERCISE 1

Compute the determinant of `M = [[3, 2], [1, 4]]`. Since it's nonzero, compute the full inverse `M⁻¹` using this chapter's own formula, and verify your answer by computing `M × M⁻¹` and confirming it equals the identity matrix.

View solution

EXERCISE 2

Reusing Chapter 6's Exercise 1 system (`3x + y = 11` , `x − y = 1` , coefficient matrix `A = [[3, 1], [1, -1]]`), compute `A⁻¹` and use it to solve for `x = A⁻¹ b` with `b = [11, 1]` . Confirm your answer matches the `x = 3, y = 2` found by Gaussian elimination in Chapter 6.

View solution

EXERCISE 3

Given `N = [[6, 3], [4, 2]]`, compute its determinant and state whether `N` has an inverse. Then look at `N`'s two columns, `[6, 4]` and `[3, 2]`, and explain — in terms of one column being a scalar multiple of the other — why this particular matrix was always going to be singular, before you even computed the determinant.

View solution

CHAPTER 7 QUICK REFERENCE

- **2×2 determinant:** `det([[a,b],[c,d]]) = ad − bc` — same formula as Chapter 3's 2D cross product
- **Geometric meaning:** `|det(A)|` is the transformation's area-scaling factor; the **sign** reveals whether orientation flips (negative = reflection-like)
- **Singular matrix:** `det(A) = 0` — the transformation collapses the plane to a line or point, and no inverse exists
- A zero determinant is exactly why a Chapter 6 system has no unique solution — no solution, or infinitely many
- **2×2 inverse:** `A⁻¹ = (1/det(A)) × [[d, −b], [−c, a]]`, satisfying `A A⁻¹ = A⁻¹ A = I`
- A system can be solved as `x = A⁻¹ b`, but Gaussian elimination is usually preferred in practice for efficiency and numerical stability
- **Next chapter:** Vector spaces, span, basis & dimension

CHAPTER 8 OF 10

Vector Spaces, Span, Basis & Dimension

Linear Algebra Fundamentals

Chapter 8 · Vector Spaces, Span, Basis & Dimension

This chapter is the most abstract in the course, so it stays deliberately concrete: no axioms, no formal proofs — just the four ideas (span, linear independence, basis, dimension) explained through the same kind of worked, checkable examples used throughout, reusing Chapter 6's row reduction and Chapter 7's determinant as the actual tools that answer these questions.

Linear Combinations & Span

A **linear combination** of vectors is just Chapter 2's addition and scalar multiplication, applied together: $c_1v_1 + c_2v_2 + \dots$ for any scalars $c_1, c_2, \dots$. The **span** of a set of vectors is the set of *every* linear combination reachable from them — everywhere those vectors, scaled and combined in every possible way, can reach.

VECTORS	THEIR SPAN
A single nonzero vector, e.g. [2, 1]	The entire line through the origin and [2, 1] — every scalar multiple of it
Two non-parallel vectors, e.g. [2, 1] and [1, 3]	The *entire* 2D plane — any point [x, y] can be reached by some combination
Two parallel vectors, e.g. [2, 4] and [1, 2]	Still just a line — the second vector adds nothing new, since [1, 2] = 0.5 × [2, 4]

Linear Independence — Testing With Chapter 7's Determinant

A set of vectors is **linearly independent** if none of them is redundant — none can be written as a combination of the others. For exactly two vectors in 2D, Chapter 7's determinant gives an instant test: **nonzero determinant means independent** (and their span is the whole plane); **zero determinant means dependent** (redundant — their span collapses to a line).

PAIR	DET (AS A MATRIX OF ROWS)	INDEPENDENT?
v_1 = [2, 1], v_2 = [1, 3]	2(3) − 1(1) = 5	Yes — spans all of R^2
w_1 = [2, 4], w_2 = [1, 2]	2(2) − 4(1) = 0	No — w_2 = 0.5 × w_1, redundant

Linear Independence for More Vectors — Chapter 6's Row Reduction

The determinant shortcut only works for exactly two vectors in 2D. For any other number of vectors, or higher dimensions, the general tool is exactly Chapter 6's **row reduction**: stack the vectors as rows of a matrix and eliminate. Every row that reduces to **all zeros** represents a vector that added no new information — it was already reachable from the others. The number of surviving nonzero rows is called the **rank** of the set.

Worked example: $u_1 = [1, 2, 1]$, $u_2 = [0, 1, 2]$, $u_3 = [1, 3, 3]$. Notice first that $u_1 + u_2 = [1, 3, 3] = u_3$ exactly — a strong hint this set is dependent.

STEP	ROWS
Start	[1,2,1] / [0,1,2] / [1,3,3]
R3 → R3 − R1	[1,2,1] / [0,1,2] / [0,1,2]
R3 → R3 − R2	[1,2,1] / [0,1,2] / [0,0,0]

Row 3 reduced to all zeros — confirming u_3 really was redundant. The rank is **2**, not 3: this set of three vectors spans only a 2D plane sitting inside 3D space, not the full 3D space.

Basis

A **basis** for a space is a set of vectors that is *both* linearly independent *and* spans the entire space — the minimal, non-redundant "building block" set that reaches everywhere. The simplest example is the **standard basis**: $i = [1, 0]$ and $j = [0, 1]$ for the 2D plane — independent ($\det = 1 \neq 0$), and obviously spanning everything, since any point $[x, y] = x \cdot i + y \cdot j$.

Testing whether $c = [1, 0, 0]$, $d = [0, 1, 0]$, $e = [1, 1, 1]$ form a basis for 3D space, by row reduction:

STEP	ROWS
Start	[1,0,0] / [0,1,0] / [1,1,1]
R3 → R3 − R1	[1,0,0] / [0,1,0] / [0,1,1]
R3 → R3 − R2	[1,0,0] / [0,1,0] / [0,0,1]

All three rows survive — rank 3, fully independent. Since there are exactly three independent vectors in 3D space, they automatically span all of it: $\{c, d, e\}$ is a valid basis.

Dimension

A FACT WORTH STATING PLAINLY, WITHOUT A FULL PROOF

Every basis for a given space has exactly the same number of vectors — this number is the space's **dimension**. The 2D plane always needs exactly 2 independent vectors for a basis, never 1 (too few to reach everywhere) or 3 (the third is guaranteed redundant, per this chapter's own row reduction argument). This is why "dimension" isn't a vague description — it's a precise, countable property of a space, computed directly from its own basis size.

RANK VS. DIMENSION — A DISTINCTION WORTH KEEPING STRAIGHT

Dimension describes the space itself (2D, 3D, ...). **Rank** describes how many independent vectors a *particular set* actually contributes — which can be less than the space's own dimension, exactly as Chapter 8's own $\{u_1, u_2, u_3\}$ example showed (rank 2, inside a 3-dimensional space). A set's rank is always less than or equal to the dimension of the space it lives in.

Span, Basis & Rank in Code

```
import numpy as np

vectors = np.array([[1, 2, 1], [0, 1, 2], [1, 3, 3]])
print(np.linalg.matrix_rank(vectors))  # 2 — matches the hand row-reduction above

basis_candidate = np.array([[1, 0, 0], [0, 1, 0], [1, 1, 1]])
print(np.linalg.matrix_rank(basis_candidate))  # 3 — full rank, a genuine basis for R^3
```

Hands-On Exercises

EXERCISE 1

Given `a = [3, 1]` and `b = [6, 2]`, use the Chapter 7 determinant shortcut to determine whether they're linearly independent. If they're not, express `b` as a scalar multiple of `a`, and state what their span actually is (a line, or the whole plane).

View solution

EXERCISE 2

Given `p = [1, 1, 0]`, `q = [0, 1, 1]`, `r = [1, 2, 1]`, use row reduction (stacking them as rows and eliminating) to find the rank of this set. State whether they're linearly independent, and if not, express the redundant vector as a combination of the other two.

View solution

EXERCISE 3

Given `m = [2, 5]` and `n = [-1, 3]`, determine whether `{m, n}` forms a basis for the 2D plane. Show the calculation, and explain in one sentence why exactly two independent vectors are both necessary and sufficient for a basis of a 2-dimensional space — no more, no fewer.

View solution

CHAPTER 8 QUICK REFERENCE

- **Linear combination:** $c_1v_1 + c_2v_2 + \dots$; **span** is the set of every linear combination reachable from a set of vectors
- **Linear independence, 2 vectors in 2D:** nonzero determinant (Ch.7) — independent; zero determinant — dependent, redundant
- **Linear independence, general case:** row-reduce (Ch.6) the vectors as rows; a row reducing to all zeros means that vector was redundant
- **Rank:** the number of surviving nonzero rows after elimination — how many genuinely independent vectors a set actually contains
- **Basis:** a linearly independent set that also spans the whole space — the minimal non-redundant building blocks
- **Dimension:** the number of vectors in any basis for a space — always the same number, regardless of which basis is chosen
- Rank ≤ dimension of the space; a set's rank being less than the space's dimension means that set doesn't span everything
- **Next chapter:** Eigenvalues and eigenvectors

CHAPTER 9 OF 10

Eigenvalues & Eigenvectors

Linear Algebra Fundamentals

Chapter 9 · Eigenvalues & Eigenvectors

Most vectors, transformed by a matrix, get both moved to a new direction *and* scaled. This chapter is about the special exceptions: directions a given transformation only **scales**, never rotates off their own line. Those special directions — and how much they get scaled by — turn out to be some of the most useful numbers in all of linear algebra.

The Definition

EIGENVECTOR / EIGENVALUE EQUATION

`A v = λ v` — for a nonzero vector `v` (the **eigenvector**) and a scalar `λ` (the **eigenvalue**), applying `A` to `v` produces exactly `v` scaled by `λ` — nothing else.

The simplest possible example reuses Chapter 5's own reflection matrix, `Rx = [[1, 0], [0, -1]]` (reflection across the x-axis) — since it's diagonal, its eigenvalues are just the diagonal entries themselves:

EIGENVECTOR	EIGENVALUE	GEOMETRIC MEANING
[1, 0]	1	Points on the x-axis are completely unchanged by this reflection
[0, 1]	-1	Points on the y-axis get flipped to the opposite side — same line, reversed direction

Both stay on their own line — one unmoved, one flipped — which is exactly what makes them eigenvectors of this particular transformation.

Finding Eigenvalues — The Characteristic Equation

For a matrix without such an obviously convenient diagonal shape, eigenvalues need to be solved for directly. Starting from `A v = λ v`, rewrite as `A v − λ v = 0`, then `(A − λI) v = 0`. For a *nonzero* `v` to satisfy this, `(A − λI)` can't have an inverse — per Chapter 7, that means its determinant must be exactly zero:

THE CHARACTERISTIC EQUATION

`det(A − λI) = 0`

For a 2×2 matrix `A = [[a, b], [c, d]]`, this expands to a quadratic in `λ`:

```
# det([[a-L, b], [c, d-L]]) = (a-L)(d-L) - bc = 0
# expands to:  L^2 - (a+d)L + (ad - bc) = 0
#              L^2 - trace(A)*L + det(A) = 0
```

A USEFUL SHORTCUT WORTH REMEMBERING

The two coefficients of that quadratic are the matrix's **trace** (sum of the diagonal) and its **determinant** (Chapter 7). This means the sum of the two eigenvalues always equals the trace, and their product always equals the determinant — a fast way to sanity-check a computed pair of eigenvalues without redoing the full algebra.

Worked Example

Let `A = [[2, 1], [1, 2]]`. Trace `= 4`, determinant `= 2(2) − 1(1) = 3`. The characteristic equation is:

STEP	WORKING
Characteristic equation	$\lambda^2 - 4\lambda + 3 = 0$
Factor	$(\lambda - 1)(\lambda - 3) = 0$
Eigenvalues	$\lambda = 1$, $\lambda = 3$ — sum = 4 ✓ trace, product = 3 ✓ det

To find each eigenvector, substitute the eigenvalue back into `(A − λI) v = 0` and solve — this is exactly a homogeneous version of Chapter 6's system-solving, and it's guaranteed to have a whole line of solutions (not just `v = 0`), since `(A − λI)` is singular by construction.

Λ	A − ΛI	EQUATION FROM ROW 1	EIGENVECTOR DIRECTION
1	[[1, 1], [1, 1]]	$v_1 + v_2 = 0 \rightarrow v_2 = -v_1$	[1, −1]
3	[[-1, 1], [1, -1]]	$-v_1 + v_2 = 0 \rightarrow v_2 = v_1$	[1, 1]

Checking both: `A[1,-1] = [2−1, 1−2] = [1,-1] = 1×[1,-1]` ✓, and `A[1,1] = [2+1, 1+2] = [3,3] = 3×[1,1]` ✓.

A NON-EIGENVECTOR, FOR CONTRAST

Applying the same `A` to `v = [1, 0]` (not one of the eigenvectors above) gives `[2, 1]` — pointing in a genuinely different direction (rotated about 26.6° away from the x-axis), not just a scaled copy of `[1, 0]`. Most vectors behave this way; only the two special directions found above don't.

Repeated Application & the Dominant Eigenvalue

Applying `A` over and over to almost any starting vector reveals something striking: the result increasingly lines up with the eigenvector belonging to the **largest-magnitude** eigenvalue. Starting from `v₀ = [1, 0]` (not an eigenvector) and repeatedly applying `A = [[2,1],[1,2]]`:

N	$A^n v_0$	NORMALIZED DIRECTION
1	[2, 1]	[0.894, 0.447]
2	[5, 4]	[0.781, 0.625]
3	[14, 13]	[0.733, 0.680]
5	[122, 121]	[0.710, 0.704]

The direction is visibly converging toward `[0.707, 0.707]` — the normalized form of the `λ = 3` eigenvector, `[1, 1]`, which is the **larger** of the two eigenvalues. And the magnitude roughly triples at each step for large `n`, matching that same dominant eigenvalue.

Real Relevance

APPLICATION	HOW EIGENVALUES/EIGENVECTORS ARE USED
Principal Component Analysis (dimensionality reduction)	The eigenvectors of a dataset's covariance matrix are the directions of maximum spread ("variance"); the eigenvalues rank how much variance each direction captures. Keeping only the top few eigenvectors is exactly how a 50-column dataset gets reduced to its 2 or 3 most informative directions — Chapter 1's own forward reference, finally paid off.
Google's original PageRank algorithm	Models the web as a giant transition matrix (probability of clicking from one page to another). The steady-state importance ranking is the eigenvector belonging to eigenvalue 1 — precisely the "repeated application converges to the dominant eigenvector" behavior demonstrated above.
Stability analysis	For a system that gets repeatedly transformed (a simulation step, a feedback loop), whether it settles down or blows up depends entirely on the largest eigenvalue's magnitude: `< 1` shrinks toward zero over time (stable), `> 1` grows without bound (unstable), exactly as this chapter's own repeated-application table demonstrated with λ = 3.

Eigenvalues & Eigenvectors in Code

```python
import numpy as np

A = np.array([[2, 1], [1, 2]])
eigenvalues, eigenvectors = np.linalg.eig(A)

print(eigenvalues)   # [1. 3.] — matches the hand calculation above
print(eigenvectors)  # columns are the (normalized) eigenvectors
```

Hands-On Exercises

EXERCISE 1

Find both eigenvalues of `C = [[4, 2], [1, 3]]` using the characteristic equation, then find the corresponding eigenvector for each. Verify both eigenpairs by computing `C v` and confirming it equals `λ v`.

View solution

EXERCISE 2

A colleague claims `λ = 4`, `v = [2, 1]` is a genuine eigenpair of `D = [[5, -2], [1, 2]]`. Check this directly by computing `D v` and comparing it to `4v`. Then independently verify using the characteristic equation (trace and determinant) that 4 really is one of `D`'s eigenvalues.

View solution

EXERCISE 3

A repeated transformation has two eigenvalues: $\lambda_1 = 0.5$ (eigenvector direction u_1) and $\lambda_2 = 1.5$ (eigenvector direction u_2). If this transformation is applied many times in a row to a generic starting vector that isn't aligned with either eigenvector, explain — using this chapter's own repeated-application finding — which eigenvector's direction the result will end up dominated by, and whether the overall magnitude will grow, shrink, or stay bounded.

View solution

CHAPTER 9 QUICK REFERENCE

- **Eigenvector/eigenvalue:** `A v = λ v` — a direction the transformation only scales, never rotates off its own line
- **Characteristic equation:** `det(A − λI) = 0`, derived from Chapter 7's singularity requirement for a nonzero solution to exist
- For 2×2 matrices: `λ² − trace(A)λ + det(A) = 0` — sum of eigenvalues = trace, product of eigenvalues = determinant
- Once λ is known, solve `(A − λI)v = 0` (Chapter 6-style) to find the matching eigenvector direction

- Repeated application of a matrix drives almost any vector toward the eigenvector of the **largest-magnitude** eigenvalue — the "dominant" eigenvector
- Real applications: PCA (dimensionality reduction), PageRank (steady-state ranking via the eigenvalue-1 eigenvector), stability analysis (dominant eigenvalue magnitude above/below 1)
- **Next chapter:** Capstone — applying every chapter's material to a single worked access-control system

CHAPTER 10 OF 10

Capstone — Linear Algebra in Practice

Linear Algebra Fundamentals

Chapter 10 · Capstone — Linear Algebra in Practice

One continuous worked project, touching every chapter of this course in the order a real engineer would actually reach for each idea: building a small 2D sprite transform pipeline for a game, then using the exact same mathematics to analyze where players click on the resulting screen.

Part 1 — A Sprite Transform Pipeline

1 — FACING DIRECTION, AS A VECTOR (CH.2)

A sprite sits at world position `p = [10, 5]`, facing along `f = [1, 0]`. The player is at `q = [13, 9]`, so the direction from the sprite to the player is `d = q − p = [3, 4]`, magnitude `5`. The dot product `f · d = 3` is positive, and `cos(θ) = 3/(1×5) = 0.6`, giving `θ ≈ 53.1°` — the player is roughly ahead of the sprite, well within a typical detection cone.

2 — WHICH WAY TO TURN, VIA THE 2D CROSS PRODUCT (CH.3)

Knowing the player is *roughly* ahead isn't enough for AI logic — turning left or right needs a side. The 2D pseudo-cross-product from Chapter 3, $f_1d_2 - f_2d_1 = (1)(4) - (0)(3) = 4$, is positive, meaning `d` is counter-clockwise from `f` — the player is to the sprite's **left**. This single sign is the entire "which way should the AI turn" decision.

3 — COMBINING A FLIP AND A SCALE, CORRECTLY (CH.4)

This sprite needs to render mirrored (it's facing left in its base art) and scaled up 1.5×. Combining a flip matrix `Flip = [[-1, 0], [0, 1]]` and a scale matrix `Sc = [[1.5, 0], [0, 1.5]]` uses Chapter 4's real matrix multiplication, not entry-wise multiplication — and here, `Flip × Sc = Sc × Flip = [[-1.5, 0], [0, 1.5]]`.

A GENUINE EXCEPTION TO "ORDER MATTERS"

Chapter 4 warned `A × B ≠ B × A` "in general" — and diagonal matrices are exactly the honest exception: since each only scales along its own axis independently, two diagonal matrices always commute. It's still worth

combining them with real matrix multiplication rather than assuming this always holds — the moment a rotation enters the mix, order matters again immediately.

4 — PLACING THE SPRITE IN THE WORLD (CH.5)

Packaging the combined flip-and-scale matrix together with the translation to `p = [10, 5]` needs Chapter 5's homogeneous coordinates: `T = [[-1.5, 0, 10], [0, 1.5, 5], [0, 0, 1]]`. Applying `T` to the sprite's own local origin `[0, 0, 1]` gives `[10, 5, 1]` — exactly Step 1's world position `p`, confirming the transform is correct. Applying it to a corner point of the sprite's local bounding box, `[2, 1, 1]`, gives `[7, 6.5, 1]` — that corner's actual position in the world.

5 — CALIBRATING WORLD UNITS TO SCREEN PIXELS (CH.6)

The renderer needs a mapping from world x-coordinates to screen pixels, `pixel = m·world + c`. Two known calibration points — world `2 → pixel 100`, world `5 → pixel 250` — give the system `2m + c = 100`, `5m + c = 250`. Gaussian elimination (`R2 → R2 − 2.5·R1`) gives `0m − 1.5c = 0 → c = 0`, then back-substitution gives `2m = 100 → m = 50`. So `pixel = 50 × world` — checked against both original points.

6 — UN-CLICKING: SCREEN SPACE BACK TO SPRITE-LOCAL SPACE (CH.7)

A player clicks at world point `[7, 6.5]` — Step 4's own transformed corner. Before inverting, check the sprite's transform isn't degenerate: `det([[-1.5, 0], [0, 1.5]]) = -2.25`, nonzero — safe to invert. The inverse is `[[-2/3, 0], [0, 2/3]]`. Subtracting the translation first (`[7,6.5] − [10,5] = [-3, 1.5]`) and applying the inverse gives `[2, 1]` — exactly Step 4's original local corner. The click round-trips perfectly back to sprite-local space.

7 — CHECKING THE INPUT SCHEME FOR REDUNDANCY (CH.8)

The control scheme offers three inputs: `up = [0, 1]`, `right = [1, 0]`, and a diagonal `up-right = [1, 1]` (both keys held at once). Row-reducing the three as rows shows `up + right = [1, 1] = up-right` exactly — the third row reduces to `[0, 0]`. Rank 2, not 3: the diagonal input adds no genuinely new reachable direction beyond combining the other two, which is exactly what a player pressing both keys already produces.

Part 2 — Feeding Into Player-Click Analysis

8 — WHERE DO PLAYERS ACTUALLY CLICK? PCA ON REAL COORDINATES (CH.9)

A UI element sits at the sprite's own world position, `[10, 5]` (Step 1). Four recorded player clicks, centered around it, come out (relative to that center) as `(1,1)`, `(-1,-1)`, `(2,2)`, `(-2,-2)` — every click lies exactly on the line `y = x`. Computing the covariance matrix from these points: `Cov = [[2.5, 2.5], [2.5, 2.5]]`.

Solving the characteristic equation (trace 5, det 0): `λ² − 5λ = 0 → λ = 0 or λ = 5`.

λ	EIGENVECTOR	WHAT IT MEANS
5	[1, 1]	The direction of maximum spread — **all** the click variance lives along this diagonal
0	[1, -1]	Zero spread perpendicular to that diagonal — this direction carries no information at all

CHAPTER 1'S FORWARD REFERENCE, FINALLY PAID OFF

Since one eigenvalue is exactly 0, this dataset can be reduced from 2 numbers per click down to **1** — the coordinate along `[1, 1]` — with zero information lost. This synthetic dataset was deliberately built to be this clean; real click data is essentially never perfectly lossless like this, but the mechanism — eigenvectors of a covariance matrix ranking directions by variance — is exactly what a real PCA-based dimensionality reduction tool does at scale.

This is, in essence, exactly what a real 2D game engine's rendering pipeline and its analytics tooling both look like under the hood — every step traceable to a specific chapter of this course, none of it abstract math floating free of the actual code.

What This Course Doesn't Cover

In the interest of an honest accounting: **calculus-based optimization** (how gradient descent's update rule is actually derived), **formal vector space theory** (axiomatic proofs over abstract fields), and **numerical stability at scale** were all named in Chapter 1 as deliberately out of scope, each reserved for its own future course under this same Maths for Programmers subject — Calculus & Optimization, and Numerical Methods & Floating-Point Computation respectively. This course is the direct, concrete foundation those subjects will each build on, not a substitute for studying them when their own turn comes.

This Course's Throughline, Restated

A SMALL, CONSISTENT TOOLKIT REUSED EVERYWHERE

Every chapter in this course answered a version of the same question: how do you represent and manipulate multi-dimensional data with a small, consistent set of operations? The capstone above used exactly nine ideas — vectors, the cross product, matrices, transformations, linear systems, the determinant and inverse, span and independence, and eigenvalues — and nothing else, across two genuinely different problems (a rendering pipeline and a data-analysis task). That's the real payoff: the same handful of tools, reused, rather than a different specialized technique needed for every new kind of problem.

Where This Course Connects

Reusing Chapter 1's own five connections: this course is the direct foundation under Machine Learning Fundamentals and Neural Networks & Deep Learning's use of vectors and matrices, Blender Fundamentals and Vector Graphics/Figma's own transform tooling, and Data Science Fundamentals' use of PCA for dimensionality reduction — all of which this capstone touched directly. Within this subject's own future courses, Calculus & Optimization will build directly on Chapter 5's transformations and Chapter 9's eigenvalues (gradient descent is, at its core, repeatedly applying a transformation), and Numerical Methods & Floating-Point Computation will build directly on Chapter 6's elimination and Chapter 7's inverse computations, where floating-point rounding actually starts to matter at scale.

Hands-On Exercises

EXERCISE 1

A different sprite faces `f = [0, 1]` (facing "up"). The player is at direction `d = [-2, 2]` relative to the sprite. Using this chapter's own Steps 1–2 technique, compute the dot product to determine roughly how far off "ahead" the player is (find the angle), and the 2D cross product to determine whether the AI should turn left or right.

View solution

EXERCISE 2

A second calibration check uses two new points: world `3 → pixel 160`, world `6 → pixel 310`. Using this chapter's own Step 5 technique (Gaussian elimination on `pixel = m·world + c`), solve for `m` and `c`, and verify your answer against both points. Do these calibration constants match Step 5's own values, or is this a genuinely different mapping?

View solution

EXERCISE 3

For each of the eight steps in this chapter's own worked project, name the specific linear algebra topic it relied on, without looking back at the step labels — just from the description of what each step actually does.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** vectors/dot product (Ch.2) → 2D cross product (Ch.3) → matrix multiplication (Ch.4) → homogeneous transforms (Ch.5) → calibration system (Ch.6) → determinant/inverse unprojection (Ch.7) → input-scheme independence (Ch.8) → PCA on click data (Ch.9)
- Out of scope: calculus-based optimization, formal vector space theory, and numerical stability at scale — each reserved for its own future course
- This course's throughline: a small, consistent toolkit (vectors, matrices, transformations, systems, determinants, eigenvalues) reused across genuinely different problems, not a new technique per problem
- This course is the direct foundation this subject's own future courses will each build on — Calculus & Optimization on transformations/eigenvalues, Numerical Methods on elimination/inverses
- **Course complete** — Linear Algebra Fundamentals, 10 chapters, from vectors to eigenvalues