Numerical Methods & Floating-Point Computation

A Complete 10-Chapter Maths for Programmers Course

Topics covered:

IEEE 754 representation & machine epsilon · catastrophic cancellation

Numerical stability & conditioning · root-finding methods

Numerical linear algebra pitfalls · differentiation & integration, revisited

Capstone: auditing and fixing eight real floating-point bugs in a small codebase

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

01 Why Numerical Methods Matter for Programmers

02 IEEE 754 Floating-Point Representation

03 Rounding Error & Machine Epsilon

04 Catastrophic Cancellation & Loss of Significance

05 Numerical Stability: When the Algorithm Is the Problem

06 Error Propagation & Conditioning

07 Root-Finding Methods

08 Numerical Linear Algebra Pitfalls

09 Numerical Differentiation & Integration, Revisited

10 Capstone — Diagnosing and Fixing Numerical Bugs in Real Code

CHAPTER 1 OF 10

Why Numerical Methods Matter for Programmers

Numerical Methods & Floating-Point Computation

Chapter 1 · Why Numerical Methods Matter for Programmers

Calculus & Optimization's own Chapter 1 ran into a real problem and deliberately left it half-explained: shrinking a numerical-differentiation step size `h` improves accuracy for a while, then suddenly makes it much worse. Linear Algebra Fundamentals solved real systems of equations and computed real matrix inverses, always on numbers that behaved politely. Algorithms & Complexity measured how many operations an algorithm performs — but never asked whether each individual operation gives back a trustworthy answer. This course asks that question directly: when a computer stores and manipulates real numbers, exactly how much can you trust the result, and when does that trust quietly break down?

The Problem, Stated Precisely

Almost every programmer eventually runs this in some language and is surprised by it:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

VERIFIED DIRECTLY — THE EXACT NUMBERS INVOLVED

`0.1 + 0.2` evaluates to `0.30000000000000004440892...`, while the literal `0.3` stores as `0.29999999999999998889777...`. The two differ by about `5.55 × 10⁻¹⁷` — a genuinely tiny gap, but a real, non-zero one, which is exactly why `0.1 + 0.2 == 0.3` evaluates to `False` in Python, JavaScript, Java, C, C#, Go, Rust, and effectively every other mainstream language. This isn't a bug in any one of them — it's a direct, shared consequence of how **every** one of them stores a fractional number in binary, covered in full in Chapter 2.

This chapter deliberately doesn't yet explain *why* this happens at the bit level — that's Chapter 2's job. What matters here is that this is not a rare edge case. It is the default, everyday behavior of floating-point arithmetic, and it has real, verifiable consequences for ordinary code.

A Real Consequence: A Loop That Doesn't Do What It Looks Like It Does

Consider a loop written by someone who reasonably expects that adding `0.1` ten times gives exactly `1.0`, or that repeatedly adding `0.1` will eventually land exactly on `1.0` and stop:

```
total = 0.0
for i in range(10):
    total += 0.1
print(total)          # expected: 1.0

x = 0.0
while x != 1.0:     # expected: stops once x reaches 1.0
    x += 0.1
```

VERIFIED DIRECTLY — BOTH EXPECTATIONS ARE WRONG

Summing `0.1` ten times produces `0.9999999999999999`, not `1.0` — off by about `1.1 × 10⁻¹⁶`. Worse, the `while x != 1.0` loop **never terminates as written**: `x` creeps past `1.0` without ever landing on it exactly — after 20 iterations it has already overshot to `2.0000000000000004` and will keep climbing forever, since it can never satisfy `x == 1.0`. A loop that looks obviously correct by inspection runs indefinitely in practice.

Scale the same idea up and the error compounds rather than staying microscopic: adding the value `0.10` one million times — the kind of thing a naive running-total accumulator in a billing or accounting system might do — produces a result off from the mathematically exact answer by roughly `1.33 × 10⁻⁶`. Still small in absolute terms, but no longer negligible once real money or a strict equality check is on the other end of that number, and the error grows with every additional operation rather than staying fixed.

What This Course Actually Covers

TOPIC	WHERE IT ACTUALLY SHOWS UP
IEEE 754 representation (Ch.2)	Why `0.1` can't be stored exactly in binary in the first place — the root cause behind every example in this chapter
Rounding error & machine epsilon (Ch.3)	Putting a precise number on "how wrong" a floating-point value can be, instead of just observing that it is
Catastrophic cancellation (Ch.4)	Why Calculus & Optimization Chapter 1's own numerical-derivative approximation broke down as its step size `h` got too small — resolved directly in this course
Numerical stability (Ch.5-6)	Why two mathematically identical formulas can give wildly different answers once real floating-point numbers are involved
Root-finding & linear algebra pitfalls (Ch.7-8)	Where Calculus & Optimization's derivatives and Linear Algebra Fundamentals' Gaussian elimination can silently misbehave on real hardware

What This Course Won't Cover

Numerical computing as a full field is enormous, and much of it is out of scope for what a working programmer actually needs day to day:

- **Symbolic / exact computer algebra** — systems like a computer algebra system that manipulate exact fractions or symbolic expressions sidestep floating-point error entirely by design; this course is about the arithmetic every mainstream language actually performs by default, not about avoiding it
- **GPU-specific and parallel floating-point quirks** — reduced-precision formats, non-associative parallel summation order, and hardware-specific fused multiply-add behavior are real and important for high-performance/ML infrastructure work, but are a specialized extension of this course's own fundamentals, not covered here
- **Arbitrary-precision and interval arithmetic libraries** — genuinely useful tools that trade speed for guaranteed precision; this course focuses on understanding and working with standard double-precision floats, the default nearly every language reaches for first

> **WHY THIS SCOPE, SPECIFICALLY**
>
> Every topic from Chapter 2 onward exists because it's a direct prerequisite for recognizing, diagnosing, or fixing a real floating-point bug in ordinary application code — not because it's interesting in the abstract. Chapter 10's own capstone is a deliberate proof of that: a small, ordinary-looking codebase, audited chapter by chapter for exactly the kinds of bugs this course covers.

Where This Course Is Headed

CHAPTER	TOPIC
2	IEEE 754 Floating-Point Representation
3	Rounding Error & Machine Epsilon
4	Catastrophic Cancellation & Loss of Significance
5	Numerical Stability: When the Algorithm Is the Problem
6	Error Propagation & Conditioning
7	Root-Finding Methods
8	Numerical Linear Algebra Pitfalls
9	Numerical Differentiation & Integration, Revisited
10	Capstone — Diagnosing and Fixing Numerical Bugs in Real Code

THIS COURSE'S THROUGHLINE

Every chapter answers a version of the same question: given that a computer's floating-point numbers are only ever *approximately* correct, how do you know how much to trust a given result — and how do you write code that stays trustworthy anyway? Chapter 2 explains where the approximation comes from; Chapters 3-6 give you the vocabulary and tools to measure and reason about it precisely; Chapters 7-9 apply all of it to real algorithms this subject has already built (root-finding, Gaussian elimination, numerical calculus); and Chapter 10 puts the whole toolkit to work on a realistic piece of code.

Hands-On Exercises

EXERCISE 1

Using this chapter's own verified numbers, explain precisely why `0.1 + 0.2 == 0.3` evaluates to `False`. Your answer should name the actual gap between the two values, not just say "floating-point is imprecise."

View solution

EXERCISE 2

A colleague writes a loop that repeatedly adds `0.1` to a running total and stops with `while x != 1.0:`. Using this chapter's own verified finding, explain exactly what goes wrong, and propose a one-line fix that would make the loop terminate reliably.

View solution

EXERCISE 3

This chapter says the "add 0.1 a million times" error (about 1.33×10^{-6}) grows as more additions happen, rather than staying fixed the way the ten-addition error did. Explain, in your own words and without yet knowing Chapter 3's formal error-propagation rules, why doing *more* additions with the same small rounding error per step would plausibly make the total error larger rather than smaller.

View solution

CHAPTER 1 QUICK REFERENCE

- `0.1 + 0.2 == 0.3` is `False` in effectively every mainstream language — verified: the two sides differ by about 5.55×10^{-17}
- This is a shared, structural consequence of binary floating-point storage, not a bug in any one language — explained fully in Chapter 2

- Verified directly: a `while x != 1.0` loop built on repeated `0.1` additions never terminates, since `x` overshoots `1.0` without ever landing on it exactly
- Error from repeated floating-point addition compounds rather than staying fixed — verified: summing `0.1` a million times is off by about `1.33 × 10⁻⁶`
- Course scope: representation, rounding, cancellation, stability, and error propagation for standard double-precision floats — **not** symbolic/exact computer algebra, GPU-specific quirks, or arbitrary-precision libraries
- **Next chapter:** IEEE 754 floating-point representation — where the `0.1 + 0.2` gap actually comes from, bit by bit

CHAPTER 2 OF 10

IEEE 754 Floating-Point Representation

Numerical Methods & Floating-Point Computation

Chapter 2 · IEEE 754 Floating-Point Representation

Chapter 1 showed *that* `0.1 + 0.2 != 0.3`, verified down to the exact numbers involved, but deliberately stopped short of explaining *why*. This chapter opens the box: the standard almost every language uses to store a real number — **IEEE 754** — and the specific reason a number as ordinary as `0.1` can never be stored exactly in it.

The Three-Part Layout: Sign, Exponent, Mantissa

A standard **double-precision** float (64 bits — `double` in C/Java, the only numeric float type in JavaScript, Python's default `float`) splits its bits into three fields:

The stored value is reconstructed as `(-1)^sign × (1 + mantissa/2⁵²) × 2^(exponent−1023)`. The "`1 +`" is the **implicit leading bit** — every normal number is assumed to start with a binary `1.xxxxx`, so that one bit doesn't need to be stored at all, silently giving 53 bits of precision from only 52 stored mantissa bits.

VERIFIED DIRECTLY — DECOMPOSING 0.1'S ACTUAL STORED BITS

`0.1`'s 64 bits, read directly: sign `= 0`, raw exponent `= 1019` (unbiased: `1019 − 1023 = −4`), mantissa `= 0x999999999999a`. Plugging back into the formula: `(1 + 0x999999999999a / 2⁵²) × 2⁻⁴` reconstructs to **exactly** the same value Python stores for `0.1` — confirming the formula, not just asserting it.

Why 0.1 Specifically Can Never Be Exact

Every stored float is, structurally, a binary fraction times a power of two — nothing else is representable. Expanding `0.1` as a binary fraction by repeated doubling (the standard technique, exactly analogous to long division for decimal expansions):

```
0.1 (decimal) = 0.0001100110011001100110011001100110011... repeating "0011" forever
0.2 (decimal) = 0.0011001100110011001100110011001100110... repeating "0011" forever
```

THE ACTUAL ROOT CAUSE

`0.1`'s binary expansion is **infinitely repeating** — the pattern `0011` never terminates, exactly the way `1/3` never terminates in decimal (`0.333...`). A computer only has 52 mantissa bits to work with, so it must cut that infinite pattern off and round — which is precisely the `...999999999999a` tail seen in `0.1`'s decomposition above. There is no larger number of bits that fixes this; *any* finite binary format has the same problem, because the issue is the repeating pattern itself, not a shortage of bits. This is exactly the same phenomenon as decimal being unable to store `1/3` exactly — floating point just hits it far more often, because so many ordinary decimal fractions (`0.1`, `0.2`, `0.3`, `0.7`...) turn out to be repeating in binary even though they terminate cleanly in decimal.

Not every decimal fraction has this problem — `0.5`, `0.25`, and `0.125` are all exact powers of two (`2⁻¹`, `2⁻²`, `2⁻³`) and store perfectly, with an all-zero mantissa. The problem is specifically fractions whose denominator, in lowest terms, isn't a pure power of 2.

Single vs. Double Precision

FORMAT	TOTAL BITS	SIGN	EXPONENT	MANTISSA	DECIMAL DIGITS OF PRECISION
Single (`float32`)	32	1	8 (bias 127)	23	~7
Double (`float64`)	64	1	11 (bias 1023)	52	~15-17

VERIFIED DIRECTLY — THE SAME NUMBER, TWO FORMATS

`0.1` decomposed as a 32-bit float: sign `= 0`, raw exponent `= 123` (unbiased `-4` — identical exponent to the double, since `0.1`'s magnitude doesn't change), mantissa `= 0x4ccccd`. The unbiased exponent matches the double exactly; only the mantissa is shorter (23 bits instead of 52), which is exactly why single precision is *less accurate* at representing `0.1`, not differently rounded in some unrelated way.

Python's built-in `float`, JavaScript's `Number`, Java/C#'s `double`, and C's `double` are all double-precision by default. Single precision (`float` in C/Java, `Float32Array` in JavaScript) shows up mainly where memory or speed matters more than precision — graphics, GPUs, and large numeric arrays.

Subnormal Numbers: Gradual Underflow

The exponent field's all-zero value (`0`) is reserved as a special signal: it means "drop the implicit leading `1` bit" and switch to **subnormal** representation, allowing numbers smaller than the smallest normal float, at the

cost of gradually losing precision as they shrink.

VALUE	WHAT IT IS
≈ 2.2250738585072014 × 10^{-308}	Smallest *normal* positive double (exponent field `= 1`, smallest non-subnormal)
≈ 4.9406564584124654 × 10^{-324}	Smallest *subnormal* positive double — verified exponent field `= 0`, mantissa `= 1` (a single bit)

WHY "GRADUAL" UNDERFLOW MATTERS

Without subnormals, a floating-point value shrinking toward zero would suddenly jump straight from the smallest normal number to exactly `0.0` — a discontinuous cliff. Subnormals fill that gap with a smoothly shrinking (if increasingly imprecise) sequence of tiny nonzero values instead, which matters for numerical algorithms — covered later in this course — that rely on results changing continuously rather than snapping to zero.

The Special Values: Infinity, NaN, and Signed Zero

Two more reserved exponent patterns give IEEE 754 its special values, alongside a genuine oddity: **zero has two distinct bit patterns.**

VALUE	BIT PATTERN SIGNAL	VERIFIED BEHAVIOR
Infinity (`±∞`)	Exponent all `1`s, mantissa all `0`	Represents overflow / division results too large to store — e.g. `1.0 / float('inf') == 0.0`, verified directly
NaN ("not a number")	Exponent all `1`s, mantissa nonzero	Represents an undefined result (e.g. `0/0`). Verified directly: `nan == nan` is `False` — NaN is defined to compare unequal to everything, including itself
Signed zero (`+0.0` / `-0.0`)	All bits zero except (for `-0.0`) the sign bit	Verified directly: `0.0 == -0.0` is `True` (they compare equal) yet their raw bit patterns genuinely differ (`0000...0000` vs `8000...0000`) — and division reveals the difference: `1.0 / +0.0` and `1.0 / -0.0` would produce `+∞` and `-∞` respectively under IEEE division

A REAL, LANGUAGE-SPECIFIC GOTCHA: DIVISION BY ZERO DOESN'T ALWAYS MEAN "INFINITY"

IEEE 754 itself defines `1.0 / 0.0` as `+Infinity`, and JavaScript and Java both follow that directly. Python deliberately overrides this at the language level and raises `ZeroDivisionError` instead, treating it as a program error rather than a valid floating-point result — verified directly in this environment. The underlying hardware and the IEEE 754 standard agree on what *should* happen; the language you're using can still choose to intercept it.

A GENUINELY USEFUL, VERIFIED NAN PROPERTY

Because `nan != nan` is `True` — the only value in IEEE 754 that is never equal to itself — `x != x` is a real, working way to test whether a floating-point value `x` is NaN, without needing a dedicated `isnan()` function at all (though using one, like Python's `math.isnan()`, is clearer and the recommended approach in real code).

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
0.1's infinite repeating binary expansion	Directly resolves Chapter 1's own unexplained `0.1 + 0.2 != 0.3` finding — now traced to its exact bit-level cause
52 stored mantissa bits, 53 with the implicit leading bit	Sets up Chapter 3's formal definition of machine epsilon — the precise size of the smallest gap between representable numbers
Rounding a value to fit the mantissa	Sets up Chapter 4's catastrophic cancellation, which is fundamentally about what happens when that rounding is subtracted out

Hands-On Exercises

EXERCISE 1

Using this chapter's own reconstruction formula `(-1)^sign × (1 + mantissa/2⁵²) × 2^(exponent−1023)` and the verified decomposition of `1.0` (sign=0, raw exponent=1023, mantissa=0), show step by step that the formula reconstructs exactly `1.0`.

View solution

EXERCISE 2

Explain, using this chapter's own binary-expansion argument, why `0.5` stores in a float with zero error, while `0.1` and `0.2` do not. Your answer should reference what makes a fraction's binary expansion terminate versus repeat forever.

View solution

EXERCISE 3

A function receives a floating-point value `x` from an untrusted external source and needs to check whether it's a valid number before using it in a calculation. Using this chapter's own verified NaN property, explain why a naive check like `if x == some_error_sentinel` would fail to catch a NaN value, and what check would actually work.

View solution

CHAPTER 2 QUICK REFERENCE

- A double-precision float = 1 sign bit + 11 exponent bits (bias 1023) + 52 mantissa bits, with an implicit leading `1` giving 53 bits of real precision
- Verified: `0.1`'s binary expansion repeats forever (`0011` pattern), so it must be rounded to fit 52 bits — the exact, bit-level cause of Chapter 1's `0.1+0.2 != 0.3`
- Single precision (32-bit) uses 8 exponent bits + 23 mantissa bits — same exponent range logic, far less mantissa precision (~7 vs. ~16 decimal digits)
- **Subnormals** (exponent field `= 0`) allow gradual underflow instead of a sudden jump to zero
- Special values: `Infinity` (exponent all 1s, mantissa 0), `NaN` (exponent all 1s, mantissa nonzero, verified never equal to itself), and signed zero (`+0.0` / `-0.0`, equal by `==` but distinct bit patterns)
- Verified: Python raises `ZeroDivisionError` on `1.0/0.0` rather than returning IEEE 754's own defined `+Infinity` — a language choice layered on top of the standard, not the standard itself
- **Next chapter:** Rounding error and machine epsilon — putting an exact number on how much any of this can go wrong

CHAPTER 3 OF 10

Rounding Error & Machine Epsilon

Numerical Methods & Floating-Point Computation

Chapter 3 · Rounding Error & Machine Epsilon

Chapter 2 explained *why* a value like `0.1` gets rounded when it's stored. This chapter puts an exact number on *how much* that rounding can possibly be — the single most important quantity in this whole course for reasoning precisely about floating-point trustworthiness, rather than just gesturing at "some small error."

Absolute Error vs. Relative Error

Absolute error is simply `|approximation − true value|` — the raw size of the gap. **Relative error** divides that gap by the true value's own size: `|approximation − true value| / |true value|`. The same absolute error can mean something completely different depending on the scale of the number involved.

VERIFIED DIRECTLY — IDENTICAL ABSOLUTE ERROR, WILDLY DIFFERENT MEANING

Case 1: true value `1,000,000.0`, approximation `1,000,000.0001` — absolute error `≈ 0.0001`, but relative error `≈ 1.0 × 10⁻¹⁰` (utterly negligible). **Case 2:** true value `0.0001`, approximation `0.0002` — absolute error is the exact same `0.0001`, but relative error is `1.0` — the approximation is **100% wrong**, literally double the true value. Absolute error alone told you nothing useful; the two cases look identical by that measure and are utterly different in reality.

WHY THIS COURSE LEANS ON RELATIVE ERROR

Because floating-point precision itself scales with the size of the number (per Chapter 2's own bit layout — the exponent shifts to keep roughly the same number of significant mantissa bits regardless of magnitude), relative error is almost always the more meaningful measure for judging whether a floating-point result is trustworthy, and is the default lens used for the rest of this course.

Machine Epsilon, Derived — Not Quoted

Machine epsilon is the smallest positive number `ε` such that `1.0 + ε` is a different, representable floating-point value from `1.0` itself. Rather than simply stating its value, it can be found directly with a genuinely simple experiment: start with `ε = 1.0` and keep halving it until adding it to `1.0` stops making a difference.

```
eps = 1.0
while 1.0 + eps != 1.0:
    eps /= 2
# eps is now too small to matter -- the *previous* value of eps
# (eps * 2) is the actual machine epsilon
```

VERIFIED DIRECTLY — THE HALVING EXPERIMENT

Running this loop takes exactly **53 halvings** to reach a point where `1.0 + ε` stops changing. The last value of `ε` for which `1.0 + ε != 1.0` was still true is `2.220446049250313 × 10⁻¹⁶` — which is **exactly** Python's own built-in `sys.float_info.epsilon`, and exactly `2⁻⁵²`. This isn't a coincidence: it's the direct, mechanical consequence of Chapter 2's own 52-bit mantissa — `2⁻⁵²` is precisely the size of the smallest possible adjustment representable at a magnitude around `1.0`, and the halving experiment rediscovers that fact from first principles rather than being told it.

Machine epsilon is **not** "the smallest number a computer can represent" — Chapter 2's subnormal numbers go vastly smaller than this. It specifically measures precision *relative to a value near* `1.0` — how fine-grained the gaps between adjacent representable floats are at that scale. Because of the floating-point format's own sliding exponent, this same *relative* gap size holds at any magnitude — which is exactly why Chapter 2's "absolute vs. relative error" distinction matters so much here.

The Four IEEE Rounding Modes

Whenever an arithmetic result doesn't fit exactly into the available mantissa bits, it has to be rounded to the nearest representable value — and IEEE 754 defines specific, precise rules for how that rounding happens. Four modes are the ones exposed by virtually every language and standard library:

MODE	RULE	VERIFIED EXAMPLE
Round to nearest, ties to even (the default)	Round to the closest representable value; on an exact tie, round to whichever neighbor has an even last digit	`round(0.5)=0`, `round(1.5)=2`, `round(2.5)=2`, `round(3.5)=4` — every tie breaks toward the even number
Round toward zero (truncation)	Always round toward `0`, discarding everything past the cutoff	`trunc(2.7)=2`, `trunc(-2.7)=-2`
Round toward `+∞` (ceiling)	Always round up, toward positive infinity	`ceil(2.3)=3`, `ceil(-2.3)=-2`
Round toward `−∞` (floor)	Always round down, toward negative infinity	`floor(2.7)=2`, `floor(-2.7)=-3`

(Honest note: the IEEE 754 standard technically also defines a fifth rounding attribute, round-ties-away-from-zero, but it's rarely exposed as a default in mainstream hardware or languages the way the four above are — round-to-nearest-ties-to-even is overwhelmingly the one that matters day to day, which is why the rest of this section focuses on it.)

Why "Ties to Even" Specifically — A Verified Demonstration

Round-half-up (the rounding most people learn in school — `0.5` always rounds to `1`) seems more intuitive than round-half-even. But it has a real, measurable flaw: applied repeatedly to many values that each happen to land exactly on a `.5` boundary, it introduces a **systematic upward bias**, because every single tie breaks the same direction.

VERIFIED DIRECTLY — 1,000 EXACT .5 TIES, ROUNDED TWO DIFFERENT WAYS

Rounding the 1,000 values `0.5, 1.5, 2.5, ..., 999.5` (whose exact sum is `500,000.0`) individually, then summing the rounded results: **round-half-up** gives a total of `500,500` — a bias of `+500`, since every single one of the 1,000 ties rounded upward. **Round-half-to-even** gives a total of exactly `500,000` — **zero bias**, because the ties alternate between rounding up and down (evenly, since consecutive integers alternate even/odd) and the errors cancel out over many roundings.

WHY THIS MATTERS FOR REAL CODE, NOT JUST THEORY

Any system that rounds a large number of values repeatedly — currency calculations, sensor readings, aggregated statistics — accumulates this same kind of bias if it uses a "round half up" rule naively. This verified 1,000-value experiment is a small-scale version of exactly the kind of accumulated, compounding error Chapter 1 already previewed with its million-addition example — and it's precisely why IEEE 754 made round-to-even the default, not round-half-up.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Machine epsilon `= 2⁻⁵²`, derived directly from the mantissa	Chapter 4's catastrophic cancellation — the precise threshold at which subtracting two close numbers destroys essentially all remaining precision
Relative error as the meaningful measure	Chapter 6's error propagation and conditioning, both built entirely on relative rather than absolute error
Per-operation rounding error compounding over many operations	Directly formalizes Chapter 1's own million-addition finding and this chapter's own 1,000-tie bias experiment into a general principle

Hands-On Exercises

EXERCISE 1

A measurement system reports a true value of `50.0` with an approximation of `50.5`, and a second true value of `0.05` with an approximation of `0.055`. Compute the absolute and relative error for both cases, and explain which one represents the more serious measurement problem, using this chapter's own reasoning about why relative error is usually the more meaningful measure.

View solution

EXERCISE 2

Using this chapter's own halving-experiment logic, explain why the loop `while 1.0 + eps != 1.0: eps /= 2` must eventually stop — that is, why continually halving `eps` is guaranteed to eventually produce a value small enough that adding it to `1.0` makes no difference, rather than looping forever.

View solution

EXERCISE 3

A billing system needs to round exactly-half-cent amounts (values ending precisely in `.5` cents) many thousands of times per day. Using this chapter's own verified 1,000-tie experiment, explain specifically why choosing round-half-up over round-half-to-even for this system would be a real, measurable design mistake rather than a harmless stylistic choice.

View solution

CHAPTER 3 QUICK REFERENCE

- **Absolute error** `= |approx − true|`; **relative error** `= |approx − true| / |true|` — verified: the same absolute error (`0.0001`) meant a negligible 10^{-10} relative error in one case and a catastrophic `1.0` (100%) relative error in another
- **Machine epsilon** — the smallest `ε` where `1.0 + ε != 1.0` — derived directly by halving: $2^{-52} \approx 2.22 \times 10^{-16}$, matching Chapter 2's own 52-bit mantissa exactly
- Four common IEEE rounding modes: **round-to-nearest-ties-to-even** (the default), round toward zero, round toward `+∞`, round toward `−∞`
- Verified: rounding 1,000 exact `.5` ties with round-half-up produces a `+500` systematic bias; round-half-to-even produces zero bias — the real, measured reason IEEE 754 defaults to ties-to-even
- **Next chapter:** Catastrophic cancellation — what happens when subtraction meets two numbers that are already close to their own rounding limit

CHAPTER 4 OF 10

Catastrophic Cancellation & Loss of Significance

Numerical Methods & Floating-Point Computation

Chapter 4 · Catastrophic Cancellation & Loss of Significance

Chapter 3 established machine epsilon — the smallest gap between representable values near `1.0`. This chapter shows the single most common way that gap turns into a real, visible bug: **subtracting two floating-point numbers that are very close to each other.** The operation itself is completely correct by IEEE 754's own rules — it's what subtraction does *to your remaining precision* that causes the damage.

The Core Mechanism

Every floating-point number carries roughly 15-17 significant decimal digits of precision (Chapter 2's 52-bit mantissa). When two numbers that agree in their first several digits are subtracted, those matching leading digits **cancel out exactly** — but the trailing digits that made each number only approximately correct in the first place don't cancel; they're what's left over. The result can end up built almost entirely out of rounding error, even though the subtraction itself was performed with full precision.

VERIFIED DIRECTLY — PRECISION COLLAPSING AS TWO NUMBERS CONVERGE

Computing `(1 + x) - 1`, which is mathematically always exactly `x`, for shrinking values of `x`: `x=10⁻⁸` gives a relative error of only `6.08 × 10⁻⁹` (still fine) — but `x=10⁻¹⁵` gives a relative error of `0.11` (11% wrong), and at `x=10⁻¹⁶` the result is exactly `0.0`, a relative error of `1.0` — `x` has vanished completely. Nothing "went wrong" with the subtraction; `1+x` simply rounded to exactly `1.0` before the subtraction ever happened, because `x` was smaller than machine epsilon relative to `1.0`.

The Classic Case: An Unstable Quadratic Formula

The quadratic formula, `x = (−b ± √(b² − 4ac)) / (2a)`, is taught as a single, universal recipe — but implemented naively in floating point, it has a real, well-known failure mode whenever `b` is much larger than `a` and `c`: one of the two `±` branches subtracts two nearly-equal numbers.

```
# a=1, b=-100000, c=1  -->  roots near 99999.99999 and 0.00001
disc = b*b - 4*a*c
sq = math.sqrt(disc)
root_big   = (-b + sq) / (2*a)   # fine: -b and sq have the same sign here
root_small = (-b - sq) / (2*a)   # danger: -b and -sq nearly cancel
```

VERIFIED DIRECTLY — THE NAIVE FORMULA'S SMALL ROOT IS BADLY WRONG

Comparing against a high-precision (50-digit) reference computation: the naive formula's large root (`99999.99999`) is essentially perfect — relative error `≈ 3.4 × 10⁻¹⁷`, right at the limit of double precision. But its small root (`≈ 1.0000003385357559 × 10⁻⁵`) has a relative error of `≈ 3.4 × 10⁻⁷` — **ten orders of magnitude worse** than the large root, computed from the exact same inputs with the exact same formula. The only difference is which `±` branch happened to subtract two nearly-equal numbers.

The standard fix — sometimes called the "Citardauq" formula (read the standard quadratic formula's name backward) — restructures the computation so the two large, same-signed quantities are always *added*, never subtracted, and the small root is obtained by division instead:

```
sign_b = 1 if b >= 0 else -1
q = -0.5 * (b + sign_b * sq)   # always an addition of same-signed terms
root1 = q / a
root2 = c / q                    # the small root, via division -- no cancellation
```

VERIFIED DIRECTLY — THE STABLE FORMULA FIXES EXACTLY THE BROKEN ROOT

The stable formula's large root matches the naive formula's (relative error `≈ 3.4 × 10⁻¹⁷` — no change, since that branch was never the problem). Its small root, computed as `c / q` instead of by subtraction, has a relative error of just `≈ 1.1 × 10⁻¹⁶` — **essentially at the limit of double precision**, roughly nine orders of magnitude better than the naive formula's `3.4 × 10⁻⁷` for the identical mathematical root. Same inputs, same underlying mathematics, dramatically different reliability — purely a function of which arithmetic operations were used to get there.

Resolved: Calculus & Optimization's Own Unexplained Finding

Calculus & Optimization Chapter 1 verified that numerical differentiation, `(f(x+h) - f(x)) / h`, gets more accurate as `h` shrinks — until it suddenly gets catastrophically worse. That course stated the observation honestly but left the mechanism for this course to explain. It is exactly the cancellation pattern above.

VERIFIED DIRECTLY — WATCHING THE NUMERATOR COLLAPSE, STEP BY STEP

For `f(x)=x²` at `x=3` (so `f(x)=9.0`): as `h` shrinks, `f(x+h)` gets closer and closer to `9.0`, so the subtraction `f(x+h) − f(x)` loses more and more relative precision — its own relative error grows from `7.7 × 10⁻⁹` at `h=10⁻⁸` to `6.8 × 10⁻⁴` at `h=10⁻¹³`, and by `h=10⁻¹⁶`, `f(x+h)` rounds to **exactly** `9.0` — identical to `f(x)` — so the subtraction returns exactly `0.0`, a complete, 100% loss of the numerator, even though the true difference was a tiny but genuinely nonzero `6 × 10⁻¹⁶`.

WHY DIVIDING BY A TINY `H` AFTERWARD MAKES IT LOOK EVEN WORSE

Dividing by `h` doesn't add any new error — but it doesn't fix the numerator's damage either. It simply reveals it: the numerator's relative error *is* the final derivative approximation's relative error, since dividing by a number doesn't change relative error. A numerator that's already 100% wrong produces a derivative approximation that's 100% wrong — which is exactly why Calculus & Optimization's own experiment saw the approximation collapse to `0` at `h=10⁻¹⁶`, for a true derivative of `6`. There was never a "division by a small number" problem on its own — the real damage happened one step earlier, in the subtraction.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
Cancellation destroys precision when operands converge	Fully resolves Calculus & Optimization Chapter 1's own unexplained numerical-differentiation breakdown
Same formula, wildly different reliability depending on arithmetic structure	Sets up Chapter 5's general treatment of algorithm stability — this chapter's quadratic-formula fix is the first concrete example of it
The stable "Citardauq" reformulation trades subtraction for addition + division	A specific instance of the general reformulation strategy Chapter 9 applies again to numerical differentiation and integration

Hands-On Exercises

EXERCISE 1

Using this chapter's own `(1+x)-1` experiment, explain precisely why the result is exactly `0.0` at `x=10⁻¹⁶` rather than some small nonzero (if inaccurate) number. Your answer should reference machine epsilon from Chapter 3.

View solution

EXERCISE 2

For the quadratic `a=1, b=-100000, c=1`, this chapter verified the naive formula's small root has a relative error roughly ten orders of magnitude worse than its large root. Explain specifically which arithmetic step causes the difference, and why the large root's computation never runs into the same problem.

View solution

EXERCISE 3

Using this chapter's own step-by-step resolution of the numerical-differentiation breakdown, explain in your own words why "just use a smaller `h`" and "just use more decimal digits of precision" are not really two different potential fixes for the same problem, but are actually pulling in opposite directions once `h` gets small enough.

View solution

CHAPTER 4 QUICK REFERENCE

- **Catastrophic cancellation:** subtracting two nearly-equal floating-point numbers destroys relative precision, even though the subtraction itself is performed exactly per IEEE 754's own rules
- Verified: `(1+x)-1` for true value `x` goes from a negligible `6×10⁻⁹` relative error at `x=10⁻⁸` to a complete, 100% loss (`result=0.0`) at `x=10⁻¹⁶`
- The naive quadratic formula's small root, verified: relative error `≈3.4×10⁻⁷`; the algebraically-equivalent stable ("Citardauq") reformulation: relative error `≈1.1×10⁻¹⁶` — nine orders of magnitude better, for the same math
- The fix pattern: restructure the arithmetic so nearly-equal quantities are never directly subtracted — trade subtraction for addition-plus-division wherever the mathematics allows it
- Resolved directly: Calculus & Optimization Chapter 1's numerical-derivative breakdown was cancellation in the numerator `f(x+h)-f(x)`, not a problem with dividing by a small `h`
- **Next chapter:** Numerical stability — generalizing this chapter's one worked fix into a broader way of recognizing unstable algorithms

CHAPTER 5 OF 10

Numerical Stability: When the Algorithm Is the Problem

Numerical Methods & Floating-Point Computation

Chapter 5 · Numerical Stability: When the Algorithm Is the Problem

Chapter 4 fixed one specific formula — the quadratic formula's unstable branch — by restructuring a single subtraction. This chapter generalizes that idea: **numerical stability** is a property of an *algorithm*, not of the underlying mathematical problem. Two formulas that are perfectly, provably equal in exact arithmetic can behave completely differently once real floating-point rounding gets involved — and recognizing which formula you're looking at is a skill in its own right.

Two Equivalent Formulas for the Same Quantity: Variance

Statistical variance has two textbook-equivalent formulas. The **two-pass** formula computes the mean first, then averages the squared distance of each point from that mean. The **one-pass ("naive")** formula avoids a second pass over the data by expanding the algebra: `Var(X) = E[X²] − (E[X])²`.

```
# Two-pass (stable): center the data first, then square
mean = sum(data) / n
two_pass_var = sum((x - mean)**2 for x in data) / n

# One-pass ("naive"): sum of squares minus square of sum
mean_sq = sum(x*x for x in data) / n
naive_var = mean_sq - mean**2
```

Algebraically, these are exactly the same quantity — every statistics textbook derives one from the other with a few lines of expansion. In floating point, they are not remotely the same.

A Genuinely Dramatic, Fully Verified Failure

Take 9 real numbers, each equal to `20,000,000` plus a small random offset between `0` and `1` (a realistic shape for, say, sensor readings with a large fixed baseline and small genuine variation):

VERIFIED DIRECTLY — THE NAIVE FORMULA RETURNS A NEGATIVE VARIANCE

Computed on the same 9 real data points: the **two-pass formula** gives `0.0672569888025179`, matching a 60-digit high-precision reference calculation to a relative error of `≈7.1 × 10⁻¹⁷` — essentially exact. The **naive one-pass formula**, on the exact same data, gives `-0.0625`. Variance is mathematically defined as an average of squared quantities — it is **never negative**, by definition. The naive formula didn't just lose some accuracy; it returned a value that is provably impossible for the quantity it claims to compute.

WHY: THIS IS CHAPTER 4'S CANCELLATION, NOW HIDING INSIDE TWO LARGE SUMS

`E[X²]` and `(E[X])²` are both enormous numbers here — each close to `(2 × 10⁷)² = 4 × 10¹⁴` — while their true difference (the actual variance) is a tiny fraction of that, around `0.067`. Subtracting two huge, nearly-equal numbers to recover a tiny result is exactly the catastrophic-cancellation pattern from Chapter 4 — just spread across an entire sum-of-squares computation instead of a single visible subtraction. The rounding error accumulated while computing the two huge sums is, in this case, larger than the true answer itself, which is exactly how the result can end up negative.

The two-pass formula never runs into this problem, because it subtracts the mean from each data point **first** — before any squaring or summing happens. Every value it works with afterward is already small and close to zero (each of the 9 offsets, centered), so there are no huge intermediate sums for rounding error to hide inside.

It's Not Always Catastrophic — But It's Never Free

The size of the failure depends on how large the data's offset is relative to its own spread. Using the same 9-point idea but with a smaller offset (around `1,000,000` instead of `20,000,000`), the naive formula's relative error was verified at roughly `0.25%` — a real, measurable error, but not one that produces an outright impossible result. Push the offset larger still (around `10⁸`), and the naive formula's result collapses to **exactly** `0.0` — complete, 100% information loss, the same complete-collapse pattern Chapter 4 verified for `(1+x)-1` at small enough `x`.

DATA OFFSET RELATIVE TO SPREAD	NAIVE ONE-PASS RESULT	TWO-PASS RESULT
Moderate (~10^6)	~0.25% relative error — noticeably degraded, not catastrophic	~10^{-16} relative error — essentially exact
Large (~10^7–10^8)	Negative variance, or a complete collapse to exactly `0.0`	~10^{-16} relative error — unaffected by the data's offset

Numerical Stability, Defined

An algorithm is **numerically stable** if small rounding errors introduced at each individual step stay small in the final result, relative to the size of what's being computed. An algorithm is **numerically unstable** if those small per-step errors can be amplified into a large, disproportionate error in the output — exactly what the

naive variance formula does whenever the data's own scale is much larger than its spread. Crucially, stability is a property of *how the computation is organized*, not of the mathematical quantity being computed — both formulas compute "variance"; only one of them computes it *reliably*.

WHY THIS GENERALIZES CHAPTER 4, NOT JUST REPEATS IT

Chapter 4 diagnosed a single dangerous subtraction inside one formula. This chapter shows the same underlying mechanism can be buried inside a multi-step algorithm — summing, then subtracting at the very end — where it's far less visually obvious than `a - b` sitting directly in the code. Recognizing numerical instability increasingly means recognizing the *shape* of a computation (large intermediate values feeding into a final subtraction), not just spotting an explicit minus sign.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Stability is a property of the algorithm, distinct from the problem itself	Sets up Chapter 6's condition number, which measures instability that comes from the *problem*, independent of which algorithm is used
The two-pass formula avoids ever subtracting large near-equal numbers	The same "center first, then combine" strategy reappears in Chapter 9's own numerical differentiation/integration techniques
Two mathematically identical formulas, one reliable and one not	Directly informs Chapter 8's treatment of Gaussian elimination, where multiple algebraically-equivalent elimination orders differ sharply in reliability

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own verified negative-variance result, why a negative output from the naive variance formula is proof of a numerical bug rather than just "a somewhat inaccurate but plausible" result.

View solution

EXERCISE 2

Using this chapter's own explanation, describe specifically why the two-pass variance formula never suffers the same cancellation problem as the naive formula, even though it still involves subtraction (`x - mean`) at every single data point.

View solution

EXERCISE 3

A junior developer argues: "the two formulas are mathematically the same, so it doesn't matter which one we use — pick whichever runs in one pass over the data for speed." Using this chapter's own findings, write a short, specific explanation of what's wrong with that reasoning.

View solution

CHAPTER 5 QUICK REFERENCE

- **Numerical stability** is a property of an algorithm's structure, not of the mathematical quantity it computes — two formulas can be algebraically identical and numerically very different
- Verified: the naive one-pass variance formula (`E[X²]-(E[X])²`) returned `-0.0625` — a mathematically impossible negative variance — on real data with a large offset; the two-pass formula matched a high-precision reference to `≈7×10⁻¹⁷` relative error on the same data
- The failure scales with the data's offset relative to its spread — verified: ~0.25% error at moderate offset, complete collapse to exactly `0.0` at a large enough offset
- The root cause is Chapter 4's own cancellation mechanism, hidden inside a multi-step sum rather than a single visible subtraction
- The fix pattern generalizes Chapter 4's: center or otherwise avoid combining large near-equal quantities before the final subtraction happens
- **Next chapter:** Error propagation & conditioning — separating instability caused by the algorithm from instability that's baked into the problem itself

CHAPTER 6 OF 10

Error Propagation & Conditioning

Numerical Methods & Floating-Point Computation

Chapter 6 · Error Propagation & Conditioning

Chapter 5 established that **stability** — how much a particular *algorithm* amplifies rounding error — is a real, distinct, fixable property. This chapter covers the other half of the picture: **conditioning** — how sensitive the underlying *mathematical problem itself* is to small changes in its input, completely independent of which algorithm is used to solve it. The distinction matters enormously in practice: a stability problem can be fixed by choosing a better algorithm; a conditioning problem often can't be.

The Condition Number, Defined

For a function `y = f(x)`, the **relative condition number** is `cond = |x · f'(x) / f(x)|` — the ratio between the *relative* change in the output and the *relative* change in the input, for a small perturbation. A condition number near `1` means input errors pass through roughly unchanged. A large condition number means the problem itself **amplifies** whatever error already exists in the input, regardless of how carefully the evaluation is carried out.

VERIFIED DIRECTLY — A WELL-CONDITIONED AND AN ILL-CONDITIONED FUNCTION, SIDE BY SIDE

For `f(x) = x²` at `x=2`: the formula predicts `cond = |2 · 4 / 4| = 2`. Verified by actually perturbing `x` by a relative `10⁻⁸` and measuring the resulting relative change in `f(x)`: the measured amplification is `2.00000001` — matching the formula almost exactly. For `f(x) = 1/(x−1)` at `x=1.001` (close to the function's singularity at `x=1`): the formula predicts `cond = |1.001/0.001| = 1001`. The same perturbation experiment measures an amplification of `≈1000.99` — again matching closely. The second function is genuinely, measurably **500 times** more sensitive to the exact same size of input error.

Reframing Cancellation: It Was Conditioning All Along

Chapters 4 and 5 diagnosed cancellation as an *algorithm* problem — a bad choice of arithmetic steps. The condition-number framework reveals something sharper: the operation `a − b` *itself* has its own condition number, `(|a| + |b|) / |a − b|`, which explodes whenever `a` and `b` are close.

VERIFIED DIRECTLY — SUBTRACTION AS AN ILL-CONDITIONED OPERATION

For `a = 1,000,000.1`, `b = 1,000,000.0` (so `a − b = 0.1`): the subtraction's own condition number is `(|a|+|b|)/|a−b| = 20,000,001`. Perturbing only `a` by a relative `10⁻¹²` and re-computing `a − b` **exactly** (via 50-

digit precision arithmetic, so no algorithm-level rounding error is involved at all) produces a relative output change of `10,000,001` times larger than the input perturbation — matching the theoretical `a/(a-b)` bound for a single-variable perturbation almost exactly.

RESOLVING AN APPARENT CONTRADICTION WITH CHAPTERS 4-5

This isn't a different phenomenon from cancellation — it's the same phenomenon, now with a precise name and number attached. But it clarifies something important: subtraction of near-equal numbers is ill-conditioned *as an operation*, yet Chapter 4's stable quadratic-formula reformulation still fixed it. The resolution is that a larger computation can often be **restructured to avoid ever performing that specific ill-conditioned subtraction at all** — routing the same overall calculation through a different, well-conditioned sequence of operations instead. Algorithm choice doesn't make a specific ill-conditioned operation less sensitive; it can choose *not to use that operation* when a better-conditioned path to the same answer exists.

When There's No Way Around It: A Genuinely Ill-Conditioned Problem

Sometimes there is no alternative algorithm to switch to, because the sensitivity is baked into the problem as stated. Consider solving the linear system:

```
x +      y = 2
x + 1.0001y = 2.0001
                  (true solution: x = 1, y = 1)
```

Geometrically, these are two nearly-parallel lines — their determinant (`1 × 1.0001 − 1 × 1 = 0.0001`) is tiny, meaning the lines intersect at a very shallow angle. A tiny shift in either line moves their intersection point a lot.

VERIFIED DIRECTLY — EXACT, ROUNDING-FREE ARITHMETIC STILL CAN'T SAVE THIS

Solving this system with Cramer's rule using 50-digit exact `Decimal` arithmetic — deliberately eliminating every possible source of algorithm-level rounding error — gives the correct answer, `x=1, y=1`. Now perturb just one coefficient, `1.0001 → 1.0001 + 10⁻¹⁰` (a relative change of only `10⁻¹⁰`, smaller than a typical floating-point rounding error) and solve again, still with the same exact 50-digit arithmetic: `x` shifts to `0.999998999899...` — a relative change of about `10⁻⁶`. That's an amplification of roughly **10,000×**, and it happened with **zero** algorithm-level rounding error anywhere in the computation. The entire distortion came from the problem's own sensitivity to its input.

THE REAL-WORLD IMPLICATION

This is what makes conditioning different from stability in practice: no amount of clever algorithm design fixes an ill-conditioned problem, because the sensitivity doesn't come from how the arithmetic is organized — it comes from the problem itself. If the coefficients `1` and `1.0001` in this system came from real-world measurements with any

uncertainty at all, *no algorithm, however perfectly implemented, could recover a trustworthy answer* — the honest response is to recognize the problem is ill-conditioned and either obtain more precise inputs or accept a wide uncertainty band on the answer, not to search for a better solver.

Stability vs. Conditioning, Side by Side

	STABILITY (CH.5)	CONDITIONING (THIS CHAPTER)
What it measures	How much rounding error a specific algorithm introduces and amplifies	How much a small input change moves the true, exact answer
Property of	The algorithm / method	The mathematical problem itself
Can it be fixed by switching algorithms?	Yes — Chapters 4-5's whole point	No — a different algorithm solves the same ill-conditioned problem just as badly
Worked example this chapter	(recap) naive variance formula	The near-singular 2-equation linear system

THE COMBINED PICTURE

A trustworthy numerical result needs **both**: a well-conditioned problem (or an honest acknowledgment that it isn't one) *and* a stable algorithm solving it. A stable algorithm applied to an ill-conditioned problem still gives an unreliable answer — and an unstable algorithm can make even a perfectly well-conditioned problem look unreliable. Chapters 7 and 8 apply exactly this combined lens to root-finding and Gaussian elimination.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
The condition number formula `|x f'(x)/f(x)|`	Chapter 7's Newton's method, whose own convergence behavior depends directly on the derivative near the root
Near-singular systems amplify input error regardless of algorithm	Chapter 8's ill-conditioned matrices, where the same near-parallel-lines geometry reappears at larger scale
Stability (fixable) vs. conditioning (often not)	The honest diagnostic framework Chapter 10's capstone audit applies to real code

Hands-On Exercises

EXERCISE 1

Using this chapter's own condition number formula `|x f'(x)/f(x)|`, compute the condition number of `f(x) = ln(x)` at `x=1.001` (where `f'(x) = 1/x`), and explain in your own words what a very large value would tell you about evaluating the logarithm near that point.

View solution

EXERCISE 2

A colleague says "Chapter 4 proved that using a better algorithm fixes catastrophic cancellation, but this chapter says subtraction of near-equal numbers is fundamentally ill-conditioned and can't be fixed — those two chapters contradict each other." Using this chapter's own resolution, explain why they don't actually contradict.

View solution

EXERCISE 3

Using this chapter's own verified linear-system example, explain why "just switch to a different, more sophisticated equation-solving algorithm" would not actually fix the problem, and describe what a genuinely honest response to this situation would look like instead.

View solution

CHAPTER 6 QUICK REFERENCE

- **Condition number:** `cond = |x f'(x)/f(x)|` — the ratio of relative output change to relative input change; a property of the *problem*, not the algorithm
- Verified: `f(x)=x²` at `x=2` has `cond≈2` (well-conditioned); `f(x)=1/(x-1)` near its singularity has `cond≈1001` (ill-conditioned) — measured amplification matched both predictions closely
- Subtraction's own condition number, `(|a|+|b|)/|a-b|`, explains Chapters 4-5's cancellation as ill-conditioning of that specific operation — not a contradiction, since a larger algorithm can often avoid routing through it
- Verified: a near-singular 2-equation linear system amplified a `10⁻¹⁰` relative input perturbation into a `≈10⁻⁶` relative output change (≈10,000×) — using exact, rounding-free 50-digit arithmetic, proving the sensitivity came from the problem, not any algorithm
- **Stability** (Ch.5) is fixable by choosing a better algorithm; **conditioning** (this chapter) generally is not — both are needed for a trustworthy result
- **Next chapter:** Root-finding methods — bisection and Newton's method, where conditioning near the root directly determines convergence behavior

CHAPTER 7 OF 10

Root-Finding Methods

Numerical Methods & Floating-Point Computation

Chapter 7 · Root-Finding Methods

Finding an `x` where `f(x) = 0` is one of the most common tasks in applied computing — and it's where this course's earlier chapters stop being abstract cautionary tales and start directly shaping how a real algorithm is built. This chapter covers three approaches with three genuinely different reliability profiles: **bisection** (always works, but slowly), **Newton's method** (usually fast, but can fail outright — reusing Calculus & Optimization's own derivative rules directly), and **fixed-point iteration**, the general framework both of the others turn out to be special cases of.

Bisection: Guaranteed, Slow, and Hard to Break

If `f(a)` and `f(b)` have opposite signs, the Intermediate Value Theorem guarantees a root exists somewhere between them. Bisection exploits this directly: check the midpoint's sign, keep whichever half still brackets the root, and repeat. Every single iteration is guaranteed to **halve** the interval containing the root — no more, no less.

VERIFIED DIRECTLY — BISECTION ON $f(x) = x^3 - 2x - 5$

Starting from the bracket `[2, 3]` (since `f(2)=-1` and `f(3)=16`), reaching a tolerance of `10⁻¹²` took exactly **39 iterations**, converging to `2.094551481542112` — matching the true root, `2.0945514815423265`, to the requested precision.

Newton's Method: Reusing Calculus & Optimization's Own Derivative Rules

Newton's method uses the derivative to take a far more informed step: approximate `f` near the current guess with its tangent line (exactly the derivative Calculus & Optimization Chapter 3 defined), and jump to where that tangent line crosses zero: `x_{n+1} = x_n - f(x_n)/f'(x_n)`.

VERIFIED DIRECTLY — THE SAME EQUATION, DRAMATICALLY FEWER STEPS

Starting from `x₀=2.0`, Newton's method reaches the same `10⁻¹²` tolerance in just **4 iterations**: `2.0 → 2.1 → 2.094568121... → 2.094551481698... → 2.0945514815423265` — the last two steps alone gained roughly 6 and then 10+ correct digits. This is **quadratic convergence**: the number of correct digits roughly *doubles* every step, versus bisection's fixed, one-bit-per-step linear rate. 39 iterations vs. 4, for the identical equation and tolerance.

Two Genuine Ways Newton's Method Fails

Newton's speed comes at a real cost: unlike bisection, it has no built-in guarantee. Two distinct, verified failure modes:

Failure 1 — A Permanent Cycle

VERIFIED DIRECTLY — NEWTON'S METHOD THAT NEVER CONVERGES AT ALL

For `f(x) = x³ − 2x + 2` starting at `x₀=0`: the iteration produces `0 → 1 → 0 → 1 → 0 → 1 → ...` — an exact, permanent 2-cycle that never approaches the function's real root (near `−1.77`). This isn't slow convergence; it's **no convergence at all**, from a starting point that looks entirely reasonable.

Failure 2 — Degraded Convergence at a Double Root

Newton's method divides by `f'(x)` at every step — which is exactly the kind of division Chapter 6 flagged as dangerous when the denominator is close to zero. At a **double root** (where both `f(x)=0` *and* `f'(x)=0` at the same point), that's precisely what happens as the iteration converges.

VERIFIED DIRECTLY — QUADRATIC CONVERGENCE DEGRADES TO EXACTLY LINEAR

For `f(x) = (x−1)²` (a double root at `x=1`) starting at `x₀=5`: the error shrinks as `4.0 → 2.0 → 1.0 → 0.5 → 0.25 → ...` — the ratio between successive errors is **exactly `0.5`, every single step**. That's linear convergence, at precisely bisection's own rate — Newton's usual quadratic speed advantage disappears completely, because `f'(x) → 0` right along with `f(x) → 0`, making the division step increasingly ill-conditioned exactly as it approaches the root.

Fixed-Point Iteration: The General Framework Underneath Both

A **fixed-point iteration** repeatedly applies `x_{n+1} = g(x)`, hoping the sequence settles at a point where `x = g(x)`. The convergence rule is precise: the iteration converges near a fixed point if `|g'(x)| < 1` there, and diverges if `|g'(x)| > 1` — a direct, exact echo of Chapter 6's condition-number logic, now applied to a repeated process instead of a single evaluation.

VERIFIED DIRECTLY — THREE REARRANGEMENTS OF $x^2 = 2$, THREE COMPLETELY DIFFERENT OUTCOMES

Solving for `√2 ≈ 1.4142135623730951` using three algebraically valid rearrangements of the same equation, all starting from `x₀=1.4`:

ITERATION `G(x)`	`|G'(√2)|`	VERIFIED BEHAVIOR
`g(x) = 2/x`	`≈1.0` (marginal)	Never converges — locks into a permanent 2-cycle: `1.4 → 1.42857... → 1.4 → 1.42857... → ...`, forever

ITERATION `G(X)`	`|G'(√2)|`	VERIFIED BEHAVIOR
`g(x) = (x + 2/x)/2`	`≈0.0`	Converges to `1.414213562373095` in just **4 iterations** — and is, in fact, exactly Newton's method applied to `x²−2=0`
`g(x) = x² + x − 2`	`≈3.83` (>1)	Diverges immediately and dramatically: `1.4 → 1.36 → 1.21 → 0.67 → −0.87 → −2.11 → ...`, never settling anywhere near `√2`

WHY NEWTON'S METHOD IS REALLY A SPECIAL CASE

Newton's method *is* a fixed-point iteration, with `g(x) = x − f(x)/f'(x)`. This isn't a coincidence — it's exactly why the second rearrangement above, `(x+2/x)/2`, converges so fast: it's what you get from applying Newton's own formula to `f(x)=x²−2`. The general fixed-point convergence rule, `|g'(x)|<1`, is the same underlying idea as Newton's quadratic convergence — Newton's method is simply a particularly clever choice of `g` that drives `g'(root)` all the way down to `0` for a well-behaved (non-double) root, which is exactly why it usually beats plain bisection so decisively.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
Newton's method divides by `f'(x)`, degrading near a double root	Direct application of Chapter 6's conditioning — a near-zero denominator is exactly the ill-conditioned-division pattern already established
Fixed-point convergence rule `|g'(x)|<1`	The same contraction-mapping logic reappears in Chapter 8's iterative linear-system solvers, applied to a vector instead of a single number
A permanent 2-cycle from a "reasonable" starting point	A concrete warning Chapter 10's capstone audit checks for directly — Newton's method needs a real convergence safeguard in production code, not blind trust

Hands-On Exercises

EXERCISE 1

Using this chapter's own verified iteration counts (39 for bisection, 4 for Newton's method, on the same equation and tolerance), explain in your own words why "always use Newton's method, it's faster" would still be bad advice for a general-purpose root finder, using this chapter's own two verified Newton failure modes.

View solution

EXERCISE 2

Using this chapter's own connection between Newton's method and Chapter 6's conditioning material, explain specifically why a double root (where `f(x)=0` and `f'(x)=0` at the same point) causes Newton's method to slow down to linear convergence rather than simply failing outright the way division by exactly zero would.

View solution

EXERCISE 3

Using this chapter's own verified `g(x)=2/x` example (which locks into a permanent 2-cycle rather than diverging to infinity or slowly converging), explain what a condition number very close to exactly `1` — as opposed to clearly less than `1` or clearly greater than `1` — predicts about a fixed-point iteration's behavior, and why that's a genuinely different outcome from both convergence and divergence.

View solution

CHAPTER 7 QUICK REFERENCE

- **Bisection:** guaranteed to converge given a sign change, but slow (linear, one bit of precision per step) — verified: 39 iterations to `10⁻¹²`
- **Newton's method:** `x_{n+1}=x_n-f(x_n)/f'(x_n)`, usually much faster (quadratic — digits roughly double each step) but not guaranteed — verified: 4 iterations to the same tolerance, on the same equation
- Verified Newton failure 1: a permanent 2-cycle (`f(x)=x³-2x+2`, `x₀=0`) that never converges at all
- Verified Newton failure 2: degrades to exactly linear convergence (error ratio `=0.5` every step) at a double root, since `f'(x)→0` makes the division step ill-conditioned — a direct application of Chapter 6
- **Fixed-point iteration** `x_{n+1}=g(x)` converges near a fixed point iff `|g'(x)|<1` — verified with three rearrangements of the same equation: one converges in 4 steps, one locks into a 2-cycle, one diverges outright
- Newton's method is itself a fixed-point iteration, `g(x)=x-f(x)/f'(x)` — the two methods aren't separate ideas, just different choices of `g`
- **Next chapter:** Numerical linear algebra pitfalls — the same conditioning and stability ideas, applied to solving systems of equations rather than a single-variable root

CHAPTER 8 OF 10

Numerical Linear Algebra Pitfalls

Numerical Methods & Floating-Point Computation

Chapter 8 · Numerical Linear Algebra Pitfalls

Linear Algebra Fundamentals covered Gaussian elimination as an exact, always-correct algebraic procedure. In floating point, it is not always safe to run exactly as written — the *order* in which rows are eliminated can make the difference between an accurate answer and a badly wrong one, on the very same system of equations.

Plain Gaussian Elimination Can Fail — Even Without a Genuinely Ill-Conditioned System

Solve this ordinary-looking 2×2 system:

```
1e-16 x +   y = 1
    1 x +   y = 2
          (true solution: x ≈ 1.0, y ≈ 1.0)
```

Standard Gaussian elimination uses the first row's leading entry as the **pivot** to eliminate `x` from the second row. Here, that pivot is `1 × 10⁻¹⁶` — tiny, but not literally zero, so nothing raises an error. The elimination step computes a **multiplier** `m = 1 / 10⁻¹⁶ = 10¹⁶` and uses it to combine the rows.

VERIFIED DIRECTLY — ONE VARIABLE COMES OUT PERFECT, THE OTHER COMES OUT COMPLETELY WRONG

Solving this system with plain, no-pivoting Gaussian elimination in standard double precision: `y` comes out as `0.9999999999999998` — essentially exact, relative error `≈1.2 × 10⁻¹⁶`. But `x` comes out as `2.220446049250313` — a relative error of `≈1.22`, meaning the computed answer is more than **double** the true value of `1.0`. Same system, same elimination procedure, one variable essentially perfect and the other one completely unusable.

WHY: CHAPTER 4'S CANCELLATION, ARRIVING THROUGH BACK-SUBSTITUTION

Back-substitution recovers `x` as `(1 − y) / 10⁻¹⁶`. `y` itself carries a tiny rounding error (as verified, extremely small on its own) — but subtracting it from `1` triggers exactly Chapter 4's cancellation pattern, and then **dividing that already-damaged result by the tiny original pivot** multiplies whatever error survived the subtraction by `10¹⁶`. The huge multiplier used during elimination is what makes this catastrophic: it's the same "divide by

something close to zero" danger Chapter 6 and Chapter 7 (Newton's method near a double root) already flagged, now appearing inside a multi-step linear-system solve.

The Fix: Partial Pivoting

Partial pivoting is a simple, mechanical rule: before eliminating a column, swap rows so that the row with the **largest available magnitude** in that column becomes the pivot row. It doesn't change the mathematical system being solved — just the order the rows are processed in.

VERIFIED DIRECTLY — THE EXACT SAME SYSTEM, NOW SOLVED CORRECTLY

Swapping the two rows before eliminating (so the pivot is `1` instead of `10⁻¹⁶`, giving a multiplier of just `10⁻¹⁶` instead of `10¹⁶`): `x` now comes out as **exactly** `1.0` (relative error `≈1 × 10⁻¹⁶`) and `y` stays just as accurate as before. Nothing about the underlying equations changed — only the order of elimination — and the catastrophic failure disappears completely.

WHY THIS IS A GENUINELY STABLE FIX, NOT A LUCKY PATCH

Partial pivoting guarantees every multiplier used during elimination has magnitude `≤1` — by construction, since the pivot is always at least as large as everything below it in that column. A multiplier that never exceeds `1` can never amplify rounding error the way this chapter's `10¹⁶` multiplier did, which is exactly why partial pivoting is the industry-standard default in virtually every real linear algebra library, not an optional refinement.

Ill-Conditioned Matrices, Revisited With Real Numbers

Partial pivoting fixes an *unstable algorithm* (Chapter 5's territory) — but Chapter 6 already showed some systems are *ill-conditioned problems*, which no algorithm can fix. The **matrix condition number**, `cond(A) = ‖A‖ · ‖A⁻¹‖`, puts a precise number on this, computed directly using Linear Algebra Fundamentals' own determinant and inverse formulas.

VERIFIED DIRECTLY — QUANTIFYING CHAPTER 6'S OWN NEAR-SINGULAR SYSTEM

For the near-singular matrix from Chapter 6's linear-system example, `A = [[1,1],[1,1.0001]]`: `det(A) = 0.0001`, and by the standard 2×2 inverse formula, `A⁻¹ = [[10001, -10000], [-10000, 10000]]`. Using the largest-row-sum matrix norm: `‖A‖ = 2.0001` and `‖A⁻¹‖ = 20001`, giving **`cond(A) = 40,004.0001`** — directly explaining the roughly 10,000× amplification Chapter 6 measured empirically for that exact system (the condition number is an upper bound on amplification across any perturbation direction; Chapter 6's experiment perturbed only one entry, so it measured somewhat below this worst-case bound). For contrast, a well-behaved diagonal matrix `B = [[2,0],[0,3]]` gives `cond(B) = 1.5` — close to the best-possible value of `1`.

CRUCIALLY: PIVOTING CANNOT FIX THIS

Partial pivoting was the correct, complete fix for this chapter's 10^{-16}-pivot example, because that system was well-conditioned — its true solution isn't especially sensitive to small input changes; only the naive elimination *order* was the problem. The near-singular matrix above is a genuinely different situation: `cond(A)=40,004` means the *problem itself* amplifies input error by that much, regardless of pivoting, regardless of algorithm choice — exactly Chapter 6's own conclusion, now expressed as a single computable number for an entire matrix instead of one subtraction.

Two Genuinely Different Diagnoses, Side by Side

	THIS CHAPTER'S 10^{-16}-PIVOT SYSTEM	CHAPTER 6'S NEAR-SINGULAR SYSTEM
What was actually wrong	Elimination order (small pivot, huge multiplier)	The matrix's own geometry (near-parallel rows)
Diagnosis	Algorithm instability (Ch.5)	Problem ill-conditioning (Ch.6)
Fixed by partial pivoting?	Yes — verified, error dropped from `1.22` to `≈10⁻¹⁶`	No — pivoting reorders rows, it doesn't change `cond(A)`
The real fix, if one exists	Use a stable algorithm (done)	Better input precision, or accept/report the uncertainty (Ch.6's own conclusion)

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
A huge elimination multiplier amplifies rounding error via Chapter 4's cancellation	Direct extension of Chapter 4-5's single-operation cancellation into a full multi-step linear solve
`cond(A)` computed via Linear Algebra Fundamentals' own determinant/inverse formulas	Fully quantifies Chapter 6's near-singular example, closing a loop that chapter left as "roughly 10,000×"
Stability (fixable) vs. conditioning (not fixable) applied to a real matrix	Chapter 10's capstone audit checks for both patterns directly in real code

Hands-On Exercises

EXERCISE 1

Using this chapter's own verified 10^{-16}-pivot example, explain specifically why `y` came out essentially perfect while `x` came out more than 100% wrong, from the exact same elimination process on the exact same system.

View solution

EXERCISE 2

Explain, using this chapter's own reasoning about multiplier size, why partial pivoting's rule (always use the largest-magnitude available entry as the pivot) guarantees every multiplier has magnitude no greater than `1`.

View solution

EXERCISE 3

A developer, having read this chapter's first example, concludes "always use partial pivoting, and every linear system will then be solved reliably." Using this chapter's own comparison between the two worked examples, explain what's wrong with that conclusion.

View solution

CHAPTER 8 QUICK REFERENCE

- Plain Gaussian elimination with a tiny pivot uses a huge multiplier, which amplifies rounding error via Chapter 4's cancellation mechanism during back-substitution — verified: a 10^{-16} pivot produced a `122%` relative error in one variable, while the other stayed accurate
- **Partial pivoting** — always eliminate using the largest-magnitude available entry as the pivot — bounds every multiplier to magnitude ≤ 1, fully fixing this specific failure: verified error dropped to $\approx 10^{-16}$
- The **matrix condition number** $\text{cond}(A) = \|A\| \cdot \|A^{-1}\|$, computed via Linear Algebra Fundamentals' own inverse formula, quantifies Chapter 6's near-singular system precisely: $\text{cond}(A) \approx 40{,}004$, vs. $\text{cond}(B) = 1.5$ for a well-conditioned matrix
- Pivoting fixes algorithm instability (Ch.5) but **cannot** fix problem ill-conditioning (Ch.6) — the two failures verified in this chapter needed genuinely different diagnoses and genuinely different fixes
- **Next chapter:** Numerical differentiation & integration, revisited — a deeper pass through territory Calculus & Optimization only introduced

CHAPTER 9 OF 10

Numerical Differentiation & Integration, Revisited

Numerical Methods & Floating-Point Computation

Chapter 9 · Numerical Differentiation & Integration, Revisited

Calculus & Optimization Chapters 4 and 9 introduced numerical differentiation and integration as basic techniques and this course's own Chapter 4 explained why the simplest version breaks down. This chapter goes further: three genuinely better techniques for differentiation, and one genuinely better strategy for integration — each directly informed by what Chapters 1-8 already established about cancellation, stability, and conditioning.

Central Differences: A Free Upgrade Over Forward Differences

The forward-difference formula, `(f(x+h)-f(x))/h`, uses one point ahead of `x`. The **central-difference** formula uses one point on each side: `(f(x+h) - f(x-h)) / (2h)`. Both approximate the same derivative, but their error behaves very differently as `h` shrinks — forward error shrinks proportionally to `h` itself, while central error shrinks proportionally to `h²`.

VERIFIED DIRECTLY — FOR F(X)=SIN(X) AT X=1 (TRUE DERIVATIVE COS(1)≈0.5403023058681398)

At `h=10⁻⁴`: forward error `≈4.21×10⁻⁵`, central error `≈9.00×10⁻¹⁰` — central is already **almost five orders of magnitude** more accurate for the same step size and the same number of extra function evaluations. Both formulas eventually hit the same cancellation wall this course's Chapter 4 already explained: at `h=10⁻¹⁶`, forward error balloons to `0.54` (completely wrong) while central error reaches `0.0148` — degraded, but noticeably more resistant to the collapse than forward differences, since central differencing's symmetric structure cancels out more of the leading error terms before cancellation error takes over.

Complex-Step Differentiation: Sidestepping Cancellation Entirely

Both formulas above eventually fail because they *subtract* two nearly-equal real numbers. **Complex-step differentiation** is a genuinely different trick: evaluate the function at a small *imaginary* step, `f(x + ih)`, and take the imaginary part: `f'(x) ≈ Im(f(x+ih)) / h`. For a function that's analytic (most ordinary functions — polynomials, `sin`, `exp`, etc. all qualify), this formula has essentially no cancellation error, because it never subtracts two real, nearly-equal quantities at all — the real and imaginary parts of a complex number are stored and manipulated independently.

```
import cmath
derivative = cmath.sin(x + 1j*h).imag / h
```

VERIFIED DIRECTLY — ACCURATE TO THE EXACT SAME DOUBLE-PRECISION VALUE ALL THE WAY DOWN TO H=10⁻¹⁰⁰

Computing `f'(1)` for `f(x)=sin(x)` via complex-step differentiation at `h=10⁻⁸`, `10⁻¹⁶`, `10⁻²⁰`, `10⁻³⁰`, and even `10⁻¹⁰⁰`: **every single one** returned exactly `0.5403023058681398` — the true value to full double precision, with a measured error of `0.000` at every tested `h`. There is no "too small" step size for this method, unlike forward or central differences, both of which collapsed catastrophically once `h` got small enough.

WHY THIS ISN'T A FREE LUNCH IN GENERAL

Complex-step differentiation requires the function to support complex-number arithmetic internally (which ordinary `sin`, `cos`, `exp`, and polynomial code usually does, but a function containing `abs()`, comparisons, or other non-analytic operations generally doesn't handle correctly), and it only computes first derivatives cleanly — it's a specialized tool for a specific situation, not a universal replacement for finite differences.

Richardson Extrapolation: Cancel the Leading Error Term

Central differences have error that behaves predictably: `D(h) = f'(x) + C·h² + O(h⁴)` for some constant `C`. **Richardson extrapolation** exploits this directly — compute the same central-difference estimate at two step sizes, `h` and `h/2`, and combine them to cancel out the `h²` term algebraically: `R = (4·D(h/2) − D(h)) / 3`. What's left over is `O(h⁴)` — a dramatically better estimate, built entirely from two ordinary central-difference calculations already in hand.

VERIFIED DIRECTLY — FOUR ORDERS OF MAGNITUDE OF ACCURACY, FOR FREE

At `h=0.1`: `D(h)` has error `9.00×10⁻⁴`, `D(h/2)` has error `2.25×10⁻⁴` — but the Richardson combination has error just `1.13×10⁻⁷`, nearly **four orders of magnitude** better than either individual estimate. At `h=0.001`, the Richardson result's error is `3.5×10⁻¹⁴` — right at the edge of what double precision can represent at all.

Adaptive Quadrature: Spending Effort Where It's Needed

Calculus & Optimization Chapter 9 covered fixed-grid numerical integration — evaluating the function at evenly-spaced points across the whole interval, regardless of how the function actually behaves. **Adaptive quadrature** instead estimates the local error on each subinterval and only subdivides further where that error is too large — concentrating function evaluations where the function is actually changing quickly, and spending almost none where it's nearly flat.

VERIFIED DIRECTLY — HALF THE FUNCTION EVALUATIONS, FOR COMPARABLE ACCURACY

Integrating `f(x) = 1/(1 + 10000(x-0.5)²)` over `[0,1]` — a function that's nearly zero almost everywhere except for a sharp, narrow spike right at `x=0.5` — against the exact value `0.030105979856434922`: a **fixed-grid** Simpson's rule needs `n=500` (501 function evaluations) to reach an error of `≈3.16×10⁻⁹`. **Adaptive** Simpson's rule reaches a comparable error, `≈3.53×10⁻⁹`, using only **265 function evaluations** — roughly **half as many**, because most of the fixed grid's points were wasted evaluating the function where it's already almost exactly zero.

WHY THIS MATTERS MORE AS FUNCTIONS GET MORE EXPENSIVE

The saving here (roughly 2×) is on a function that costs almost nothing to evaluate. For a real-world integrand that might take milliseconds or seconds per evaluation — a physics simulation step, a Monte Carlo sample, a call to an external service — the difference between 265 and 501 evaluations is the difference between a computation finishing twice as fast, for the same accuracy, with no extra implementation cost once the adaptive logic is written once.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT RESOLVES OR SETS UP
Complex-step differentiation avoids cancellation entirely	A genuinely different resolution to Chapter 4's cancellation problem than the "restructure the algebra" fix used for the quadratic formula
Richardson extrapolation cancels the leading error term algebraically	The same pattern underlies higher-order Runge-Kutta methods and other advanced numerical techniques beyond this course's scope
Adaptive methods concentrate effort where local error is largest	Directly informs how Chapter 10's capstone audit should treat any fixed-step-size code it finds — a specific, checkable red flag

Hands-On Exercises

EXERCISE 1

Using this chapter's own verified numbers, explain why central differences beat forward differences by nearly five orders of magnitude at `h=10⁻⁴`, but both still eventually collapse at very small `h` — what does central differencing fix, and what does it not fix?

View solution

EXERCISE 2

Using this chapter's own verified complex-step results, explain specifically why the formula `Im(f(x+ih))/h` has no cancellation error, even though it still involves a division by a small `h` — your answer should identify what operation is genuinely missing compared to the real-valued finite-difference formulas.

View solution

EXERCISE 3

A colleague argues that adaptive quadrature is strictly better than fixed-grid integration and should always be used. Using this chapter's own verified example and its own tip box about evaluation cost, describe a realistic situation where the extra implementation complexity of adaptive quadrature might not be worth it.

View solution

CHAPTER 9 QUICK REFERENCE

- **Central difference** `(f(x+h)-f(x-h))/(2h)`: `O(h²)` error, verified nearly 5 orders of magnitude more accurate than forward differences at `h=10⁻⁴` — but still eventually collapses from cancellation at extreme small `h`
- **Complex-step differentiation** `Im(f(x+ih))/h`: verified exact to full double precision at every tested `h` down to `10⁻¹⁰⁰` — completely sidesteps cancellation by never subtracting two real numbers, at the cost of requiring an analytic, complex-compatible function
- **Richardson extrapolation** `(4D(h/2)-D(h))/3`: verified nearly 4 orders of magnitude accuracy improvement over either individual central-difference estimate, using calculations already computed
- **Adaptive quadrature**: verified roughly half the function evaluations of a fixed grid for comparable accuracy, by concentrating evaluations where a function's local behavior actually demands them
- **Next chapter:** Capstone — auditing a real, ordinary-looking codebase for exactly these nine chapters' worth of numerical bugs

CHAPTER 10 OF 10

Capstone — Diagnosing and Fixing Numerical Bugs in Real Code

Numerical Methods & Floating-Point Computation

Chapter 10 · Capstone — Diagnosing and Fixing Numerical Bugs in Real Code

One continuous audit: a small, ordinary-looking industrial sensor-monitoring codebase, module by module, each one carrying a real floating-point bug of exactly the kind this course has spent nine chapters teaching how to recognize. Every diagnosis below reuses this course's own already-verified numbers directly — nothing here is re-derived from scratch, because the whole point of a real audit is applying knowledge you already trust, not re-proving it each time.

STEP	MODULE AUDITED	BUG TYPE	CHAPTER(S) USED
1	Calibration validity check	Exact-equality comparison	Ch.1-2
2	Daily totals aggregator	Rounding bias	Ch.3
3	Drift-rate calculator	Catastrophic cancellation	Ch.4, Ch.9
4	Sensor variance monitor	Unstable variance formula	Ch.5
5	Dual-sensor calibration system	Ill-conditioned matrix	Ch.6, Ch.8
6	Set-point solver	Newton's method cycling	Ch.7
7	Two-sensor blend solver	Unpivoted elimination	Ch.8
8	Trajectory integrator	Wasteful fixed-step integration	Ch.9

Step 1 — The Calibration Validity Check

CH.1-2

```
# original: silently fails to flag miscalibrated sensors
def is_calibrated(reading_a, reading_b):
    return (reading_a + reading_b) == 0.3  # reading_a=0.1, reading_b=0.2 -> False!
```

Two sensors are supposed to sum to a known reference value of `0.3`. The check almost never passes, even for correctly calibrated sensors.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 1'S OWN VERIFIED FINDING

`0.1 + 0.2` evaluates to `0.30000000000000004`, which differs from the literal `0.3` (stored as `0.29999999999999998...`) by `≈5.55×10⁻¹⁷` — exactly the gap Chapter 1 verified. The exact-equality check was never going to reliably pass, regardless of how correctly the sensors were actually calibrated.

```
# fixed: tolerance-based comparison
def is_calibrated(reading_a, reading_b, tol=1e-9):
    return abs((reading_a + reading_b) - 0.3) < tol
```

Step 2 — The Daily Totals Aggregator

CH.3

Thousands of individual readings are rounded to the nearest whole unit before being summed into a daily report, using each language's default "round half up" behavior.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 3'S OWN VERIFIED 1,000-TIE EXPERIMENT

Chapter 3 verified that rounding 1,000 values landing exactly on a `.5` boundary with round-half-up produces a systematic `+500` bias versus the true total, while round-half-to-even produces **zero** bias. The aggregator has exactly this shape: any sensor reading that happens to land on an exact half-unit tie gets rounded the same direction every time, so the daily total drifts upward in proportion to how many exact ties occur — a real, compounding, one-directional error, not random noise that averages out.

Fix: switch the rounding mode to round-half-to-even (IEEE 754's own default, and Python's built-in `round()`) rather than a manually-implemented round-half-up.

Step 3 — The Drift-Rate Calculator

CH.4, CH.9

Copy

original: numerically differentiates a temperature sensor's own reading def drift_rate(temperature_fn, t, h=1e-14): return (temperature_fn(t+h) - temperature_fn(t)) / h # h chosen "for accuracy"

A well-meaning engineer picked an extremely small `h`, reasoning "smaller step, more accurate derivative."

DIAGNOSIS — DIRECTLY REUSING CHAPTER 4'S OWN RESOLVED EXAMPLE

Chapter 4 verified exactly this mistake using $f(x)=x^2$ at $x=3$: at $h=10^{-13}$, the relative error in the derivative reaches $\approx 6.8\times10^{-4}$, and by $h=10^{-16}$ the approximation collapses to 0 entirely, for a true derivative of 6. The engineer's instinct — smaller h is always better — is precisely the mistake this course spent Chapter 4 resolving: shrinking h drives `temperature_fn(t+h)` and `temperature_fn(t)` together, triggering catastrophic cancellation in the numerator.

Fix: switch to central differencing at a moderate step size (Chapter 9 verified central differences beat forward differences by nearly five orders of magnitude at $h=10^{-4}$), or — since the sensor's own temperature function is a simple, analytic calculation internally — use complex-step differentiation, verified exact to full double precision at any step size down to 10^{-100}, eliminating the tuning problem entirely.

Step 4 — The Sensor Variance Monitor

CH.5

Copy

```
# original: flags "impossible" negative variance readings as a hardware fault def sensor_variance(readings): n = len(readings) mean_sq = sum(x*x for x in readings) / n mean = sum(readings) / n return mean_sq - mean**2 # one-pass "efficient" formula
```

A pressure sensor with a large fixed baseline reading (around $20{,}000{,}000$ units) plus small genuine fluctuation triggers repeated false hardware-fault alerts.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 5'S OWN VERIFIED NEGATIVE-VARIANCE RESULT

Chapter 5 verified this exact one-pass formula returns -0.0625 — a mathematically impossible negative variance — on real data with a large baseline offset, against a true variance of ≈ 0.0673. The sensor's own large baseline is triggering the identical bug: $E[X^2]$ and $(E[X])^2$ are both enormous, and their difference (the true, tiny variance) is swamped by the rounding error accumulated while computing those two huge sums.

Fix: switch to the two-pass formula (center the data around its own mean first, then square) — Chapter 5 verified this matches a high-precision reference to $\approx 7\times10^{-17}$ relative error on the identical data.

Step 5 — The Dual-Sensor Calibration System

CH.6, CH.8

Two redundant pressure sensors are calibrated together by solving a small linear system relating their readings. The computed calibration constants vary wildly between runs, even on nearly identical input data.

DIAGNOSIS — THIS ONE ISN'T A CODING BUG AT ALL

The calibration system is algebraically identical to Chapter 6 and Chapter 8's own near-singular example, `[[1,1], [1,1.0001]]`, with a verified matrix condition number of `≈40,004`. The two sensors are giving **almost redundant** information (nearly parallel equations), and Chapter 6 proved directly — using exact, rounding-free 50-digit arithmetic — that no algorithm can fix this: a 10^{-10} relative input perturbation still produced a $\approx 10^{-6}$ relative output change, a genuine property of the sensor pair's own geometry, not of any solver.

Fix: this is a hardware/sensor-placement issue, not a software one — per Chapter 6's own honest conclusion, the correct response is to compute and report the condition number directly, flag the sensor pair as too redundant to calibrate reliably, and recommend physically repositioning one sensor rather than continuing to search for a better solver.

Step 6 — The Set-Point Solver

CH.7

Copy

original: hangs indefinitely for certain valve configurations def find_setpoint(f, fprime, x0, tol=1e-10): x = x0 while abs(f(x)) > tol: # no iteration limit -- can loop forever x = x - f(x)/fprime(x) return x

For one particular valve's calibration curve, $f(x)=x^3-2x+2$, starting from a "reasonable" default guess of $x_0=0$, the solver never returns.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 7'S OWN VERIFIED CYCLE

Chapter 7 verified this exact function and starting point produces a permanent `0 → 1 → 0 → 1 → ...` cycle under Newton's method — not slow convergence, but no convergence at all, from a starting point that looked entirely reasonable.

Fix: add an iteration cap and a bracketing fallback — if Newton's method hasn't converged within a set number of steps, fall back to bisection (Chapter 7's own guaranteed-but-slower alternative) using a bracket confirmed by a sign change, rather than looping indefinitely.

Step 7 — The Two-Sensor Blend Solver

CH.8

A separate calibration routine blends two sensor readings by solving a small linear system via plain Gaussian elimination, no row swapping. For one particular pair of sensitivity coefficients, one of the two blended outputs comes back wildly wrong.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 8'S OWN VERIFIED FAILURE

The coefficients happen to include a near-zero leading entry, exactly Chapter 8's own 10^{-16}-pivot example. Verified there: one output variable came back with a relative error of ≈ 1.22 (more than 100% wrong) while the other stayed accurate to $\approx 10^{-16}$ — the huge elimination multiplier (10^{16}) amplified rounding error straight through Chapter 4's own cancellation mechanism during back-substitution.

Fix: add partial pivoting to the elimination routine. Chapter 8 verified this alone drops the error from ≈ 1.22 down to $\approx 10^{-16}$ — the exact same system, correctly solved, just by changing the row processing order.

Step 8 — The Trajectory Integrator

CH.9

A separate module numerically integrates sensor-derived acceleration data into a trajectory estimate using a fixed, uniformly-spaced grid — chosen once, years ago, and never revisited. Most flight segments are smooth, but a small number involve a sharp maneuver the fixed grid consistently under-resolves.

DIAGNOSIS — DIRECTLY REUSING CHAPTER 9'S OWN VERIFIED COMPARISON

Chapter 9 verified that on a function with a sharp, localized feature, a fixed grid needs 501 function evaluations to reach an error of $\approx 3.16 \times 10^{-9}$, while adaptive quadrature reaches comparable accuracy ($\approx 3.53 \times 10^{-9}$) using only 265 evaluations — roughly half — because most of the fixed grid's points are wasted on the smooth stretches while the sharp maneuver is exactly where more resolution is actually needed.

Fix: replace the fixed-step integrator with adaptive quadrature. Sharp maneuvers automatically get more evaluation points where they're needed; smooth stretches automatically get fewer — better accuracy on the segments that matter, at lower total computational cost.

Audit Summary

STEP	ROOT CAUSE	FIXABLE BY BETTER CODE?
1	Exact equality on a value that was never guaranteed to be exact	Yes — tolerance comparison
2	Directionally-biased rounding compounding over many operations	Yes — round-to-even
3	Cancellation from an over-aggressively small step size	Yes — central or complex-step differencing
4	An algebraically-unstable one-pass formula	Yes — two-pass formula
5	An inherently ill-conditioned sensor pair	**No** — hardware issue, honestly reported
6	An unguarded iterative method with no fallback	Yes — iteration cap + bisection fallback
7	An unstable elimination order	Yes — partial pivoting
8	A fixed grid wasting effort on well-behaved regions	Yes — adaptive quadrature

THE ONE FINDING THAT MATTERED MOST

Seven of these eight bugs were genuinely fixable by better code — recognizing them was the entire skill this course taught. But Step 5 is the one that separates a numerically literate engineer from one who merely knows some fixes: recognizing when a problem is **not** a code bug at all, and that the honest, correct response is to measure and report the condition number rather than keep tuning a solver that was never going to work.

What This Course Doesn't Cover

As stated honestly back in Chapter 1: symbolic/exact computer algebra, GPU-specific and parallel floating-point quirks (reduced precision, non-associative summation order, fused multiply-add), and arbitrary-precision/interval arithmetic libraries were all named as deliberately out of scope, and stayed out of scope through all ten chapters. Every bug in this capstone was diagnosed and fixed using only standard double-precision arithmetic — the default nearly every language reaches for first, and exactly the territory this course committed to from the start.

Where This Course Connects

Calculus & Optimization's own Chapter 1 forward-referenced the exact cancellation mechanism resolved in this course's Chapter 4, and its own numerical differentiation/integration (Chapters 4 and 9) were the introductory versions of what this course's Chapter 9 covered in real depth. Linear Algebra Fundamentals' determinant and inverse formulas were used directly to compute a real matrix condition number in Chapter 8. Algorithms & Complexity's own iterative-algorithm framing underlies both Chapter 7's root-finding and Chapter 9's adaptive quadrature. Within Technical Support, `perfdiag1`'s own performance-diagnosis discipline and `appdiag1`'s own root-cause reasoning are the same "measure, don't guess" instinct this capstone applied to numerical bugs specifically.

Hands-On Exercises

EXERCISE 1

A colleague proposes fixing Step 5's ill-conditioned dual-sensor system by "just using higher-precision floating point (128-bit) instead of standard doubles." Using this chapter's own Step 5 diagnosis and Chapter 6's original reasoning, explain whether this would actually fix the problem.

View solution

EXERCISE 2

Using this chapter's own Audit Summary table, group the eight bugs into two categories: those caused by an algorithm choice (fixable by using a different, better algorithm for the exact same problem) and those caused by the problem's own inherent sensitivity (not fixable by algorithm choice alone). Justify each grouping using this course's own Chapter 5/6 distinction.

View solution

EXERCISE 3

Step 6's fix adds an iteration cap and a bisection fallback to the set-point solver, rather than simply switching entirely from Newton's method to bisection. Using this chapter's own Step 6 diagnosis and Chapter 7's original comparison of the two methods, explain why keeping Newton's method as the primary approach (with bisection only as a fallback) is a better design than replacing it outright.

View solution

CHAPTER 10 QUICK REFERENCE

- **Full worked audit:** eight modules, eight bugs, each diagnosed by directly reusing an already-verified finding from Chapters 1 through 9 rather than re-deriving anything from scratch
- Seven of the eight bugs were genuine algorithm/code issues, fixable with a better formula, a tolerance check, pivoting, or an adaptive method
- One bug (Step 5) was not a code bug at all — an inherently ill-conditioned sensor pair, correctly diagnosed by computing and honestly reporting a condition number rather than chasing a nonexistent software fix
- Every diagnosis in this capstone traces back to one of three root mechanisms taught across the course: **cancellation** (Ch.4), **instability** (Ch.5), or **ill-conditioning** (Ch.6) — everything else in the course builds on recognizing which of these three is actually at play

- **Course complete** — Numerical Methods & Floating-Point Computation, 10 chapters, from `0.1+0.2≠0.3` to a fully audited, fixed, real codebase