Calculus & Optimization

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

Limits & derivatives · common function derivatives & the sigmoid

Partial derivatives & the gradient · gradient descent

Convexity & optimization landscapes · the chain rule & backpropagation

Integrals & numerical integration

Capstone: fitting a real model to data via gradient descent, from scratch

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 Calculus & Optimization Matters for Programmers

Calculus & Optimization

Chapter 1 · Why Calculus & Optimization Matters for Programmers

Linear Algebra Fundamentals built vectors. Algorithms & Complexity built the idea of a repeated, improving loop. This course fuses them: a **gradient** is a vector (Chapter 5), and **gradient descent** (Chapter 6) is exactly the kind of repeated-improvement loop Algorithms & Complexity already covered — applied to a genuinely enormous real payoff: it's the literal algorithm that trains every machine learning model in production use today.

What a Derivative Actually Measures

A **derivative** is the instantaneous rate of change of a function — how fast its output moves as its input moves, at one exact point. Geometrically, it's the slope of the line tangent to the function's curve at that point. Practically, for a programmer, it answers one specific and enormously useful question: *if I nudge this input slightly, which direction does the output move, and how fast?*

A Real Demonstration: Derivatives Are Computable — With a Real Catch

For `f(x) = x²`, the true derivative is `f'(x) = 2x` — a fact this chapter states, not yet proves (Chapter 3 derives it properly). But a derivative can also be *approximated numerically*, with no symbolic calculus at all, using the definition itself: `(f(x+h) - f(x)) / h`, for a small step `h`.

VERIFIED DIRECTLY — CONVERGENCE TO THE TRUE VALUE

At `x=3`, the true derivative is `6`. Numerical approximation as `h` shrinks: `h=1 → 7.0`, `h=0.1 → 6.1`, `h=0.01 → 6.01`, `h=0.0001 → 6.0001` — the error shrinks by roughly a factor of 10 every time `h` does, converging cleanly toward the true value of `6`.

A GENUINE, VERIFIED CATCH: SMALLER ISN'T ALWAYS BETTER

Pushing `h` even smaller doesn't keep improving things forever. At `h=1e-10` the approximation is still excellent (`error ≈ 0.0000005`) — but at `h=1e-13` the error jumps back up to `0.004`, and by `h=1e-16` the computed "derivative" is `0` — completely wrong, for a true value of `6`. Shrinking `h` past a certain point makes `f(x+h)` and `f(x)` so close together that floating-point subtraction loses almost all its precision. "Just make `h` smaller" is not a free lunch — a real, verified subtlety this course won't dodge.

Five Concrete Connections to Real Code

CALCULUS TOPIC	WHERE IT ACTUALLY SHOWS UP
Gradient descent (Ch.6)	The literal training algorithm behind linear regression, logistic regression, and every deep neural network
The chain rule / backpropagation (Ch.8)	How a neural network actually learns — computing how every internal weight should change
Numerical integration (Ch.9)	Physics engines — simulating position and velocity over time from acceleration
Derivatives & interpolation (Ch.4-5)	Smooth animation easing curves and shading gradients in graphics
Optimization generally (Ch.6-7)	Any "minimize this cost" or "maximize this score" problem — hyperparameter tuning, resource allocation, curve fitting

What This Course Won't Cover

Calculus as a full field is enormous, and this course deliberately covers only what's needed to understand and implement real optimization:

- **Full real-analysis rigor** — formal epsilon-delta limit proofs and the deeper theory behind why calculus works stay out of scope; this course treats limits and derivatives practically, not axiomatically
- **Multivariable calculus beyond gradients** — Hessians, Jacobians, vector calculus (divergence, curl) are genuinely useful in specialized contexts but aren't needed for gradient descent itself, this course's own central destination
- **Differential equations** — beyond the basic numerical integration (Euler's method) covered in Chapter 9, solving differential equations analytically is its own substantial topic

> **WHY DRAW THE LINE AT GRADIENTS AND OPTIMIZATION SPECIFICALLY**
>
> Everything from Chapter 2 through Chapter 10 is chosen because it's a direct prerequisite for understanding gradient descent and backpropagation — the two ideas that, between them, explain how the vast majority of modern machine learning actually works under the hood.

Where This Course Is Headed

CHAPTER	TOPIC
2	Limits & Continuity
3	Derivatives: Definition & Rules

CHAPTER	TOPIC
4	Derivatives in Practice: Common Functions & Real Interpretation
5	Partial Derivatives & the Gradient
6	Gradient Descent: The Optimization Algorithm Behind Machine Learning
7	Convexity, Local vs. Global Minima & Optimization Landscapes
8	The Chain Rule & Backpropagation
9	Integrals & Numerical Integration
10	Capstone — Optimizing a Function From Scratch

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: given a function whose behavior you can measure, how do you systematically find where it's smallest (or largest)? Derivatives measure direction and steepness; gradients extend that to many variables at once; gradient descent turns that measurement into a repeatable search procedure; and the chain rule makes that procedure work even through deeply nested, composed functions — exactly the shape of a real neural network.

Hands-On Exercises

EXERCISE 1

For `f(x) = x³`, the true derivative at `x=2` is `f'(x)=3x²=12`. Using this chapter's own numerical approximation formula `(f(x+h)-f(x))/h`, compute the approximation for `h=0.1` and `h=0.001`, and confirm the error shrinks as `h` gets smaller.

View solution

EXERCISE 2

A colleague says "just use the smallest possible `h` your language allows, to get the most accurate numerical derivative." Using this chapter's own verified finding, explain specifically why this advice is wrong, and what actually happens to the approximation's accuracy as `h` keeps shrinking past a certain point.

View solution

EXERCISE 3

Using this chapter's own five connections, explain in your own words why "gradient descent" and "how a neural network learns" are not actually two separate topics, but the same underlying idea applied at two different chapters of this course (Chapter 6 and Chapter 8).

View solution

CHAPTER 1 QUICK REFERENCE

- A **derivative** measures instantaneous rate of change — the slope of the tangent line at a point
- Numerical differentiation `(f(x+h)-f(x))/h` converges to the true derivative as `h` shrinks — verified directly
- **But not indefinitely** — verified directly that shrinking `h` too far causes floating-point cancellation error, producing a completely wrong result
- Five direct connections: gradient descent, backpropagation, numerical integration, animation/shading curves, general optimization
- Deliberately out of scope: full real-analysis rigor, multivariable calculus beyond gradients, differential equations
- **Next chapter:** Limits and continuity

CHAPTER 2 OF 10

Limits & Continuity

Calculus & Optimization

Chapter 2 · Limits & Continuity

Chapter 1's numerical experiment shrank `h` toward zero and watched the approximation "converge toward" the true derivative — without ever setting `h` exactly to zero. That word, *converge*, has a precise name: a **limit**. This chapter formalizes it, practically rather than with full epsilon-delta rigor, per Chapter 1's own honest scope note.

What a Limit Actually Captures

THE INTUITIVE DEFINITION

`lim(x→a) f(x) = L` means: as `x` gets arbitrarily close to `a` — without necessarily ever reaching it — `f(x)` gets arbitrarily close to `L`. The limit describes what a function *approaches*, which is a genuinely different question from what the function's value *is* at that exact point — sometimes those two things differ entirely.

Why This Matters for Derivatives Specifically

The derivative is *defined* as a limit: `f'(x) = lim(h→0) (f(x+h)−f(x))/h`. Plugging in `h=0` directly gives `0/0` — genuinely undefined, a real division by zero. The limit is precisely the tool that lets Chapter 1's whole numerical experiment make sense: reasoning about what the expression approaches as `h` shrinks toward zero, without ever actually dividing by zero.

A Worked Example: A Limit That Exists Where the Function Doesn't

Consider `f(x) = (x²−4)/(x−2)`. Substituting `x=2` directly gives `0/0` — undefined, a genuine division-by-zero error.

RESOLVED ALGEBRAICALLY, VERIFIED NUMERICALLY FROM BOTH SIDES

Factoring: `(x²−4)/(x−2) = (x−2)(x+2)/(x−2) = x+2`, valid everywhere `x≠2`. Approaching from below: `x=1.9→3.9`, `x=1.99→3.99`, `x=1.999→3.999`. Approaching from above: `x=2.001→4.001`, `x=2.01→4.01`, `x=2.1→4.1`. Both directions converge on **4** — confirming `lim(x→2) f(x) = 4`, even though `f(2)` itself is a genuine `ZeroDivisionError`, confirmed directly.

Continuity: When the Limit and the Function Actually Agree

THE DEFINITION

`f` is continuous at `a` when three things all hold: `f(a)` is defined, `lim(x→a) f(x)` exists, and the two are equal.

THIS CHAPTER'S OWN EXAMPLE IS DISCONTINUOUS — A "REMOVABLE" HOLE

`f(x)=(x²−4)/(x−2)` is **discontinuous** at `x=2` — the limit exists (`4` , verified above), but `f(2)` itself is undefined, so the first requirement of continuity fails outright. The simplified function `g(x)=x+2` is continuous everywhere, including at `x=2` where it equals `4` — the two functions agree everywhere *except* at the single point the original one has a hole.

A Real Case: Continuous, But Not Differentiable

Continuity is *necessary* for differentiability, but not *sufficient* — a function can be perfectly continuous and still have no well-defined derivative at a point. `f(x) = |x|` at `x=0` is the classic case.

VERIFIED DIRECTLY — CONTINUITY HOLDS

Approaching `x=0` from both sides: `f(-0.01)=0.01` , `f(-0.001)=0.001` , `f(0.001)=0.001` , `f(0.01)=0.01` — the limit is `0` , matching `f(0)=0` exactly. `|x|` is genuinely continuous at `0` .

VERIFIED DIRECTLY — DIFFERENTIABILITY FAILS

The difference quotient `(f(0+h)−f(0))/h` , approached from the right (`h=0.1, 0.01, 0.001, 0.0001`): consistently **1.0**. Approached from the left (`h=−0.1, −0.01, −0.001, −0.0001`): consistently **−1.0**. The two one-sided limits genuinely disagree — no single limit exists, so `|x|` has **no derivative** at `x=0` , despite being perfectly continuous there. A sharp corner, not a smooth curve, at that exact point.

Real Relevance

Chapter 6's gradient descent assumes it can always ask "which direction does this function decrease in?" — a question that only has a clean answer where the function is differentiable. Functions with sharp corners (like ReLU, an extremely common neural-network activation function, which is literally `max(0,x)` — a close cousin of `|x|`) genuinely have points where the derivative doesn't exist in the classical sense, a real practical wrinkle real ML libraries handle by defining a sensible value at that single point rather than pretending the problem doesn't exist.

Limits in Code

```python
def f(x):
    return (x**2 - 4) / (x - 2)

# approaching x=2 from both sides -- never substituting x=2 directly
for x in [1.9, 1.99, 1.999, 2.001, 2.01, 2.1]:
    print(f"f({x}) = {f(x)}")   # all approach 4

try:
    f(2)   # ZeroDivisionError -- f(2) is genuinely undefined
except ZeroDivisionError:
    print("f(2) is undefined, even though the limit at x=2 is 4")
```

Hands-On Exercises

EXERCISE 1

Find `lim(x→3) (x²−9)/(x−3)` algebraically (factor first), then verify your answer numerically by evaluating the function at `x=2.99`, `x=2.999`, `x=3.001`, and `x=3.01`.

View solution

EXERCISE 2

Using this chapter's own three-part definition of continuity, explain specifically which part fails for `f(x)=(x²−9)/(x−3)` at `x=3`, given your answer to Exercise 1.

View solution

EXERCISE 3

Using this chapter's own left/right difference-quotient method for `|x|`, compute the one-sided difference quotients for `f(x) = -|x|` at `x=0` (approaching from the right with `h=0.01`, and from the left with `h=-0.01`). Do they agree? What does this tell you about whether `-|x|` is differentiable at `x=0`?

View solution

CHAPTER 2 QUICK REFERENCE

- **Limit:** what a function approaches as its input approaches a point — not necessarily what the function equals there
- The derivative is defined as a limit specifically to avoid ever dividing by zero directly

- An indeterminate `0/0` form can often be resolved by algebraic simplification first — verified on `(x²−4)/(x−2) → 4` at `x=2`
- **Continuity:** `f(a)` defined, the limit exists, and they're equal — all three, together
- Continuity is necessary but not sufficient for differentiability — verified directly: `|x|` is continuous but not differentiable at `x=0` (left/right difference quotients disagree, `-1` vs. `1`)
- **Next chapter:** Derivatives — definition and rules

CHAPTER 3 OF 10

Derivatives: Definition & Rules

Calculus & Optimization

Chapter 3 · Derivatives: Definition & Rules

Chapter 2 formalized the derivative as a limit: `f'(x) = lim(h→0) (f(x+h)−f(x))/h`. Recomputing that limit from scratch for every function would be unworkable. This chapter derives — not just states — the mechanical rules that make derivatives fast to compute, ending with the one rule this entire course is really building toward.

The Power Rule, Derived From the Limit Definition

For `f(x) = x²`, applying Chapter 2's own limit definition directly:

DERIVED, NOT ASSUMED

`f'(x) = lim(h→0) ((x+h)² − x²)/h = lim(h→0) (x²+2xh+h² − x²)/h = lim(h→0) (2xh+h²)/h = lim(h→0) (2x+h) = 2x`. The same expansion for `f(x)=x³` gives `lim(h→0) (3x²h + 3xh² + h³)/h = lim(h→0)(3x² + 3xh + h²) = 3x²`. Both match the general **power rule**: `d/dx[xⁿ] = n·xⁿ⁻¹`.

VERIFIED DIRECTLY AGAINST NUMERICAL DIFFERENTIATION, FOR N=2 THROUGH 5

At `x=3`: `n=2 → 6`, `n=3 → 27`, `n=4 → 108`, `n=5 → 405` — each matching the power rule's own prediction (`n·3ⁿ⁻¹`) and a numerical central-difference approximation, to within `0.0001`.

The Product Rule

THE RULE

`d/dx[f(x)·g(x)] = f'(x)g(x) + f(x)g'(x)`

VERIFIED DIRECTLY — A CROSS-CHECK AGAINST THE POWER RULE

`f(x)=x²`, `g(x)=x³` — their product is just `x⁵`, so the power rule alone already predicts the derivative directly: `5x⁴`. At `x=2`: product rule gives `f'(2)g(2)+f(2)g'(2) = 4×8 + 4×12 = 32+48 = 80`. Direct power rule on `x⁵`: `5×2⁴=80`. Numerical differentiation: `80.000000`. All three agree exactly.

The Quotient Rule

THE RULE

`d/dx[f(x)/g(x)] = (f'(x)g(x) − f(x)g'(x)) / g(x)²`

VERIFIED DIRECTLY — ANOTHER CROSS-CHECK

`f(x)=x³`, `g(x)=x` — their quotient is `x²`, predicted directly by the power rule as `2x`. At `x=4`: quotient rule gives `(3×16×4 − 64×1)/16 = (192−64)/16 = 8`. Direct power rule: `2×4=8`. Numerical differentiation: `8.000000`. All three agree.

The Chain Rule — the Single Most Important Rule in This Course

Every other rule in this chapter handles functions built from simple combination (sum, product, quotient). The **chain rule** handles functions built from *composition* — one function's output feeding directly into another's input — which is exactly the shape of every layer of a neural network feeding into the next. Chapter 8's entire subject, backpropagation, is nothing more than this one rule, applied repeatedly.

THE RULE

For `h(x) = f(g(x))`: `h'(x) = f'(g(x)) · g'(x)` — the derivative of the "outer" function, evaluated at the inner function's own value, multiplied by the derivative of the "inner" function.

VERIFIED DIRECTLY

`h(x) = (x²+1)³` — an outer function `f(u)=u³` wrapped around an inner function `g(x)=x²+1`. Chain rule: `h'(x) = 3(x²+1)² · 2x`. At `x=2`: `3×5²×4 = 3×25×4 = 300`. Numerical differentiation at `x=2`: `300.000000`. Exact match.

Rules at a Glance

RULE	FORMULA
Power	`d/dx[xⁿ] = nxⁿ⁻¹`
Product	`d/dx[fg] = f'g + fg'`
Quotient	`d/dx[f/g] = (f'g − fg')/g²`
Chain	`d/dx[f(g(x))] = f'(g(x))·g'(x)`

Derivative Rules in Code

```
def numerical_deriv(f, x, h=1e-6):
    return (f(x+h) - f(x-h)) / (2*h)   # central difference

# chain rule check: h(x) = (x^2+1)^3
def h(x): return (x**2 + 1)**3
def h_chain_rule(x): return 3*(x**2+1)**2 * (2*x)

x0 = 2
print(numerical_deriv(h, x0), h_chain_rule(x0))   # 300.0 300 -- match
```

Hands-On Exercises

EXERCISE 1

Using the power rule, find the derivative of `f(x) = x⁴`. Then, using the product rule on `f(x) = x·x³` (the same function, written as a product), confirm you get the identical result at `x=2`.

View solution

EXERCISE 2

Using the chain rule, find the derivative of `h(x) = (3x+1)²`, and evaluate it at `x=1`. Then verify your answer using numerical differentiation (central difference, `h=1e-6`) at the same point.

View solution

EXERCISE 3

Explain, using this chapter's own framing, why the chain rule specifically — rather than the product or quotient rule — is the rule most directly connected to how a neural network computes its own gradients.

View solution

CHAPTER 3 QUICK REFERENCE

- **Power rule:** `d/dx[xⁿ] = nxⁿ⁻¹` — derived directly from Chapter 2's own limit definition, not just stated
- **Product rule:** `f'g + fg'`. **Quotient rule:** `(f'g − fg')/g²` — both cross-verified against direct power-rule results and numerical differentiation

- **Chain rule:** `f'(g(x))·g'(x)` — for composed functions, verified numerically to match exactly
- The chain rule is the single most important rule in this course — Chapter 8's backpropagation is this rule, applied repeatedly through a computational graph
- **Next chapter:** Derivatives in practice — common functions and real interpretation

CHAPTER 4 OF 10

Derivatives in Practice: Common Functions & Real Interpretation

Calculus & Optimization

Chapter 4 · Derivatives in Practice: Common Functions & Real Interpretation

Chapter 3 built the rules. This chapter applies them to the specific functions that show up constantly in real code — exponentials, logarithms, and trig functions — and formalizes the numerical differentiation technique this course has been using informally since Chapter 1 into a proper, comparable tool.

Common Function Derivatives, Verified

FUNCTION	DERIVATIVE
e^x	`e^x` — its own derivative
ln(x)	`1/x`
sin(x)	`cos(x)`
cos(x)	`-sin(x)`

VERIFIED DIRECTLY, ALL FOUR

At `x=1.5`: `e^1.5=4.481689`, numerical derivative `=4.481689` — identical, confirming `e^x` genuinely is its own derivative. At `x=2`: `ln'(x)` numerically `=0.5`, matching `1/x=0.5` exactly. At `x=π/4`: `sin'(x)` numerically `=0.707107`, matching `cos(π/4)=0.707107`; `cos'(x)` numerically `=-0.707107`, matching `-sin(π/4)=-0.707107`.

Real Relevance: The Sigmoid Function's Own Derivative

The **sigmoid** function, `σ(x) = 1/(1+e⁻ˣ)`, is one of the most common activation functions in machine learning — built entirely from `e^x`. Applying the chain and quotient rules (Chapter 3) gives it an unusually elegant derivative:

VERIFIED DIRECTLY

`σ'(x) = σ(x)·(1−σ(x))` — the derivative is expressible entirely in terms of the function's own output, needing no re-evaluation of `e^x` at all. At `x=0.5`: formula gives `0.235004`, numerical differentiation gives `0.235004` — exact

match. This clean form is precisely why sigmoid was historically such a popular activation function: computing its gradient during training is nearly free once the forward pass has already computed `σ(x)`.

What the Sign and Magnitude of a Derivative Actually Mean

DERIVATIVE VALUE	WHAT IT MEANS, GEOMETRICALLY
f'(x) > 0	Function is increasing at that point
f'(x) < 0	Function is decreasing
f'(x) = 0	A critical point — a local max, local min, or saddle (Chapter 7 sorts out which)
|f'(x)| large	Steep — the function is changing rapidly
|f'(x)| small	Nearly flat — the function is barely changing

Numerical Differentiation, Formalized: Forward vs. Central Difference

Every prior chapter used the difference quotient informally. There are actually two natural versions: **forward difference**, `(f(x+h)-f(x))/h`, and **central difference**, `(f(x+h)-f(x-h))/(2h)` — averaging a step forward and a step back.

VERIFIED DIRECTLY — CENTRAL DIFFERENCE IS DRAMATICALLY MORE ACCURATE FOR THE SAME H

For `f(x)=x³` at `x=3` (true derivative `27`): at `h=0.1`, forward difference error is `0.91`, central difference error is only `0.01` — **91× more accurate**, using the identical step size. At `h=0.01`: forward error `0.09`, central error `0.0001` — roughly **900× more accurate**. Central difference's error shrinks roughly with the *square* of `h`, while forward difference's error shrinks only linearly with `h`.

WHY THIS MATTERS IN PRACTICE

Chapter 1's own numerical experiment used a forward-style difference. Central difference reaches the same accuracy with a far larger, floating-point-safer `h` — directly softening the exact catastrophic-cancellation problem Chapter 1 flagged, without needing `h` to shrink nearly as far.

Common Derivatives in Code

```python
import math

def forward_diff(f, x, h): return (f(x+h) - f(x)) / h
def central_diff(f, x, h): return (f(x+h) - f(x-h)) / (2*h)

def sigmoid(x): return 1 / (1 + math.exp(-x))
def sigmoid_deriv(x):
    s = sigmoid(x)
    return s * (1 - s)   # the clean closed form -- no re-computing e^x

x0 = 0.5
print(sigmoid_deriv(x0), central_diff(sigmoid, x0, 1e-6))   # match
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own derivative table, find the derivative of `f(x) = ln(x)` at `x=5`, then verify it numerically using central difference with `h=1e-6`.

View solution

EXERCISE 2

Compute the sigmoid function's own derivative at `x=0` using this chapter's own closed-form formula `σ(x)(1−σ(x))`, given that `σ(0)=0.5` exactly. Explain, using this chapter's own sign/magnitude table, what a derivative of this specific value means about how steeply the sigmoid function is changing at `x=0` compared to at very large or very negative `x` (where `σ(x)` approaches 0 or 1).

View solution

EXERCISE 3

For `f(x)=x²` at `x=1` (true derivative `2`), compute the forward difference and central difference approximations at `h=0.5`, and compare their errors. Does this example show the same "central difference is dramatically more accurate" pattern this chapter found for `x³`? If not, explain why not, referring to what's special about `x²` specifically.

View solution

CHAPTER 4 QUICK REFERENCE

- **e^x** is its own derivative. **ln(x)' = 1/x**. **sin(x)' = cos(x)**, **cos(x)' = −sin(x)** — all verified numerically

- **Sigmoid's derivative:** `σ(x)(1−σ(x))` — an elegant closed form, directly why sigmoid was historically popular in ML
- Sign of `f'(x)`: increasing/decreasing. Magnitude: how steeply. Zero: a critical point (Chapter 7 sorts out which kind)
- **Central difference is dramatically more accurate than forward difference** for the same step size — verified directly (91× to 900× more accurate)
- Central difference's error shrinks with `h²`; forward difference's error shrinks only with `h`
- **Next chapter:** Partial derivatives and the gradient

CHAPTER 5 OF 10

Partial Derivatives & the Gradient

Calculus & Optimization

Chapter 5 · Partial Derivatives & the Gradient

Every derivative so far has tracked one variable. A real loss function has as many variables as a model has parameters — sometimes millions. This chapter extends derivatives to that world, and lands on the exact object Chapter 6 needs: a vector, built directly from Linear Algebra Fundamentals' own material, that tells gradient descent which way to move.

Partial Derivatives: One Variable at a Time

THE IDEA

A **partial derivative** `∂f/∂x` is the derivative of a multivariable function with respect to one variable, treating every other variable as a constant — literally reusing every rule from Chapter 3, just applied to one variable while the rest sit still.

For `f(x,y) = x²y + y³`:

VERIFIED DIRECTLY AGAINST NUMERICAL DIFFERENTIATION, AT (2,3)

`∂f/∂x = 2xy` (treating `y` as constant, applying the power rule to the `x²` part). At `(2,3)`: formula gives `2×2×3=12`; numerically holding `y` fixed and varying `x` gives `12.000000` — exact match. `∂f/∂y = x²+3y²` (treating `x` as constant). At `(2,3)`: formula gives `4+27=31`; numerically holding `x` fixed gives `31.000000` — exact match.

The Gradient: A Vector of Partial Derivatives

THE DEFINITION

The **gradient**, `∇f`, is the vector formed by stacking every partial derivative together: `∇f = (∂f/∂x, ∂f/∂y, ...)`. This is exactly the kind of object Linear Algebra Fundamentals already built machinery for — magnitude, direction, dot products all apply directly.

A Full, Verified Demonstration: The Gradient Points in the Direction of Steepest Ascent

For `f(x,y) = x²+y²` at the point `(1,2)`: `f(1,2)=5`, and the gradient is `∇f=(2x,2y)=(2,4)`.

VERIFIED DIRECTLY — TESTING SIX DIRECTIONS, SAME STEP SIZE

Taking a step of `0.1` in the *unit gradient direction* `(0.447, 0.894)`: `f` increases by **0.457** — the largest increase of every direction tested. The *perpendicular* direction: increase of only `0.010` — almost flat. The *negative gradient* direction: `f` **decreases** by `0.437` — the steepest possible decrease. Two arbitrary directions, `(1,0)` and `(0,1)`: increases of `0.210` and `0.410` — both real increases, but neither beats the gradient direction's own `0.457`.

Why: the Directional Derivative Is a Dot Product

The rate of change of `f` in *any* unit direction `u` is `∇f · u` — a genuine dot product, straight from Linear Algebra Fundamentals. Since a dot product is maximized exactly when two vectors point the same way, the direction that maximizes `∇f · u` is `u = ∇f` itself — which is *why* the gradient is the direction of steepest ascent, not merely an observed pattern.

VERIFIED DIRECTLY — THE DOT PRODUCT RANKS EVERY DIRECTION CORRECTLY

`∇f·u` for each of the six directions tested above: gradient direction `4.472` (the largest), `(0,1)` direction `4.0`, `(1,0)` direction `2.0`, perpendicular direction `0.0` (exactly zero rate of change), negative gradient direction `-4.472` (the most negative). The ranking matches the actual measured `f`-increases from the six-direction test exactly.

Real Relevance

This is the entire reason Chapter 6's gradient descent works at all: at any point, the negative gradient — `-∇f` — points in the direction of steepest *decrease*, exactly the direction needed to minimize a loss function. Every parameter update in every gradient-descent-trained model is, at its core, exactly the demonstration verified above, just run in reverse and repeated many times.

Partial Derivatives & Gradients in Code

```python
def f(x, y): return x**2 + y**2

def gradient(f, x, y, h=1e-6):
    dfdx = (f(x+h, y) - f(x-h, y)) / (2*h)
    dfdy = (f(x, y+h) - f(x, y-h)) / (2*h)
    return (dfdx, dfdy)

import math
gx, gy = gradient(f, 1, 2)
mag = math.sqrt(gx**2 + gy**2)
unit = (gx/mag, gy/mag)
print(gx, gy, unit)   # (2.0, 4.0), (0.447, 0.894)
```

Hands-On Exercises

EXERCISE 1

For `f(x,y) = 3x²y − y³`, find `∂f/∂x` and `∂f/∂y` symbolically, then evaluate both at `(1,2)`.

View solution

EXERCISE 2

Using your answer to Exercise 1, write out the gradient `∇f` at `(1,2)` as a vector, then compute its magnitude.

View solution

EXERCISE 3

Using this chapter's own directional-derivative dot-product formula, explain why the directional derivative in the direction *exactly perpendicular* to the gradient is always precisely zero, for any function and any point — not just the specific example this chapter tested.

View solution

CHAPTER 5 QUICK REFERENCE

- **Partial derivative:** derivative with respect to one variable, holding the rest constant — reuses every Chapter 3 rule directly
- **Gradient:** the vector of every partial derivative, `∇f = (∂f/∂x, ∂f/∂y, ...)`
- Verified directly: stepping in the gradient's own direction gives strictly the largest increase, among six directions tested

- **Directional derivative = ∇f · u** — a dot product, maximized exactly when `u` aligns with the gradient (Linear Algebra Fundamentals)
- The negative gradient points toward steepest decrease — the exact mechanism Chapter 6's gradient descent is built on
- **Next chapter:** Gradient descent — the optimization algorithm behind machine learning

CHAPTER 6 OF 10

Gradient Descent: The Optimization Algorithm Behind Machine Learning

Calculus & Optimization

Chapter 6 · Gradient Descent: The Optimization Algorithm Behind Machine Learning

Chapter 5 proved the negative gradient points toward steepest decrease. This chapter turns that single geometric fact into a repeatable algorithm — the literal procedure that trains linear regression, logistic regression, and every deep neural network in production use.

The Algorithm

GRADIENT DESCENT, IN FULL

Start at some point. Repeat: compute the gradient there; move a small step in the *negative* gradient direction — `x_new = x_old − α·∇f(x_old)`, where `α` (the **learning rate**) controls the step size. Stop when the gradient is close enough to zero, or after a fixed number of steps.

A Fully Traced, Verified Minimization

Minimizing `f(x,y) = (x−3)² + (y+1)² + 5` — true minimum at `(3,−1)`, value `5` — starting from `(0,0)`, learning rate `α=0.1`:

VERIFIED DIRECTLY

`iter 0: (0,0), f=15.0` → `iter 1: (0.6,−0.2), f=11.4` → `iter 2: (1.08,−0.36), f=9.096` → `iter 5: (2.017,−0.672), f=6.074` → `iter 9: (2.597,−0.866), f=5.180` — steadily, monotonically approaching the true minimum. After **50 iterations**: `(2.999957, −0.999986)`, `f=5.00000000` — converged.

The Learning Rate: Five Regimes, All Verified

The same starting point, the same function, five different learning rates — the outcome changes completely.

LEARNING RATE	WHAT HAPPENS
α=0.1 (above)	Smooth, monotonic convergence to the minimum

LEARNING RATE	WHAT HAPPENS
α=0.5	Converges to the exact minimum in a **single step**
α=1.0	Oscillates forever between two points, never converging or diverging
α=1.1	Genuinely diverges — cost grows every iteration
α=1.5	Diverges explosively — cost roughly quadruples every iteration

VERIFIED DIRECTLY — A=0.5, THE SINGLE-STEP-OPTIMAL RATE FOR THIS FUNCTION

Starting at `(0,0)`: after exactly **one** step, `(x,y) = (3.0, -1.0)`, `f=5.0` — the exact true minimum, reached immediately, and stable there forever after.

VERIFIED DIRECTLY — A=1.0, THE EXACT OSCILLATION BOUNDARY

`(0,0) → (6,-2) → (0,0) → (6,-2) → (0,0) → (6,-2)`, forever — `f` stuck at exactly **15.0** on every single iteration, neither improving nor getting worse.

VERIFIED DIRECTLY — A=1.1 AND A=1.5, GENUINE DIVERGENCE

At `α=1.1`: `f` climbs `15.0 → 19.4 → 25.7 → 34.9 → 48.0 → 66.9 → 94.2 → 133.4`. At `α=1.5`: `f` climbs `15.0 → 45.0 → 165.0 → 645.0 → 2565.0 → 10245.0` — roughly quadrupling every step, a genuine runaway.

Why: the Exact Convergence Threshold

For this specific function, each gradient descent step is a simple linear update: `x_new = x·(1−2α) + 6α`. Whether this converges depends entirely on `|1−2α|`.

VERIFIED DIRECTLY, EXACTLY MATCHING EVERY REGIME OBSERVED ABOVE

`α=0.1: |1−0.2|=0.8 < 1` — converges. `α=0.5: |1−1|=0 exactly` — the fastest possible convergence, a single step. `α=1.0: |1−2|=1 exactly` — the precise oscillation boundary. `α=1.1: |1−2.2|=1.2 > 1` — diverges. `α=1.5: |1−3|=2 > 1` — diverges roughly twice as fast, matching the observed near-quadrupling (each step's error multiplies by `2`, and cost grows with the *square* of the distance from the minimum).

WHY THIS MATTERS BEYOND THIS ONE EXAMPLE

Every real function has its own version of this threshold, tied to how sharply it curves — this is exactly why choosing a learning rate is a genuinely important, non-trivial part of training any real model, not an arbitrary setting.

Gradient Descent in Code

```
def f(x, y): return (x-3)**2 + (y+1)**2 + 5
def gradient(x, y): return (2*(x-3), 2*(y+1))

def gradient_descent(x, y, lr, iterations):
    for _ in range(iterations):
        gx, gy = gradient(x, y)
        x, y = x - lr*gx, y - lr*gy   # step in the NEGATIVE gradient direction
    return x, y

print(gradient_descent(0, 0, 0.1, 50))   # (2.999957, -0.999986) -- converged
print(gradient_descent(0, 0, 0.5, 1))    # (3.0, -1.0) -- exact, one step
```

Hands-On Exercises

EXERCISE 1

For `f(x) = (x−5)²`, run one step of gradient descent starting at `x=0` with learning rate `α=0.2`, showing the gradient computed and the resulting new `x` value.

View solution

EXERCISE 2

Using this chapter's own convergence-threshold reasoning (`|1−2α|`) applied to `f(x)=(x−5)²`, what learning rate would reach the exact minimum in a single step, starting from any point? Verify your answer by running one step of gradient descent from `x=0` with that learning rate.

View solution

EXERCISE 3

A colleague sets a learning rate of `α=1.05` for training a model and says "it's close enough to 1, it should be fine." Using this chapter's own verified findings about `α=1.0` and `α=1.1`, explain specifically why this reasoning is dangerous, even though `1.05` genuinely is close to `1.0`.

View solution

CHAPTER 6 QUICK REFERENCE

- **Gradient descent:** `x_new = x_old − α·∇f(x_old)`, repeated until the gradient is near zero
- Verified: 50 iterations at `α=0.1` converge to within `0.00005` of the true minimum

- Learning rate genuinely changes the outcome: too small is just slow, but too large causes real, verified divergence — not merely inefficiency
- An exact convergence threshold exists per function (`|1-2α|<1` for this chapter's own quadratic) — `α=1.0` is the precise boundary, verified to oscillate forever rather than converge or diverge
- A well-chosen learning rate can converge in a single step for a simple quadratic — verified directly at `α=0.5`
- **Next chapter:** Convexity, local vs. global minima, and optimization landscapes

CHAPTER 7 OF 10

Convexity, Local vs. Global Minima & Optimization Landscapes

Calculus & Optimization

Chapter 7 · Convexity, Local vs. Global Minima & Optimization Landscapes

Chapter 6's function had exactly one minimum, and gradient descent found it reliably from any starting point, at any working learning rate. That was a special property of the function, not a general guarantee. This chapter asks the honest question: what happens when a function has more than one minimum?

Convexity: The Property That Makes Chapter 6's Guarantee Work

THE SECOND-DERIVATIVE TEST

A function is **convex** where its second derivative `f''(x) ≥ 0`. Geometrically: the function curves upward like a bowl, never a wiggle. A function that's convex *everywhere* has at most one local minimum — and that minimum is automatically the **global** minimum. Gradient descent on a convex function is guaranteed to find it, from any starting point, with any small-enough learning rate.

VERIFIED DIRECTLY

Chapter 6's own function, `f(x,y)=(x−3)²+(y+1)²+5`, has second derivative `2` in each variable — constant, always positive, convex everywhere. This is *why* every learning rate below the threshold converged to the exact same point regardless of starting location.

A Non-Convex Function: Verified Multiple Minima

Consider `f(x) = x⁴/4 − x³ − 2x² + 3x`. Its derivative has three roots — three critical points.

VERIFIED DIRECTLY — THREE CRITICAL POINTS, CHECKED WITH THE SECOND-DERIVATIVE TEST

`x≈−1.398`: `f≈−4.416`, `f''≈10.25>0` — a **local minimum**. `x≈0.559`: `f''≈−6.42<0` — a local maximum, sitting between the two minima. `x≈3.838`: `f≈−20.236`, `f''≈17.17>0` — a **second local minimum**, and by direct comparison, the **global** one — nearly five times lower than the first.

Gradient Descent Genuinely Gets Stuck

VERIFIED DIRECTLY — TWO STARTING POINTS, TWO COMPLETELY DIFFERENT OUTCOMES

Starting at `x=-2` (in the shallow-minimum's own basin): gradient descent converges to `x=-1.398`, `f=-4.416`. Starting at `x=5` (in the deep-minimum's own basin): converges to `x=3.838`, `f=-20.236` — a dramatically better result, found purely because of where the search happened to begin.

VERIFIED DIRECTLY — AN EVEN SHARPER DEMONSTRATION: TWO STARTS ONE UNIT APART

Starting at `x=0` (just left of the local max at `0.559`): converges to the **shallow** minimum, `f=-4.416`. Starting at `x=1` — barely one unit away, on the other side of that same local max: converges to the **deep, global** minimum, `f=-20.236`. A tiny difference in starting point produced a nearly 5× difference in the final result.

Real Relevance: Why Neural Network Training Is Empirically Finicky

This is exactly why training a real neural network is genuinely sensitive to weight initialization, and why the same architecture can train to noticeably different final results depending on random seed — the loss surface of a real neural network is almost never convex, and gradient descent (or its more sophisticated variants) can land in different basins depending on where the search starts, exactly like this chapter's own verified example, just in a space with millions of dimensions instead of one.

Convexity & Landscapes in Code

```
def f(x): return x**4/4 - x**3 - 2*x**2 + 3*x
def fprime(x): return x**3 - 3*x**2 - 4*x + 3

def gradient_descent_1d(x0, lr, iterations):
    x = x0
    for _ in range(iterations):
        x = x - lr * fprime(x)
    return x

print(gradient_descent_1d(-2, 0.01, 200))   # -1.398 -- stuck in the shallow minimum
print(gradient_descent_1d(5, 0.01, 200))    # 3.838 -- found the deep global minimum
```

Hands-On Exercises

EXERCISE 1

Using the second-derivative test, determine whether `f(x) = x⁴` is convex everywhere. Compute `f''(x)` and check its sign at `x=-2, 0, 2`.

View solution

EXERCISE 2

Using this chapter's own verified critical points for `f(x) = x⁴/4 − x³ − 2x² + 3x`, explain why a starting point placed exactly at `x=0.559` (the local maximum itself) is a genuinely unstable, unreliable place for gradient descent to begin — referring to what the gradient's own value is there.

View solution

EXERCISE 3

A colleague argues "since gradient descent got stuck in a worse local minimum starting from x=−2, gradient descent is a broken algorithm and shouldn't be trusted." Using this chapter's own distinction between convex and non-convex functions, explain what's wrong with this conclusion.

View solution

CHAPTER 7 QUICK REFERENCE

- **Convex:** `f''(x)≥0` everywhere — at most one local minimum, which is automatically global; gradient descent is guaranteed to find it
- Verified: a non-convex quartic has two genuinely different local minima (`f=-4.416` and `f=-20.236`) separated by a local maximum
- Verified: gradient descent from `x=-2` gets stuck in the shallow minimum; from `x=5` it finds the deep global one
- Verified: starting points just one unit apart (`x=0` vs. `x=1`) land in completely different basins — a nearly 5× difference in outcome
- This is exactly why real neural network training is sensitive to initialization and random seed — loss surfaces are almost never convex
- **Next chapter:** The chain rule and backpropagation

CHAPTER 8 OF 10

The Chain Rule & Backpropagation

Calculus & Optimization

Chapter 8 · The Chain Rule & Backpropagation

Chapter 3 promised this chapter directly: the chain rule is the single most important rule in this course because backpropagation — the algorithm that trains every neural network — is nothing more than that one rule, applied repeatedly. This chapter delivers on that promise in full, on a real, if tiny, network.

A Genuine Tiny Neural Network

THE COMPUTATIONAL GRAPH

`z1 = w1·x + b1` (a linear layer) → `a1 = σ(z1)` (Chapter 4's sigmoid activation) → `z2 = w2·a1 + b2` (a second linear layer) → `L = (z2 − y_target)²` (squared-error loss). Four learnable parameters — `w1, b1, w2, b2` — and training means finding the values that make `L` as small as possible, exactly Chapter 6's own optimization problem.

A CONCRETE FORWARD PASS, VERIFIED DIRECTLY

With `x=1.5, w1=0.5, b1=0.2, w2=−0.8, b2=0.3, y_target=1.0`: `z1=0.95`, `a1=σ(0.95)=0.7211`, `z2=−0.2769`, `L=(−0.2769−1.0)²=1.6305`.

The Backward Pass: The Chain Rule, Applied Four Times

To run gradient descent, every one of the four parameters needs its own gradient — `∂L/∂w1`, `∂L/∂b1`, `∂L/∂w2`, `∂L/∂b2`. Each is a chain of derivatives, computed backward from the loss:

EVERY LINK IN THE CHAIN, COMPUTED EXPLICITLY

`∂L/∂z2 = 2(z2−y_target) = −2.5538` (squared-error derivative, Chapter 3). `∂z2/∂a1 = w2 = −0.8`. `∂a1/∂z1 = a1(1−a1) = 0.2011` (Chapter 4's own sigmoid derivative, reused directly). `∂z1/∂w1 = x = 1.5`. Chaining: `∂L/∂w1 = ∂L/∂z2 · ∂z2/∂a1 · ∂a1/∂z1 · ∂z1/∂w1 = 0.6163`. The same chain, stopping one link earlier (`∂z1/∂b1=1` instead of `x`), gives `∂L/∂b1 = 0.4109`. The output layer needs a shorter chain: `∂L/∂w2 = ∂L/∂z2 · a1 = −1.8416`, `∂L/∂b2 = ∂L/∂z2 = −2.5538`.

WHY THIS IS EXACTLY BACKPROPAGATION

Notice `∂L/∂z2` was computed once and *reused* for every downstream gradient — the "error signal" flows backward from the loss, layer by layer, each layer's own local derivative getting multiplied in along the way. That reuse is precisely what makes backpropagation efficient rather than recomputing the whole chain from scratch for every single parameter.

Verified Against the Whole Network, End to End

EVERY GRADIENT MATCHES NUMERICAL DIFFERENTIATION EXACTLY

Nudging each parameter by `±1e-6` and re-running the *entire* forward pass, ignoring the chain-rule derivation completely: `∂L/∂w1` numerically `=0.616304`, chain rule `=0.616304`. `∂L/∂b1`: `0.410869` both ways. `∂L/∂w2`: `-1.841573` both ways. `∂L/∂b2`: `-2.553784` both ways. Every single one matches exactly — the chain rule, applied by hand through four composed functions, produces identically the same answer as brute-force perturbing the whole network.

Closing the Loop: Actually Training This Network

Feeding these exact gradients into Chapter 6's own gradient descent update, learning rate `α=0.5`:

VERIFIED DIRECTLY — THE LOSS GENUINELY FALLS

`step 0: loss=1.630454` → `step 1: 0.416999` → `step 2: 0.042482` → `step 3: 0.004199` → `step 4: 0.000381` → `step 5: 0.000035` → after 6 total updates: `loss=0.000003`. Every piece of this course — the chain rule (Ch.3), the gradient (Ch.5), gradient descent (Ch.6) — working together on a real, if miniature, trained network.

Backpropagation in Code

```
import math
def sigmoid(z): return 1/(1+math.exp(-z))

def forward(x, w1, b1, w2, b2, y):
    z1 = w1*x + b1
    a1 = sigmoid(z1)
    z2 = w2*a1 + b2
    L = (z2 - y)**2
    return z1, a1, z2, L

def backward(x, w1, b1, w2, b2, y):
    z1, a1, z2, L = forward(x, w1, b1, w2, b2, y)
    dL_dz2 = 2*(z2 - y)
    da1_dz1 = a1*(1-a1)                # Chapter 4's sigmoid derivative
    dL_dw1 = dL_dz2 * w2 * da1_dz1 * x   # chain rule, 4 links
    dL_db1 = dL_dz2 * w2 * da1_dz1 * 1
    dL_dw2 = dL_dz2 * a1                 # chain rule, 2 links
    dL_db2 = dL_dz2
    return dL_dw1, dL_db1, dL_dw2, dL_db2, L

# train for 6 steps
x, w1, b1, w2, b2, y = 1.5, 0.5, 0.2, -0.8, 0.3, 1.0
for _ in range(6):
    dw1, db1, dw2, db2, L = backward(x, w1, b1, w2, b2, y)
    print(L)
    w1, b1, w2, b2 = w1-0.5*dw1, b1-0.5*db1, w2-0.5*dw2, b2-0.5*db2
```

Hands-On Exercises

EXERCISE 1

Using this chapter's own computational graph and the values `z2=-0.2769`, `y_target=1.0`, compute `∂L/∂z2` directly, showing each part of the calculation.

View solution

EXERCISE 2

Using this chapter's own chain — `∂L/∂z2=-2.5538`, `w2=-0.8`, `a1=0.7211` — compute `∂a1/∂z1` using Chapter 4's own sigmoid derivative formula, then assemble the full chain rule product for `∂L/∂b1` (using `∂z1/∂b1=1`), showing each link multiplied in.

View solution

EXERCISE 3

Explain, using this chapter's own worked example, specifically why `∂L/∂w2` 's own chain (`∂L/∂z2 · ∂z2/∂w2`) is shorter than `∂L/∂w1` 's own chain (four links). What does this tell you about how the length of the chain rule needed for a parameter's gradient relates to that parameter's own position in the network?

View solution

CHAPTER 8 QUICK REFERENCE

- **Backpropagation is the chain rule** (Chapter 3), applied repeatedly through a network's own layers, backward from the loss
- The "error signal" (`∂L/∂z2` here) is computed once and reused for every downstream gradient — the actual source of backpropagation's efficiency
- All four gradients verified to match numerical differentiation of the entire network exactly
- Chapter 4's own sigmoid derivative, `σ(x)(1−σ(x))` , is reused directly as one link in the chain
- Verified end to end: feeding these gradients into Chapter 6's gradient descent drove a real loss from `1.63` to `0.000003` in six steps
- **Next chapter:** Integrals and numerical integration

CHAPTER 9 OF 10

Integrals & Numerical Integration

Calculus & Optimization

Chapter 9 · Integrals & Numerical Integration

Chapters 3-8 built one half of calculus: given a function, find its rate of change. This chapter builds the other half — given a rate of change, find the accumulated total — and lands on the specific technique real physics engines and simulations run every single frame.

The Integral as Accumulated Area

THE FUNDAMENTAL THEOREM OF CALCULUS

`∫ f(x) dx` from `a` to `b` is the signed area between `f`'s curve and the x-axis. If `F` is an **antiderivative** of `f` (meaning `F'(x)=f(x)`), then that area equals `F(b)−F(a)` — integration and differentiation are inverse operations.

VERIFIED DIRECTLY — THE ANTIDERIVATIVE MATCHES NUMERICAL AREA EXACTLY

`∫x² dx` from `0` to `3`: the antiderivative `F(x)=x³/3` gives `F(3)−F(0)=9`. A direct numerical approximation of the actual area (midpoint rule, `n=1000` thin rectangles) gives `8.999998` — matching to within `0.000002`.

Numerical Integration: Four Methods, Compared

Real functions in code (or genuinely unknown functions from sensor data) don't come with a symbolic antiderivative. Numerical integration approximates the area directly, by summing many small pieces.

METHOD	IDEA
Left Riemann sum	Rectangles, height from the left edge of each interval
Right Riemann sum	Rectangles, height from the right edge
Midpoint rule	Rectangles, height from the interval's own midpoint
Trapezoidal rule	Trapezoids — the average of left and right heights

VERIFIED DIRECTLY — CONVERGENCE RATES GENUINELY DIFFER

Approximating `∫x²dx` from `0` to `3` (true value `9`): at `n=100` rectangles, left/right Riemann error is `≈0.135`, while midpoint and trapezoidal error is `≈0.0002–0.0005` — roughly **270-600× more accurate** at the identical `n`. Left/right error shrinks linearly with `1/n`; midpoint and trapezoidal error shrinks with `1/n²` — the exact same forward-vs-central-difference pattern Chapter 4 found for derivatives, now showing up in integration.

Euler's Method: Numerical Integration for Physics Simulation

If acceleration is known, integrating once gives velocity; integrating again gives position. A physics engine does this numerically, one small time step `dt` at a time — **Euler's method**: `v_new = v + a·dt`, `x_new = x + v·dt`.

VERIFIED DIRECTLY AGAINST THE EXACT ANALYTICAL SOLUTION

Simulating a falling object (`x0=100`, `v0=0`, `a=−9.8 m/s²`) with `dt=0.01`, for 300 steps (`t=3s`): Euler's method gives `x=56.047`, `v=−29.400`. The exact formula, `x(t)=x0+v0t+½at²`: `x=55.900`, `v=−29.400`. Velocity matches **exactly**; position is off by `0.147` — a small, real, honest numerical error, not a bug.

VERIFIED DIRECTLY — THE ERROR IS GENUINELY FIRST-ORDER

Position error at `t=3`: `dt=0.1 → error=1.470`, `dt=0.01 → error=0.147`, `dt=0.001 → error=0.0147`, `dt=0.0001 → error=0.00147` — the error shrinks by *exactly* a factor of `10` every time `dt` does. Euler's method is only first-order accurate, the physics-simulation equivalent of Chapter 4's own forward difference — real game engines use smaller time steps or more sophisticated integrators (Runge-Kutta methods, out of this course's own scope) specifically to control this exact error.

Numerical Integration in Code

```
def midpoint_rule(f, a, b, n):
    dx = (b - a) / n
    return sum(f(a + (i+0.5)*dx) for i in range(n)) * dx

def euler_step(x, v, a, dt):
    return x + v*dt, v + a*dt   # position uses the OLD velocity

# simulate a falling object
x, v, a, dt = 100, 0, -9.8, 0.01
for _ in range(300):
    x, v = euler_step(x, v, a, dt)
print(x, v)   # 56.047, -29.400 -- close to the exact 55.900, -29.400
```

Hands-On Exercises

EXERCISE 1

Using the Fundamental Theorem of Calculus, find `∫x³ dx` from `0` to `2` using the antiderivative `F(x)=x⁴/4`. Then approximate the same area using the trapezoidal rule with `n=4` intervals, and compare.

View solution

EXERCISE 2

Using Euler's method, simulate one second of a ball thrown straight up with `v0=15 m/s`, `x0=0`, `a=−9.8 m/s²`, using `dt=0.5` (2 steps). Show each step's position and velocity, then compare your final position to the exact formula `x(t)=x0+v0t+½at²` at `t=1`.

View solution

EXERCISE 3

A colleague says "Euler's method computed velocity exactly right in this chapter's own falling-object example, so it must be a perfectly accurate method for physics simulation." Using this chapter's own verified findings, explain what's wrong with generalizing from the velocity result to the method as a whole.

View solution

CHAPTER 9 QUICK REFERENCE

- **Integral:** accumulated (signed) area under a curve; the Fundamental Theorem connects it directly to antiderivatives, `F(b)−F(a)`
- Verified: the antiderivative and a fine-grained numerical approximation agree to within `0.000002`
- Midpoint and trapezoidal rules converge with `1/n²`; left/right Riemann sums only converge with `1/n` — verified directly, mirroring Chapter 4's central-vs-forward-difference finding
- **Euler's method:** `v_new=v+a·dt`, `x_new=x+v·dt` — the numerical integration real physics engines run every frame
- Verified against an exact analytical solution: velocity exact, position off by a small, genuinely first-order error that shrinks linearly with `dt`
- **Next chapter:** Capstone — optimizing a function from scratch

CHAPTER 10 OF 10

Capstone — Optimizing a Function From Scratch

Calculus & Optimization

Chapter 10 · Capstone — Optimizing a Function From Scratch

One continuous project: fitting a straight line to real data using nothing but gradient descent, built entirely from scratch — the exact same procedure, at genuinely tiny scale, that trains real machine learning models on real datasets.

STEP	TASK	CHAPTER(S) USED
1	Define the model and the loss function	Ch.3-4
2	Derive the gradients via the chain rule	Ch.3, Ch.5, Ch.8
3	Verify the gradients numerically	Ch.4
4	Run gradient descent to convergence	Ch.6
5	Cross-check against the exact closed-form answer	Ch.9-style verification discipline
6	Confirm convexity: five starting points, one answer	Ch.7

Step 1 — The Model and the Loss Function

CH.3-4

Five data points: `(1,3), (2,5), (3,7), (4,8), (5,11)`. A linear model, `ŷ = mx+b`, and Mean Squared Error loss: $L(m,b) = (1/n)\cdot\Sigma(mx_i+b-y_i)^2$ — exactly the same squared-error shape Chapter 8's own network used, now averaged across many data points instead of one.

Step 2 — Deriving the Gradients via the Chain Rule

CH.3, CH.5, CH.8

Exactly Chapter 8's own derivation pattern: an outer square wrapped around an inner linear function, differentiated per data point and summed. `∂L/∂m = (2/n)·Σ(mx_i+b−y_i)·x_i` . `∂L/∂b = (2/n)·Σ(mx_i+b−y_i)` .

Step 3 — Verified Against Numerical Differentiation

VERIFIED DIRECTLY — AT M=1.0, B=0.5

Analytical: `∂L/∂m = −23.4` , `∂L/∂b = −6.6` . Numerical (central difference, `h=1e−6`): `−23.400000` , `−6.600000` — matching to 8 decimal places, the exact same cross-check discipline Chapter 8 used on its own network.

Step 4 — Gradient Descent, Run From Scratch

CH.6

Starting at `(m,b)=(0,0)` , learning rate `α=0.01` , 2000 iterations:

VERIFIED DIRECTLY

`iter 0: m=0.484, b=0.136, loss=31.32` → `iter 10: m=1.937, b=0.558, loss=0.33` → `iter 100: m=2.005, b=0.722, loss=0.17` → `iter 1000: m=1.905, b=1.082, loss=0.1401` → `iter 1999: m=1.900, b=1.099, loss=0.1400` . Converged.

Step 5 — Cross-Checked Against the Exact Answer

VERIFIED DIRECTLY — GRADIENT DESCENT MATCHES THE CLOSED-FORM SOLUTION

The exact least-squares formula (solvable directly from the data, no iteration needed) gives `m=1.9` , `b=1.1` exactly, loss `0.14` . Gradient descent's own from-scratch result after 2000 iterations: `m=1.900169` , `b=1.099391` — differing by only `0.0002` and `0.0006` respectively. An iterative search, knowing nothing about calculus's own closed-form shortcuts, found essentially the exact right answer purely by following the gradient downhill.

Step 6 — Confirming Convexity: The Direct Opposite of Chapter 7

CH.7

Chapter 7 showed gradient descent landing in genuinely different places depending on where it started, on a non-convex function. MSE loss for linear regression is provably convex — this predicts the opposite should happen here.

VERIFIED DIRECTLY — FIVE WILDLY DIFFERENT STARTING POINTS, ONE ANSWER

`(0,0) → (1.900, 1.099)`. `(10,10) → (1.898, 1.107)`. `(-5,-5) → (1.901, 1.096)`. `(100,-50) → (1.923, 1.017)`. `(-20,30) → (1.890, 1.137)`. Every single starting point — including two genuinely extreme ones — converges toward the identical answer, `m≈1.9, b≈1.1`. Exactly the guarantee Chapter 7 promised for convex functions, now confirmed directly rather than assumed.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: full real-analysis rigor, multivariable calculus beyond gradients, and differential equations beyond Euler's method were all named as deliberately out of scope, and stayed out of scope through all ten chapters. This capstone fit a two-parameter model; real models have millions, but the underlying mathematics — gradients, the chain rule, gradient descent, convexity — is exactly what was built here, just at a scale this course never claimed to reach.

Where This Course Connects

Linear Algebra Fundamentals' own vector and dot-product material underwrote Chapter 5's gradient directly. Algorithms & Complexity's own iterative-algorithm framing described gradient descent's own shape from the start. Boolean Algebra & Digital Logic's sigmoid-adjacent reasoning (activation functions) connects directly to Chapter 8's own network. Within this subject, a future Numerical Methods & Floating-Point Computation course would pick up directly where Chapter 1's own floating-point catastrophic-cancellation finding and Chapter 9's Euler's-method error analysis left off.

Hands-On Exercises

EXERCISE 1

Using this chapter's own MSE loss formula, compute `L(m=2, b=1)` directly for the five data points `(1,3),(2,5),(3,7),(4,8),(5,11)`, showing each squared-error term.

View solution

EXERCISE 2

Using this chapter's own gradient formulas, compute `∂L/∂m` and `∂L/∂b` at `m=2, b=1` for the same five data points, and use them to compute one gradient descent step with `α=0.01` starting from `(2,1)`.

View solution

EXERCISE 3

Explain, using this chapter's own Step 6 findings compared against Chapter 7's own non-convex example, what specifically would need to be true about a loss function for a machine learning engineer to be confident that training from a completely random starting point will reliably reach the best possible answer, versus a case where trying several different random starting points and keeping the best result becomes a genuinely necessary strategy.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked project:** model + loss (Ch.3-4) → chain-rule gradients (Ch.5, Ch.8) → numerical verification (Ch.4) → gradient descent to convergence (Ch.6) → cross-checked against the exact closed-form answer → convexity confirmed via five starting points (Ch.7)
- Gradient descent from scratch matched the exact least-squares solution to within `0.0006`
- Five wildly different starting points — including two genuinely extreme ones — all converged to the same answer, confirming Chapter 7's own convexity guarantee directly
- Out of scope: full real-analysis rigor, multivariable calculus beyond gradients, differential equations beyond Euler's method
- **Course complete** — Calculus & Optimization, 10 chapters, from the limit definition of a derivative to a working, verified, from-scratch model fit